PRESENTING MULTI-LANGUAGE XML DOCUMENTS: AN ADAPTIVE TRANSFORMATION AND VALIDATION APPROACH.

a thesis submitted to
The University of Kent at Canterbury
in the subject of computer science
for the degree
of doctor of philosophy.

By
Michael Pediaditakis
September 2006

Abstract

XML addresses several HTML shortcomings, but its underdefined processing impedes the development of adequate generic presentation models for the Web. Such models must define the parsing, validation, transformation and rendering of multi-language XML documents, according to a variety of adaptation requirements. However, most existing approaches only define subsets of this functionality and do not follow the Web design principles.

We hypothesise that generic document presentation can be achieved by utilising the presentation domain constraints and addressing the document processing problem as a whole.

This thesis focuses on the document preprocessing domain and supports our hypothesis by proposing a preprocessing framework and the XMLPipe preprocessing model. Document preprocessing is the document presentation subset that only addresses parsing, validation and transformation. The preprocessing framework establishes the necessary preprocessing functionality and enables the evaluation of XMLPipe. XMLPipe utilises the presentation domain constraints to provide generic XML preprocessing.

XMLPipe consists of an integration model, an adaptation model, a transformation model, a validation model and a binding model. The integration model utilises the presentation domain constraints to infer a multi-language document's interpretation from the interpretation of its constructs. The adaptation model proposes an extensible representation of the adaptation requirements and a method to choose the optimal processing alternative among a set of independently developed specifications. The validation and transformation models use the integration model to validate and transform multi-language documents, according to a set of adaptation requirements and a distributed set of processing specifications. The binding model establishes a distribution of the processing specifications, which is adequate for processing an open set of independently developed XML languages.

The XMLPipe document processing is demonstrated to be significantly more powerful than existing approaches and its evaluation illustrates its adequacy for the Web and the soundness of our hypothesis within the preprocessing domain. The preprocessing observations are extrapolated to confirm our hypothesis within the complete document processing domain.

...Και στην μαμά, και στον μπαμπά,
στην αδερφή στην Μαίρη,
βρε χωρίς εσάς
την είχα βαμμένη απο χέρι...

...to my family

Acknowledgements

I would like to thank:

My supervisor, David Shrimpton, for his support and guidance, his insight into all Web related areas and the whole process of undertaking a PhD, and his tolerance to my constant delays and miserably failing plans.

Professors Peter Linington and Simon Thompson for their brilliant ideas, their support in many aspects of this work and their continuous encouragement.

My family and friends for their invaluable support and infinite patience to my impressively antisocial behaviour during the last years.

Katerina, Damian, Christian, Matt and all people in Darwin H4 for their amazing research methodology insight.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

The *World Wide Web* (WWW / the Web) has been widely used as a major information medium for an unrestricted variety of information. The Web is based on the *HyperText Transfer Protocol* (HTTP)[FIG+99], the initially minimalistic *HyperText Markup Language*[RHJ99] and a set of core design principles, which have been fundamental to its development and wide deployment: simplicity, modularity, tolerance, decentralisation and no fixed set of specifications. Since the inception of the Web, a multitude of HTML extensions and new languages have been introduced, which reflect its wide deployment and increased information representation requirements. Web browser developers are constantly trying to cope with the increasing set of representations, and current Web browsers resemble more generic middleware applications than purpose specific applications that follow the Web design principles.

The *eXtensible Markup Language* (XML)[BPSMM00] simplifies the earlier *Standard Generalised Markup Language* (SGML)[ISO86], in order to establish a common markup language framework. *XML documents* can represent any information as a hierarchical nesting of *XML constructs*, which are defined by one or more *XML languages*. There are several ways to process XML documents, because they can cover any information domain, but four cross-domain processing steps are widely applicable: parsing, validation, transformation and presentation.

XML *parsing* is a necessary XML processing step, because it maps the human oriented textual XML representation to a more machine processible representation. XML *validation* ensures that the syntax of a document is correct, according to its corresponding XML languages. The core XML recommendation provides a document validation method, but there are more powerful alternative *schema languages*. An XML *transformation* maps a document to an alternative representation. The core XML recommendation does not define how to transform XML documents, but W3C and other organisations have developed several powerful transformation languages. The *presentation* of an XML document allows its browsing by a user. An application can present XML documents by either natively supporting their constructs or transforming them to a natively supported representation.

Most user initiated Web information processing results in information presentation. Therefore, a well defined XML presentation process is essential to the wide adoption of XML within the Web.

This thesis focuses on the generic presentation of XML documents for the Web. This chapter summarises the existing XML processing issues (Section 1.1) and proceeds to an overview of our hypothesis and its support (Section 1.2). Finally, sections 1.3 and

1.4 provide the core Web and XML background.

1.1 XML document presentation

Despite the significance of a well defined XML presentation process, only document parsing is currently well defined. There is no well defined generic method to present, transform and validate XML documents, notwithstanding the multitude of relevant XML technologies.

An adequate presentation processing model for XML must enable the processing of multiple XML languages for a variety of adaptation requirements. XML is a common foundation for defining new languages that can cover a wide variety of information domains. XML documents that span multiple information domains can use the constructs of multiple languages. Therefore, an XML processing model must include an adequate binding mechanism that allows the location and retrieval of all necessary processing information for each language. Additionally, a processing model that enables extensive information availability must cater for a wide spectrum of Web devices, applications and users. Consequently, a generic adaptation mechanism that adapts a document according to a set of adaptation requirements is necessary.

Moreover, document validation, transformation and presentation are necessary for a generic presentation processing model. Validation does not directly relate to the presentation, but it assists the authoring and processing of a document by ensuring its validity. Well defined validation also forms the foundation of XML processing models, because it identifies the documents that a processing model must be capable of processing. Document transformation is essential, because it can map a document to an alternative natively supported interpretation. A transformation process can also adapt a document to its optimal representation, according to a set of adaptation requirements. Finally, document presentation is essential for producing the final rendering of the document.

A generic presentation processing model must address all the aforementioned issues, but existing approaches only address them individually and they are not adequate for a generic XML processing model. Specifically, there is no well defined method to derive the processing of a document from the processing of its individual constructs. Because of this limitation, document authors cannot freely mix language constructs within XML documents. Therefore, they rely on language integration profiles, which become exponentially complex to enumerate for an increasing number of languages. Alternatively, they use languages that assimilate the constructs of other languages, which result in redundant, inconsistent and complex language specifications.

There are several existing document adaptation approaches, but the most prominent ones are restricted to predefined languages and application domains, and they are not adequate for generic adaptation of XML documents. More generic approaches, which allow an open set of languages, are not sufficiently versatile and powerful to cover the variety of Web adaptation requirements. In a similar manner, there are several prominent binding approaches, but most XML processing approaches use inline document bindings. Such document specific associations are problematic, because the processing of a document can depend on document author independent factors, such as the capabilities of the target application. Moreover, the document author is typically more interested in the document information and not in its processing.

There are several powerful transformation and validation technologies, and existing Web browsers support a versatile set of presentation functionality. However, the aforementioned processing issues impede the application of the versatile existing functionality within a generic XML presentation processing model. Specifically, the lack of generic multi-language processing has led to a multitude of languages that span overlapping domains and to predefined language integration profiles. Both languages and profiles attempt to include all necessary presentation functionality, and they result in overly complex specifications, which are not adequate for all the devices that access the Web. The development of restricted language and integration profile versions has addressed a subset of the complexity issues, but it resulted in introducing even more language specifications.

Web browsers can natively support a finite set of languages and existing browsers only support a small subset of the existing XML languages and integration profiles. Therefore, document authors avoid using new XML languages, because their native support is not guaranteed and they must use inline document processing instructions to address several binding and adaptation issues.

XML has addressed several HTML issues but failed to facilitate the wider availability of Web information, as it led to underdefined processing models and a multitude of overlapping standards.

1.2 Addressing the XML presentation issues

Addressing multidimensional problems requires the identification of their common underlying cause. In the case of XML document presentation, the aforementioned XML presentation issues result from the lack of well defined domain constraints. XML documents can combine any set of languages to represent any information, and a document's presentation can be influenced by an unrestricted variety of adaptation requirements. The lack of well defined constraints impedes establishing the necessary assumptions for creating a generic processing model.

Our hypothesis is that the presentation processing domain is sufficiently constrained to enable the development of a generic XML presentation processing model. The aim of this thesis is to support the aforementioned hypothesis by investigating the existing approaches, identifying the relevant problems, proposing an XML processing model and illustrating its adequacy for presenting XML documents within the Web.

It should be noted that this thesis focuses on document presentation preprocessing, which includes the validation and transformation processing steps, but not document rendering. Even though this thesis does not address information rendering, it extrapolates the preprocessing observations to the whole spectrum of our hypothesis.

The literature review in Chapter 2 covers the validation, transformation and presentation aspects of existing technologies, focusing both on the individual processes and on the corresponding binding, adaptation and language combination mechanisms. Chapter 2 reviews all XML presentation processing literature, including XML document rendering and existing Web browser implementations, in order to cover the whole spectrum of XML presentation processing that is addressed by our hypothesis.

Chapter 3, the first step towards supporting our hypothesis, defines all the necessary terms for expressing the XML processing concepts unambiguously and defining the presentation domain constraints. This is necessary because many Web and XML concepts are either underdefined or ambiguous.

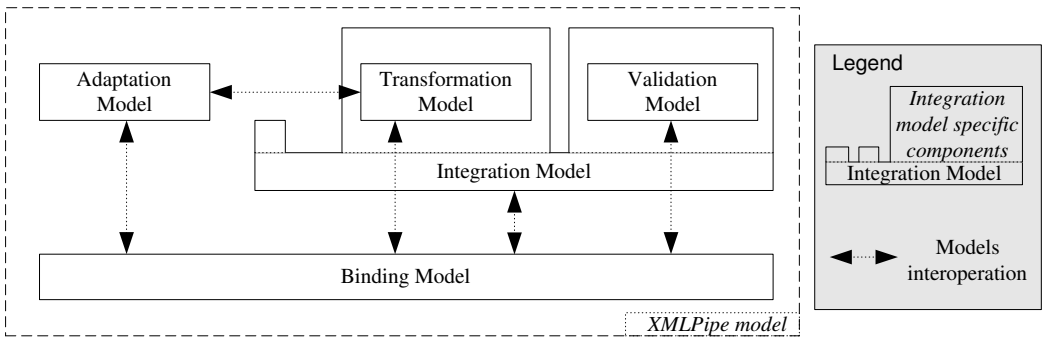

Figure 1.1: The proposed models and XMLPipe

The expressed presentation domain constraints do not provide a sufficient measure of the adequacy of an XML presentation processing model. The preprocessing framework introduced in Chapter 4 provides such an adequacy measure by using the presentation domain constraints and the Web design principles to define the necessary preprocessing functionality. The proposed framework specifies the necessary preprocessing components, their interoperation and the corresponding functionality requirements. As all framework specifications result from iteratively refining the Web and the XML design principles, the proposed framework provides a measure of how adequate is a preprocessing approach for the Web.

The preprocessing framework requires the five processing models illustrated in Figure 1.1: an integration model, an adaptation model, a transformation model, a validation model and a binding model. This thesis includes a separate proposal for each processing model and combines them into the XMLPipe processing model.

An integration model establishes a multi-language document's interpretation, according to the interpretation of its individual constructs. Chapter 5 proposes an integration model that utilises the constraints of the presentation domain and defines a classification of the XML constructs. This model enables the definition and interpretation of the valid inter-language nesting of XML constructs.

Chapter 6 proposes an adaptation model that defines an extensible and composite adaptation requirements representation and a mechanism for choosing the optimal processing specification, among a set of independently developed alternatives.

Chapters 8 and 7 introduce the transformation and validation models, responsible for transforming and validating an input document, respectively. Both models reside within the integration model, as illustrated in Figure 1.1, because they are integration model specific. They process multi-language documents by separately processing their individual single language subtrees. The transformation model interoperates with the adaptation model, in order to select the optimal transformation specification for each document portion, according to a set of adaptation requirements.

Chapter 9 introduces a binding model that combines a well defined principal mechanism and an open set of secondary mechanisms to locate the necessary XML processing information. Such a distributed binding model is essential for processing documents that combine an open set of languages.

Chapter 10 combines the above models into the XMLPipe preprocessing model,

which enables the validation and transformation of multi-language documents according to a variety of adaptation requirements. XMLPipe validation focuses on testing the semantic correctness of a document, a device-independent property of documents that have well defined interpretation. XMLPipe transformation maps a document to its optimal representation according to a set of adaptation requirements. Because the XMLPipe binding model locates and retrieves all the necessary processing information, neither the validation nor the transformation models require explicit processing information by the document author, the document user or a central processing information repository.

Chapter 10 provides support for our hypothesis by demonstrating the adequacy of XMLPipe for the Web, using the preprocessing framework. Subsequently, the feasibility of the proposed model is illustrated by a case study, which uses XMLPipe to validate and transform a document that combines multiple independently developed XML languages, according to separate sets of adaptation requirements. Since XMLPipe utilises the presentation domain constraints, its adequacy illustrates that the presentation domain is sufficiently constrained to support generic XML processing models.

Finally, Chapter 11 discusses this thesis as a whole. It discusses the support of our hypothesis, summarises the contributions of this thesis and suggests a set of future research proposals.

Throughout this thesis, we attempt to avoid ambiguous descriptions by defining all terms before their usage and by summarising all used abbreviations, terms and formal notations in appendices A, B and C, respectively.

Appendices D to F introduce reference material that is not required in the main body of this thesis: a comparison between the XML and RDF representations, a representation of all the necessary XMLPipe processing information and the case study source documents and processing information.

A subset of the work presented within this thesis has been described in three preliminary publications, which provide an overview of the XMLPipe preprocessing model[PS03a, PS03b] and our views towards a generic processing model for the presentation of XML documents[PS04]. Future publications will provide more detailed descriptions of this thesis proposals.

1.3 World Wide Web fundamentals

The *World Wide Web* (WWW / the Web) started as a means of interconnecting information to enhance group collaboration[BL98a]. The original requirement and vision for the Web was the wide availability of information:

> "Once someone somewhere made available a document, database, graphic, ..., it should be accessible (subject to authorisation) by anyone with any computer in any country. And it should be possible to make a reference –a link– to that thing, so that others could find it" [BL00]

In order to satisfy the above requirement, the *World Wide Web Consortium* (W3C) established a set of Web design principles[BL02b], which are described in Table 1.1. The adherence of the Web technologies and its fundamental concepts to the above principles has been important for its success as a communication medium[JW02]. This section describes the fundamental Web background: the communication protocols, the markup languages, the resource identifiers and the browsers.

Principle	Explanation
Simplicity	Reducing new concepts while increasing the scope of applications.
Modular Design	Break features to loosely coupled groups.
Tolerance	Strict specifications but error tolerant implementations.
Decentralisation	No central point of control, in order to limit the possibility of failure.
Test of independent invention	No restrictions on any of the processes. Individual protocols, representations and architectures must be equally applicable.
Principle of least power	Representations of minimum functionality enable reusing the same representation in multiple domains.

Table 1.1: World Wide Web design principles[BL02b]

Most Web communication uses the *Hypertext Transfer Protocol* (HTTP)[FIG+99], which allows simple and efficient communication of Web documents[BL00]. A Web resource, which can be a document, is identified by a *Uniform Resource Identifier* (URI)[BLFIM98]. The use of URIs enforces the Web design principles, because they decouple the Web from specific protocols and data representations[BL00]. A *Uniform Resource Locator* (URL) is a URI that identifies a resource via its primary access mechanism[BLFIM98], such as its network location and the corresponding communication protocol.

The Web design does not restrict resource description representations. The *Hypertext Markup Language* (HTML)[RHJ99] was created to form a common ground for all Web communications. The initial HTML version defined a minimal markup for the hypertext documents structure and not their presentation details[BL00]. However, since its inception, HTML[1] has incorporated numerous presentation specific features, in order to fulfill the increased document presentation requirements. The inclusion of such presentation features is against the fundamental design principles of HTML: minimal, common base and processible by any device. Consequently, additional languages, which range from styling enhancements to high end multimedia representations, have been developed for describing Web resources.

A document user uses a Web browser to interact with the Web information. There is no well defined set of representations that a browser must support, because of the lack of data representation constraints. Therefore, browsers have adopted extensible designs, and even the earliest browser implementations, such as WWWInda[GNSP94] and Viola[Wei94], focused on component based designs that enabled the presentation of a multitude of data representations. Current browser implementations have evolved to generic middleware platforms, instead of purpose specific applications, because of the increased presentation functionality requirements. Typical current browser functionality includes multiple media type presentation and support for generic programming languages, such as Java. Moreover, there is widespread use of extension technologies,

[1]Up to HTML 4.01. The more recent XHTML has removed all unnecessary presentational aspects.

such as plug-ins and Java applets, for incorporating additional presentation functionality. Notwithstanding the variety of supported functionality and extension mechanisms, the browser developers are constantly racing to meet the ongoing development of new languages and technologies.

1.4 Extensible Markup Language (XML) fundamentals

The *eXtensible Markup Language* recommendation (XML)[BPSMM00] aims to establish a common Web document representation. XML is extensible, because, as a metalanguage, it can be the basis of application specific languages that can describe any information. A common representation is necessary, because it provides the foundation for a common document processing layer that enables extensible browser designs, which support a multitude of languages.

Sections 1.4.1 to 1.4.3 will describe the fundamental concepts of the XML representation, the XML languages and its adequacy for the Web, respectively. Section 1.4.4 will describe the *Document Object Model* (DOM), which is a standard interface for XML information manipulation and a key component for a common document processing layer.

1.4.1 XML core

A Web language must follow the Web design principles, and it must therefore be generic, device independent and simple. Additionally, it must specify the minimum required concepts and only contain the necessary functionality. Finally, it must be both strict, in terms of the specification, and tolerant, in terms of the processing. HTML is not sufficiently generic, since it cannot be extended to represent every information domain. HTML was based on the *Standard Generalised Markup Language* (SGML)[ISO86], which is sufficiently generic, since it allows the definition of application specific languages, but its complexity is not appropriate for the Web.

XML is a simplification of SGML, and it is designed according to the Web design principles. The XML recommendation[BPSMM00] defines the core XML concepts: its syntax, a set of processing guidelines and how to define purpose specific XML languages. This section overviews the core XML concepts, but its detailed description is outside the scope of this thesis, and it can be found in the XML recommendation[BPSMM00].

The XML syntax is a device independent textual representation of trees. An XML document is a tree, and its individual nodes contain the document information. For instance, the document illustrated in Listing 1.1 contains three *elements* (lines 2, 3 and 6), an *attribute* (line 3) and a *text node* (line 4). The line 2 element is the *document element*, because it contains all other document nodes. Each XML document has one and only one document element. Each element can contain attributes, other elements and text. The tree structure of XML documents is based on the containment of the elements in a document.

An XML document must be *well formed* and may be *valid*. A well formed document does not violate the core syntactic principles of XML, such as the proper element nesting and the single document element. A valid XML document is both well formed and consistent with a syntax specification, which is expressed using a *schema language*. The XML recommendation defines the Document Type Definition (DTD), which is a minimal schema language. A DTD defines the structure of a document by specifying

```
1 <?xml version="1.0"?>
2 <element1>
3   <element2 attr1="value">
4     Text content
5   </element2>
6   <element2/>
7 </element1>
```

Listing 1.1: An XML document

its elements, their attributes and their valid nesting. There is a variety of alternative schema languages, which are described in Section 2.2.1.

An *XML language* is defined by the set of all XML documents that are valid for a corresponding syntax specification. A *language author* is the creator of an XML language, and a *document author* is the creator of an XML document. The author of a document and of its corresponding languages can be the same entity, but they generally are separate entities.

XML languages that conform to the modular design principle must only cover a well defined data domain. A document can combine multiple language constructs, in order to describe information that spans multiple data domains. The core XML syntax does not identify the separate language constructs, but the XML Namespaces recommendation[BHL99] introduces a syntax for assigning XML names to unique URI-based namespaces. XML Namespaces are the foundation for a distributed Web-based repository of unique names, and they allow the unambiguous combination of language constructs, because their origin remains explicit. A *mixed namespace document* is a document that combines constructs from more than one namespace.

The documents in Listing 1.2, illustrate the benefits of namespaces. Both documents combine a language that describes the presentation layout of a document and another language that describes furniture. Each language contains the element `table`, and the example documents describe a tabular layout of information on furniture tables. The separation between the individual table elements (lines 2, 6 and 11) is not well defined in the first document, because there are no namespaces. On the contrary, their separation is explicit in the second document, because they belong to separate namespaces.

1.4.2 XML languages

Organisations and individual language authors have used XML to create a multitude of XML languages. Table 1.2 is an illustrative subset of standard XML languages, which range from document layout to rich multimedia and interaction languages. Each language covers an application domain, using a well defined data representation and an optional processing and presentation model. Contrary to the modular design principle, the initial versions of the languages in Table 1.2 were designed as single monolithic specifications. A subset of the current specifications, such as the XHTML, SVG and SMIL, have been partitioned into a number of loosely coupled modules. Such modular specifications can apply to a wider set of devices, because a browser must only support a subset of the specified modules. Moreover, modular languages simplify the specification of language integration to mixed namespace documents. For instance, SVG animation

```
<?xml version="1.0"?>
<table>

 <row>
  <column>
   <table material="iron">
     An office table
   </table>
  </column>
  <column>
   <table material="wood">
     A kitchen table
   </table>
  </column>
 </row>
</table>
```

```
<?xml version="1.0"?>
<l:table xmlns:l="http://layout.org/"
      xmlns:f="http://furniture.org/">
 <l:row>
  <l:column>
   <f:table material="iron">
     An office table
   </f:table>
  </l:column>
  <l:column>
   <f:table material="wood">
     A kitchen table
   </f:table>
  </l:column>
 </l:row>
</l:table>
```

Listing 1.2: Mixed namespace XML documents with and without namespaces

uses the SMIL timing module. There is currently an ongoing effort to apply the same modular principles to the majority of existing specifications.

Most Web browser implementations natively support a subset of the languages in Table 1.2. Modular language design allows a browser to only support the functionality subset that can be supported by a device. However, browsers only support a finite set of languages, and an XML document might use a language that is not supported by a browser. A document transformation can transform such a document to a natively supported representation. The subsequent chapters will thoroughly investigate XML document transformations.

1.4.3 XML and the Web

XML is adequate for the Web, because it is consistent with its design principles. XML is generic, because it allows the creation of custom languages for any data domain. Its textual and concise syntax is easy to author and process, in relation to other generic languages, such as SGML. XML is device independent, because its encoding is explicitly defined, and relevant external information, such as schema specifications, is also expressed in a device independent manner. The XML recommendation specifies only the necessary core concepts, and it avoids introducing limiting device specific processing and presentation concepts. Finally, the separation of well-formedness and validity results in a syntax specification that is both strict and tolerant: schema specifications can precisely constrain the syntax of documents, but their validation is optional.

The XML adequacy for the Web does not ensure that it will successfully address the Web data representation problems. The Web is a freely evolving, ever-changing collection of data sources [Via01], and the way that the Web community uses XML cannot be controlled. A major danger with XML is that everyone can create documents, using several custom syntaxes, leading to a multitude of incompatible languages or incompatible language extensions. Nevertheless, XML namespaces can be used as a regulatory technology that averts such an outcome, because it separates language constructs and discourages uncontrolled language extensions.

Scope	Language	Description
Document layout / styling	XHTML[PAA+00]	The extensible HTML (XHTML) is an XML based representation of HTML.
	MathML[CIMP03]	The Mathematical Markup Language (MathML) is a representation of both the structure and the content of mathematical notation.
	XSL-FO[ABC+01a]	XSL Formating Objects (XSL-FO) is a vocabulary for document formating semantics, which is mainly focused towards printed media.
	CSS[BCHL04]	The Cascading Style Sheets (CSS) is a language that allows attaching style to structured documents. CSS is not an XML language, but document authors typically use it to customise the presentation of XML documents.
Graphics / multimedia	SVG[FJJ03]	Scalable Vector Graphics (SVG) is an XML representation of vector graphics, which covers their presentation, interaction and animation.
	SMIL[ABC+01b]	The Synchronised Multimedia Integration Language (SMIL) is a representation for interactive multimedia applications. It consists of separate modules that can be reused in other languages. For instance, the SVG animation is based on the SMIL timing component.
Interaction	XUL[GHHW01]	The XML User Interface Language (XUL) is a model and a language for building graphical user interfaces. For instance, the user interface of the Mozilla browser is an XUL application.
	XForms[DKMR03]	XForms evolved from HTML Forms, which is a core component of interactive Web applications. XForms provides a device neutral language for the "online interaction of a person and another, usually remote, agent"[DKMR03]
Scripting	XBL[Hya01]	The XML Binding Language (XBL) is a representation for attaching behaviour to XML elements, in a similar manner to CSS attaching style to XML elements. XBL uses device neutral scripting languages, such as ECMAScript[ECM99], to define element behaviour.
Transformation	XSL-T[Cla99b]	The XSL Transformations (XSL-T) is a language that describes transformations of XML documents to other, XML or non-XML, documents. The principal application of XSL-T is to transform XML documents that use arbitrary XML languages to documents that only use languages that are natively supported by a browser.
Generic scope	RDFXML[BM04]	The XML representation of the Resource Description Framework[KCM04], which is the foundation of the Semantic Web. It can represent any information in terms of labelled graphs that use URIs as vertices.

Table 1.2: XML languages examples

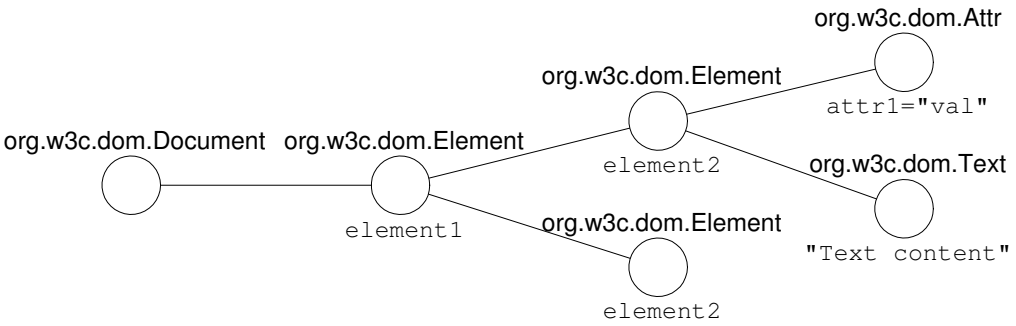

Figure 1.2: DOM example

1.4.4 The Document Object Model

The *Document Object Model* (DOM)[2] is a generic and device independent programming interface for manipulating XML data. DOM provides a well defined way for XML based applications to interoperate, and it is necessary for designing modular applications. The current DOM recommendation is DOM level 2, and there is work in progress for the DOM level 3, which focuses on better namespace and validation support. DOM level 1 specified the necessary interfaces for manipulating XML and HTML data, and it was typically used in the browser-scripting and browser-parser boundaries.

All the DOM levels are based on a "DOM core" recommendation, such as the DOM level 3 Core[HHW+04], which defines the document content access and update interfaces. The DOM interfaces expose a document as a tree structure of nodes. Each node is an instance of a subclass of the `Node` class. For instance, a node that corresponds to an XML element is an instance of an `Element` class, which is a subclass of the `Node` class. Figure 1.2 illustrates the DOM tree that corresponds to the XML example in Listing 1.1. The additional recommendations enhance or specialise the DOM core functionality. DOM level 2 (DOM-2) recommendations define a DOM based event model, a document traversal utility interface and XML presentation interfaces. The DOM-3 recommendations are not currently complete, but they appear to extend DOM-2 recommendations towards better namespace and validation support.

1.5 Concluding remarks

This chapter introduced this thesis and established the fundamental Web and XML background. XML was introduced to address the HTML shortcomings, but its under-defined processing introduced a separate set of issues. The validation, transformation and presentation of mixed namespace documents and the necessary binding and adaptation mechanisms are undefined. This thesis attempts to utilise the presentation domain constraints towards a generic preprocessing framework and a preprocessing model. The latter allows the validation and transformation of mixed namespace documents, which combine an open set of languages, using a distributed set of processing information.

[2] All DOM technical recommendations are available through the W3C Web site: `http://www.w3.org/DOM/DOMTR`.

The next chapter reviews the existing XML presentation processing literature.

Chapter 2

XML presentation processing

There are several ways to process an XML document and, within the Web, user initiated document presentation is the dominant form of XML processing. The previous chapter established the fundamental Web and XML background. This chapter reviews the XML document presentation literature, according to a top level processing model, which is described in Section 2.1. The structure of the subsequent literature review follows the proposed processing component separation. Specifically, Sections 2.2, 2.3 and 2.4 investigate the current validation, transformation and XML presentation approaches, respectively. Section 2.5 examines the extent to which current browsers implement the necessary processing functionality.

2.1 A top level presentation processing model

Successful communication of a piece of information requires a shared understanding of that information. Schema languages, which are described in Section 2.2, provide well defined and machine processible language syntax specifications. However, beyond the syntax level, XML languages do not have explicit machine processible semantics[BL02a]. As Cover states in [Cov98], the interpretation of XML documents relies on the human understanding of language specifications, and they are generally meaningless to a machine.

The foundation of a generic presentation processing model consists of both well defined machine processible semantics and their well defined association with document constructs. There is currently no generic XML presentation processing model, because of the lack of such a foundation. Section 2.1.1 will examine the core generic presentation processing requirements and Section 2.1.2 will establish the corresponding high level component separation.

2.1.1 Presentation processing requirements

The W3C organised a workshop that focused on addressing the lack of a generic XML processing model[web04]. There was no clear conclusion, but it was agreed that predefined processing step sequences, such as validation followed by transformation, are not adequate[HM01, WMF01, Heg01] for the Web, because they are only sufficient for a functionality subset of a fixed set of XML languages. Consequently, a well defined representation of the processing iterations is necessary.

An application can natively support only a finite set of languages. Therefore, the unrestricted multitude of XML languages necessitates a well defined binding between

No predefined processing sequences
Distributed binding between languages and their processing
Generic integration model for mixed namespace documents
Provision for a variety of device capabilities and user preferences
Conformance to the Web design principles

Table 2.1: Generic XML presentation processing requirements

the individual languages and their corresponding processing information. Distributed binding methods, where the owner of a resource assigns its interpretation, are more adequate for the Web than centralised ones[J+03].

An XML presentation processing model that enables XML language reuse and multi-domain information representation must support mixed namespace documents. However, in addition to the lack of well defined processing of individual languages, there is no generic language integration model, and mixed namespace document processing remains undefined[1][BL02a]. Predefined language integration profiles, such as the XHTML+SVG+MathML profile[Mas02], only provide short term solutions, because each set of XML languages requires a separate profile. Language authors continuously define new XML languages, and their interdependence becomes exponentially harder to enumerate in such profiles[Dub04]. Inline document processing semantics associations, such as the `xml-stylesheet` processing instruction[Cla99a], neither provide distributed binding nor adequately address the integration of languages, because they are as complex as the integration profiles and document-instance specific. Consequently, a generic presentation processing model requires a sufficiently generic integration model that defines the processing of mixed namespace documents, according to the processing of their individual language constructs.

Finally, an XML presentation processing model must fulfill the Web design principles, introduced in the previous chapter, and accommodate the associated variety of devices and user preferences. Table 2.1 summarises the above XML presentation processing requirements.

2.1.2 Presentation processing components

There is currently no well defined XML presentation processing model, but, as Figure 2.1 illustrates, the set of core components and external entities can be specified. Specifically, the *document author* wishes to convey some information and encodes it in an XML document, using one or more XML languages. Each XML language is created by a *language author*. The *document user* has a set of *user preferences* and uses some device, in order to interact with an XML document presentation, in the optimal way for the corresponding user preferences and device capabilities.

The presentation of an XML document consists of the creation of a data set and of its rendering to a specific device[Heg01]. A *presentation component* is a necessary XML browser component and is responsible for the XML data set rendering. The initial data set creation steps are the document parsing and the optional document validation,

[1]W3C TAG Issue: mixedNamespaceMeaning-13: October 2003.

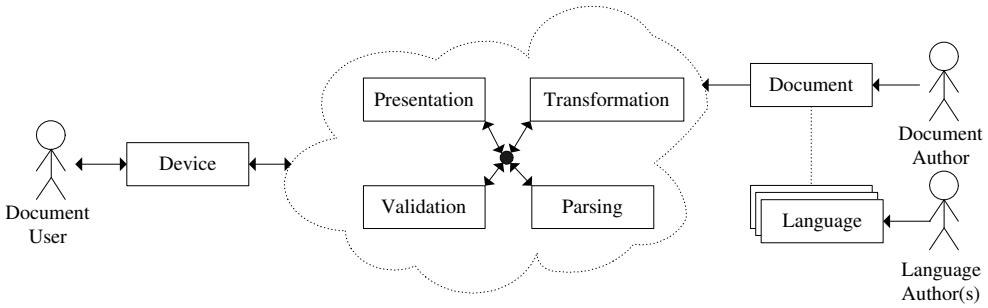

Figure 2.1: Presentation processing components and external entities

which are respectively performed by the *parsing component* and the *validation component*. A document can contain languages that are not natively supported, because presentation component can natively render a finite set of XML languages. Therefore a transformation component is necessary for mapping unsupported constructs to the corresponding natively supported constructs.

The subsequent chapters of this thesis will use a formal notation that assists the concise communication of the proposed concepts. This chapter will gradually introduce the necessary notation, which will be precisely defined in subsequent chapters. $\mathcal{L}_p$ will denote the set of a presentation component's natively supported languages. An XML document d contains constructs from the languages in $\mathcal{L}_d$. When $\mathcal{L}_d - \mathcal{L}_p \neq \emptyset$ the *transformation component* is responsible for mapping the constructs of every language $L \in \mathcal{L}_d - \mathcal{L}_p$ to their corresponding constructs of languages in $\mathcal{L}_p$.

A presentation processing model must define how the above components interoperate to create the optimal presentation of a document, according to a combination of a document, a user and a device. Currently, only document parsing is well defined for all XML documents. The rest of the components and their interoperation are undefined, and the only existing presentation processing model agreement is that there must not be any predefined processing order. Consequently, the individual processing components, illustrated in Figure 2.1, appear to loosely interoperate, and there is no concepts of predefined ordering, such as processing layers or iterations.

The lack of generic processing models did not inhibit the introduction of a multitude of approaches for the individual processing components. The remainder of this chapter will discuss the existing literature for each processing component.

2.2 Validation

The XML recommendation requires that a document is well formed, but document validity is optional. The distinction between well formedness and validity assists the development of XML parsers, efficient XML processing, and liberal evolution of schema languages. However, validation is an important component of an XML presentation processing model. Specifically, XML processing components that use the foundation of a validation component are easier and less error-prone to develop: they do not require ad hoc conditional validation logic, which is necessary, because the decentralised nature of the Web does not allow a priori document validity assumptions. Additionally, if

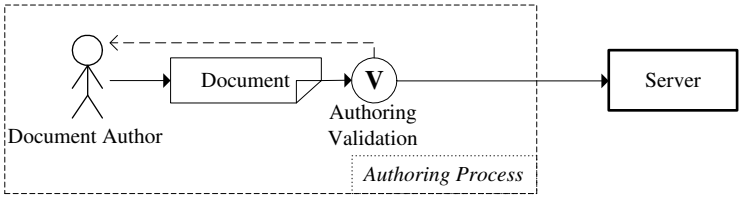

Figure 2.2: Validation during document authoring: The V-circle represents the validation process and the dashed arrow represents the validation feedback.

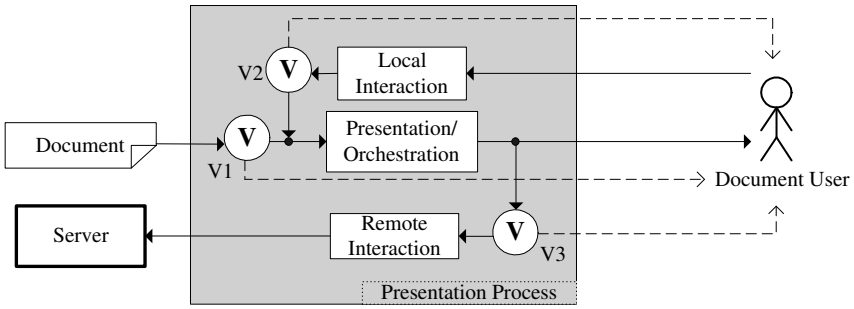

Figure 2.3: Validation during document presentation

validation is a common practice, language authors are more likely to provide language schemas, as opposed to ambiguous descriptive syntax specifications.

Validation processing applies to both document authoring and presentation. The document author can perform *authoring validation* to ensure the validity of a document. The presentation process can validate a document before its presentation (*pre-presentation validation*), and it can also validate the results of a process or user interaction during the presentation (*presentation validation*). Figure 2.2 illustrates the authoring validation, where the document author receives the validation process feedback. Figure 2.3 illustrates a processing validation example, where V_1 validates the input document, V_2 validates the result of a user interaction and V_3 validates a document prior to its submission. The XForms[DKMR03] processing model is an example of V_2 and V_3.

Autonomous validation processes use schema specifications to independently test the validity of documents. In contrast, *integrated validation* approaches use schemas to perform compile time analysis that ensures the validity of document operations. For instance, the *XML transformers typechecking* approach[MSV00] ensures the validity of a transformer's output. Integrated validation is efficient, but it requires strict component design that limits its application. For instance, in the above validation examples, only V_3 can use integrated validation, because of the well defined communication between components of the same application.

Section 2.2.1 reviews the existing integrated and autonomous validation approaches, and Sections 2.2.2 and 2.2.3 focus on the mixed namespace document validation and

the validation information binding, respectively.

2.2.1 Validation approaches

Autonomous validation approaches use either *grammar–based* or *rule–based* schema languages to describe the corresponding validation rules. The former define validity using a document grammar, and the latter use structure assertions.

The *Document Type Definition* (DTD) is a part of the XML recommendation, and it was the first XML validation approach. DTD is a minimal grammar–based language that can express element containment, attribute containment and unique identifiers. However, it does not cover current XML applications[Via01], because it does not support namespaces, data types and expressive element and attribute restrictions. Consequently, a multitude of alternative schema languages have been developed, such as the W3C XML Schema recommendation[TBMM01, BM01, Fal01]. XML Schema addresses most of the DTD shortcomings, and it includes extensible element and attribute data types, namespaces support and a modular syntax. Nevertheless, XML Schema cannot express attribute interrelation constraints, and the use of subset of its concepts, such as `any` and `all`, is overly restrictive. XML Schema does not conform to the XML minimalistic nature, because of its lengthy specification and its mutually redundant concepts, such as inheritance and substitution groups.

The most prominent current schema language is Relax NG[CM01], which is based on an XML formal model and defines grammars using element, attribute and text patterns. It provides more control over attributes than XML Schema and, even if it does not contain a data model, it can reuse existing data models, such as the XML Schema data model[BM01]. Other grammar–based approaches include the *Unified Constraint Model for XML* (UCM)[FKS01], which uses a minimal core of recursive regular expressions, and examplotron[Vli03], which uses document examples to infer the syntax of a language.

Rule–based schema languages offer several attractive features, but they tend to lack important features of the grammar–based approaches. For instance, Schematron[Jel03] uses XPath[CD99] structure assertions to provide very expressive element and attribute interrelations, and as an XSL-T meta-stylesheet, it has a device independent implementation. However, it lacks important string data types, type defaults and inheritance features. Similarly, the Document Structure Description (DSD) [KMS00, Mol03] uses a generic model of contextual regular expressions tests, but it lacks powerful restrictions between sibling nodes. Rule and grammar–based approaches complement each other, and combined approaches, such as SchemaPath[MCV04], which introduces rule–based constraints to XML Schema, result in a powerful combination of both models.

Integrated validation approaches validate the document modification processes, instead of the documents. XDuce[HP03] is an XML processing platform that treats element types as object oriented types, and the associated modification methods enforce document validity. Instead of controlling the document processing, other approaches control the document modification rules. For instance, [KSR02] describes the modification of existing transformation specifications, in order to ensure that they always produce valid documents.

Two studies[LC00, MLM01] compare the existing validation approaches either by their set of supported features or by comparing them to a generic formal model. DTD is the least expressive approach and Relax NG, and XDuce cover most of the necessary validation functionality spectrum. However, no existing approach is more expressive

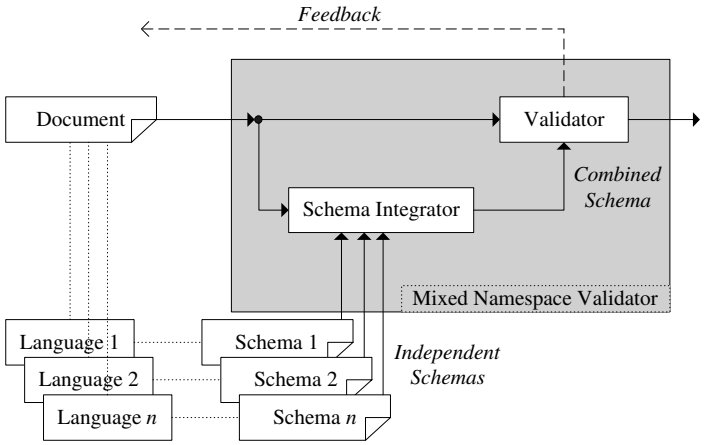

Figure 2.4: A generic schema integrator

than the others, and generic XML validation cannot rely on a single specification. On the contrary, it must either allow a multitude of validation technologies or use a more generic abstract model, such as the formal model described in [MLM01].

2.2.2 Mixed namespace validation

Apart from the DTD, all introduced validation approaches are namespace aware: they allow the association of namespaces with schemas and the use of namespace qualified elements and attributes. However, such namespace awareness is not sufficient for mixed namespace validation. Language authors develop XML languages independently, and there is a multitude of independently developed schemas. The relationship between the language L of a mixed namespace document, which combines constructs from languages $L_1, L_2, \ldots, L_n$, and the individual language specifications is not well defined. This is a consequence of the lack of a generic method to interpret mixed namespace documents[BL02c]. Well defined mixed namespace document interpretation is necessary for combining the schema specifications of any languages $L_1, L_2, \ldots, L_n$ to a mixed namespace language L for a document d where $\mathcal{L}_d = \{L_1, L_2, \ldots, L_n\}$ (Figure 2.4).

Current mixed namespace processing applications either introduce purpose specific integration profiles or abandon validation altogether, because of the lack of a generic integration model. X-Smiles[PHV02] is an XML browser that supports mixed namespace documents, but it does not perform any validation prior to the presentation. W3C defines a custom integration profile[Mas02] for combining the XHTML, SVG and MathML languages, but such custom integration profiles are limited and not adequate for the Web, as described in Section 2.1.1. Moreover, the W3C XHTML modularisation recommendation[AAB$^+$01] focuses on design principles for the creation of integration profiles, as opposed to the development of a generic integration model.

The *Namespace Routing Language*(NRL)[Cla03] and the *namespace-based validation dispatching language* (NVDL), which is a part of the Document Schema Definition Languages (DSDL)[ISO04] multi-part validation standard, attempt generic mixed namespace validation. NRL defines a syntax for associating namespace URIs with

```
1  <?xml version="1.0"?>
2  <l:table xmlns:l="..."
3     xmlns:f="...">
4
5   <l:row>
6    <l:column>
7     <f:table>
8      An office table
9     </f:table>
10   </l:column>
11   <l:column>
12    <f:table>
13     A kitchen table
14    </f:table>
15   </l:column>
16  </l:row>
17 </l:table>
```

(a)

```
1  <?xml version="1.0"?>
2  <l:table xmlns:l="..."
3     xmlns:f="...">
4
5   <l:row>
6    <f:table>
7     An office table
8    </f:table>
9    <l:column>
10   </l:column>
11   <l:column>
12    <f:table>
13     A kitchen table
14    </f:table>
15   </l:column>
16  </l:row>
17 </l:table>
```

(b)

```
1  <?xml version="1.0"?>
2  <xhtml:table
3     xmlns:f="..."
4     xmlns:xhtml="...">
5   <xhtml:tr>
6    <xhtml:td>
7     <f:table>
8      An office table
9     </f:table>
10   </xhtml:td>
11   <xhtml:td>
12    <f:table>
13     A kitchen table
14    </f:table>
15   </xhtml:td>
16  </xhtml:tr>
17 </xhtml:table>
```

(c)

Listing 2.1: Erroneous NRL and NVDL validation example

schema specifications and the corresponding mixed namespace validation process. The latter allows the combination of a multitude of schema languages by separating documents into single namespace subtrees and validating them independently. Moreover, a namespace can be associated with a sequence of schema specifications, in order to allow the combination of multiple schema languages for defining the syntax of an XML language. Finally, NRL introduces user defined execution modes that allow context dependent subtree validation, which accommodates for multi-namespace schema specifications, such as the proposed integration profiles. NVDL is similar to NRL, and it validates individual document subtrees using namespace to schema associations. Both NRL and NVDL store the necessary associations in a single file.

Both NRL and NVDL use a minimal integration model: if the individual subtrees of a document are valid, then the document is also valid. Such a model does not cover several integration cases. For instance, consider the integration example in Listing 2.1(a). The subtrees of the individual languages are valid and therefore the document is also valid. However, in Listing 2.1(b), the information in lines 6–8 appears in the table, but it is outside of any column. Such a structure is not valid, according to the semantics of a table layout construct. Nevertheless, since all subtrees are valid, both NRL and NVDL will consider Listing 2.1(b) as a valid document. Listing 2.1(c) is an alternative where the document author uses XHTML tables. The XHTML syntax only allows tables within the context of a `body` element; therefore, NRL and NVDL will produce validation errors. Nevertheless, the document is semantically equivalent to the first valid example.

Unlike the above approaches, there are generic languages that allow well defined information integration. RDF is the foundation of the semantic Web, and it can represent any type of information, in a similar manner to XML. However, the processing of RDF documents that combine multiple RDF syntaxes is well defined. Therefore, a possible solution to the XML integration problems might be based on RDF.

A comparison between the XML and RDF versions of a structured document, which

is included in Appendix D (page 299), illustrates the benefits and drawbacks of such an approach. An initial observation is that the RDF version is more than twice the size of the XML version, more difficult to author and more difficult to comprehend. These are consequences of the principal information focus of RDF, as opposed to the structure focus of XML. Specifically, the RDF document contains more information than the XML document, it is more precise and it clearly defines the associations between the individual information items. In contrast, XML is more focused on the information structure and, as Berners-Lee states in [BL98b], it provides order, which makes more sense and is easier to comprehend than the RDF unordered sets of statements.

Information integration depends on the association between different pieces of information, and RDF integration is straightforward, because such associations are explicit. However, neither substituting XML with RDF nor using RDF-style information associations within XML are adequate solutions to the XML integration problems. XML and RDF belong to separate information representation domains, which fulfill different sets of requirements. RDF is primarily machine oriented, and it is not easy to author and comprehend. XML focuses on the order of information, instead of the information associations, which results in a more human oriented syntax that is a major factor in its success. Both of the above solutions would result in a representation that does not fit within the human oriented scope of XML. Consequently, this thesis will not cover presentation approaches that are specific to either the RDF or the semantic Web and cannot be adapted to the XML information representation model.

2.2.3 Schema binding

Schema binding concerns the explicit or implicit association of the necessary schema specifications for validating a document. The validation approaches introduced in Section 2.2.1 either do not specify such associations or propose inline document associations. For instance, a document can either have an inline DTD specification or refer to an external DTD resource, using a declaration at the beginning of the document. XML Schema defines attributes, within the "document instance" namespace, that can provide schema location hints to the validator. NRL and DSDL approaches introduce namespace to schema associations, but their schema binding is absent, because the associations reside in a separate file, and there is no well defined way to locate it.

Explicit document associations ensure the availability of validation information, but they delegate the validation information specification responsibility to the document author. However, the document author typically focuses in the document content and not in its processing. Moreover, "the design choice for the Web is that the owner of a resource assigns the authoritative interpretation of representations of the resource"[J+03]. Consequently, the *language author*, who is the "owner" of a language, should be responsible for specifying its processing information, which includes the validation information.

Namespaces can enable the wide availability of processing specifications that are developed by language authors. Namespaces provide unique XML construct identification, which is the basis of any form of association. Language authors typically control their chosen namespace URIs and any URI-associated Web resources, such as Web documents, and they can use them to associate a language to its human and machine processible information. Such an approach enables the distributed specification of language processing information and makes the Web a "distributed registry of languages"[BL03].

There is currently no consensus on the association between namespace URIs and the corresponding human and machine processible language information. If the namespace URIs are also URLs, they enable well defined resource retrieval by a browser. Schema specifications were initially considered as the adequate resource to associate with a namespace URI. However, schemas only provide syntax information, and they are not adequate for human consumption. [BL02d] proposes embedding RDF-based language processing information within XHTML human readable language descriptions, but there is currently no corresponding well defined RDF vocabulary.

The Resource Directory Description Language (RDDL)[BB02] is an XML syntax that specifically addresses the representation of namespace related resources. RDDL extends XHTML to allow the incorporation of links to machine processible resources within human readable XHTML descriptions. Each link has a "nature" and a "purpose". The *nature* specifies the type of the linked resource, such as an XML Schema specification type, and the *purpose* further refines the usage of the resource. There is a predefined collection of the most commonly used natures and purposes, but language authors can add their own. RDDL is not currently widely used, but it provides an easy transition from the existing plain XHTML descriptions and allows the use of several resource types. Therefore, it is a prominent approach for the association between namespace URIs and language processing information.

2.2.4 Validation summary

Document validation is an important processing step during both document authoring and presentation. The existing set of validation approaches and schema languages covers a wide range of functionality, but no individual approach covers all existing functionality. Therefore, a generic XML presentation processing model must enable the incorporation of the existing multitude of validation approaches. Mixed namespace document validation is currently based on several short term solutions, such as integration profiles. NRL and DSDL proposed a minimal language integration model, but it is not sufficiently generic for many integration cases. Finally, most approaches either provide document based or no validation information binding, instead of using namespace URI based associations. RDF and RDDL have been proposed as alternatives for encoding such associations.

2.3 Transformation

Early adoption of XML in Web browsers was impeded by the lack of a well defined XML presentation processing model. The CSS recommendation[BCHL04] was initially used to attach style information to XML constructs, but CSS is restricted by the initial document structure, because it cannot transform a document. The *XSL Transformations* (XSL-T)[Cla99b] address the CSS restrictions by separating the document presentation from its structure and transforming XML documents to other XML or textual representations. Transforming an XML document to a natively supported representation enables its presentation.

A *transformation* maps the constructs of a language L_1 to the constructs of a language L_2, according to an optional external input. The external input can be any relevant information, such as an XML document or a set of transformation parameters. $\mathcal{T}_{L_1}^{L_2}$ will denote the set of all transformations that map documents of L_1 to documents

of L_2. $d_1 \xrightarrow[I]{T} d_2$, or $d_1 \xrightarrow{T} d_2$ for insignificant external transformation input, denotes the mapping of d_1 to d_2 according to a transformation T and an external input I. If $T \in \mathcal{T}_{L_1}^{L_2}$ and d_1 is a valid document of L_1, then d_2 is a valid document of L_2.

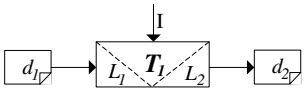

Figure 2.5: Transformation symbol

Figure 2.5 represents the mapping $d_1 \xrightarrow[I]{T_1} d_2$ of d_1 to d_2, according to the transformation $T_1 \in \mathcal{T}_{L_1}^{L_2}$. A transformation can have multiple inputs and produce multiple outputs, which are not necessarily restricted to XML. However, XML can represent any information, and multiple inputs and multiple outputs can be incorporated into a single XML document. Moreover, any non-XML information can be represented by an equivalent XML document. Therefore, without reducing the generality, this thesis will only consider single XML input and output transformations.

XML transformations can apply in multiple points of several applications, such as generic data manipulation, content styling, Web publishing, and device specific content adaptation. For instance, consider an XML document that lists a set of computer science literature authors and the example transformation steps, illustrated in Figure 2.6. T_1 can select a subset of the authors, and T_2 can transform the resulting author data to a language, such as XHTML, which is natively supported by the client browser. T_3 can perform pre-presentation processing, such as converting XML Links to their equivalent XHTML representation. During the presentation, T_4 can modify the presented data, in order to generate presentation effects, such as animation. Finally, if the document user can edit the presented information, T_5 can transform the user input to an XML representation adequate for transmission to a server. The above transformation steps can be freely distributed, since the inter-transformation communication uses XML, which is device independent. The illustrated transformation steps illustrate an example distribution over a server, a client, and an adaptation proxy. However, different configurations can be used; for instance, all the processing can take place in the client.

The remainder of this Section (2.3) investigates the existing XML transformation literature. Section 2.3.1 summarises existing transformation approaches that can be enhanced by the transformation pipeline approaches in Section 2.3.2. Sections 2.3.3, 2.3.4 and 2.3.5 focus on content adaptation using transformations, binding of transformation information for mixed namespace documents and the interoperation between transformation and validation, respectively.

2.3.1 Transformation approaches

XSL-T[Cla99b] was the first XML transformation language and it enables mapping XML documents to other, XML or non-XML, documents. XSL-T uses an XML syntax for its transformation rules, which consist of a set of output templates that are executed recursively, according to either the structure of the input document or to explicit template invocations. *XPath*[CD99] is an integral part of XSL-T, because it provides the

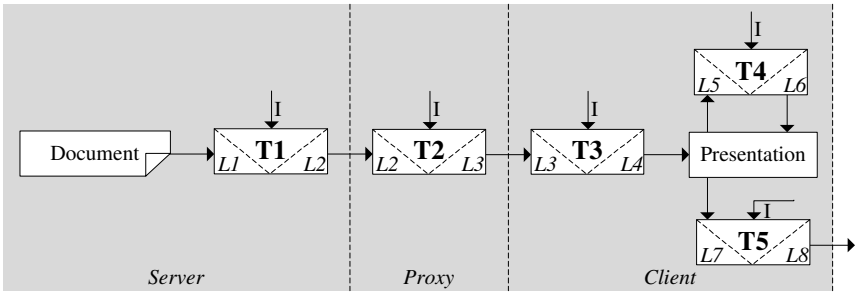

Figure 2.6: Multiple transformation applications during document presentation

necessary syntax for referring to source document constructs and defining functional data computations. XSL-T has been widely adopted, but it introduced multiple practical problems, such as the lack of data types, the lack of multiple inputs and outputs and the restricted error handling and namespace support. The recently standardised XSL-T 2.0 addresses these problems.

The main criticism of XSL-T is the disproportional complexity of its syntax to the complexity of the defined transformations, which is particularly apparent for minimal transformations, such as removal and copying of elements[ET01]. SXSLT[KK03] is an alternative approach that claims to have a more concise syntax and better formal properties. SXSLT syntax is based on Scheme[KCR98], and it is more concise than XSL-T. However, using non-XML syntaxes for transformations is against the concept of a common data representation, and it can complicate several transformation applications, such as meta-stylesheets. Meta-stylesheets are transformation specifications that produce other transformation specifications.

Additional transformation approaches include the XDuce[HP03] transformations and the Streaming Transformations for XML(STX)[CBN+03]. The former defines transformations as operations on XML types, as introduced in Section 2.2.1, and pattern based production rules. XDuce allows both declarative and imperative transformation specifications and their validation at compilation time. STX focuses on efficient transformation application, and it minimises the memory and processing requirements by manipulating sequences of XML structure events, as opposed to DOM trees. The streaming nature of STX reduces its expressiveness, but the use of look-ahead and history techniques permits the support of a substantial subset of XSL-T functionality.

Finally, in addition to generic transformation approaches, the processing of other XML languages can also result in document transformation. A typical example is the XML Inclusions language[MO03], which defines a syntax for including external content to an XML document. The external content inclusion results in document modifications. Most such languages can be implemented by the above generic approaches; therefore they only provide a functionality subset of the generic approaches.

2.3.2 Transformation pipelines

The interoperation between multiple transformation steps is not well defined, because each of the aforementioned approaches uses separate transformation rule syntaxes and processing models. For instance, consider the example illustrated in Figure 2.6, where

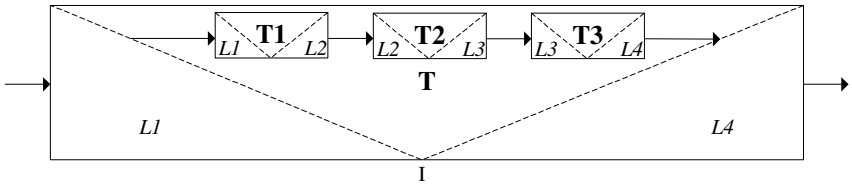

Figure 2.7: Transformation pipeline example

T_3 is responsible for mapping the XML Links constructs to their XHTML equivalents. Processing additional languages would either require a combination of separate transformations, or a single transformation that combines their functionality. The former modular approach is preferable, because it allows complex transformation specifications through division into simpler ones, and it combines the capabilities of multiple approaches. *Transformation pipelines* enable the interoperation of separate transformation technologies, and they provide a declarative approach for connecting multiple transformation steps, which is less error-prone than the alternative of using generic scripting.

Existing transformation pipeline approaches fit into the single framework of describing the document flow between four component types: *sources*, *transformations*, *mergers* and *sinks*. The sources and the sinks are responsible for the initial input and the final output of the document, and they provide the input/output interface of a pipeline. The transformations are the core processing components, since they modify the XML document. Mergers exist in a subset of the existing pipeline proposals, and they are purpose specific transformations that use multiple document inputs. A pipeline is also a transformation. Figure 2.7 illustrates a composite transformation that corresponds to the sequential pipeline composition of the first three transformation steps illustrated in Figure 2.6. Specifically, $T_1 \in \mathcal{T}_{L_1}^{L_2}$, $T_2 \in \mathcal{T}_{L_2}^{L_3}$ and $T_3 \in \mathcal{T}_{L_3}^{L_4}$. The resulting pipeline T is a transformation: $T \in \mathcal{T}_{L_1}^{L_4} \in \mathcal{T}$

XPipe[McG01] is an early pipeline approach, which aims to reduce complexity by composing transformations out of small and reusable components. XPipe architecture is based on four layers. The first layer contains an extensible set of reusable components that perform simple transformations. The second layer combines these components into reusable pipelines, which provide enhanced transformation functionality. XRigs, which is the third layer, uses the pipelines to build more complex structures, such as transformation loops and duplexers. Finally, XGrid, the topmost layer, focuses on transformation clustering for efficient execution using parallel processing. XPipe was an ambitious project, and it had the potential to allow efficient execution and modular design of arbitrary complex transformations. However, there has been only a partial implementation of the first two layers. XPipe was subsequently subsumed within a commercial solution, and details of further developments are not publicly available.

Cocoon[Maz02] is a widely deployed approach that uses pipelines for Web publishing. Cocoon offers an extensive library of predefined reusable transformations, which are the basis for pipelines that perform the most common Web publishing tasks. Cocoon and other similar approaches, such as the *Lazy XML processing* proposal[NSL02],

Device type	Examples	Description
generic	`http://www.example.org/document.html`	The original URI

mobile	`http://www.example.org/document.wml`
	
	`http://www.example.org/mobile/document.html`
		Use of file extension
		or
		path component

printer	`http://www.example.org/document.pdf`
	
	`http://www.example.org/print/document.html`
		Use of file extension
		or
		path component

Table 2.2: Example of implicit device capabilities information within URIs

target server side transformations, and they use a data-pull model: the last step of the pipeline controls the data flow. On the other hand, push-based approaches, such as the W3C XML Pipeline[WM02], use models similar to tools like *make* (part of POSIX[Por04]). Such models organise the individual transformation steps according to separate processing targets, and the document data are pushed through them.

In addition to integrating existing approaches, *Transmorfer*[ET01] proposes a transformation pipeline model within a complete transformation language. Transmorfer allows using regular expressions for more concise transformation definitions and provides recursive application of transformations. The latter simplifies the processing of naturally recursive transformations. For instance, an XML Inclusions processing transformation can introduce new content that also contains inclusion constructs. Transmorfer will recursively invoke the same transformation, until all of the constructs have been processed.

2.3.3 Transformations for content adaptation

Transformations can adapt XML documents to the characteristics of a variety of devices. Cocoon offers some primitive adaptation support by associating the transformation pipelines with URIs, because the current use of URIs has device capabilities implications that assist the identification of the basic characteristics of a client device. For instance, consider the URI examples in Table 2.2. The first URI does not have any device capabilities implications. The second and third imply that the associated resource targets are mobiles, because of either the "wml" extension or the "mobile" part in the URI. In a similar manner, the last two URIs imply an associated resource that is optimised for printing. The Cocoon associations between URIs and pipelines enable a Web server to process the same source document using separate pipelines, which correspond to separate URI implied device capabilities.

The benefits of such adaptation approaches are limited, because the URI implicit adaptation information is not well defined and restricted. Additionally, such device dependent URIs are against the authoring principles for the Web[GFMS03], because a URI identifies an information resource and not a device dependent information representation. Nevertheless, they enable simple adaptation that does not require additional client-server cooperation, and there are currently no other generic multi-language transformation approaches that offer more powerful content adaptation.

In contrast, the *device independent authoring* domain includes several proposals for adapting a document according to a set of device capabilities and user preferences. However, they do not allow generic document transformation, because they either restrict the document languages set or rely on a static presentation component that

closely interoperates with the transformation process. The investigation of device independent authoring approaches is included in a subsequent section (2.4), because the restricted set of languages and the presentation component specific adaptation are more relevant to the presentation component.

The application of transformations for content adaptation extends further than the introduced XML transformations. Specifically, Web content may also use binary data, such as images and videos, which also require adaptation to a variety of devices. [PZB02] proposes a binary pipeline, which combines binary data transformations that adapt binary content to a variety of devices. The adaptation of XML document associated binary data is necessary, but it is outside the scope of this thesis, which is primarily concerned with XML processing.

2.3.4 Binding and mixed namespace transformations

The stylesheet processing instruction[Cla99a] is the dominant way of associating a transformation definition with a document. Such document based approaches share most of the document based validation binding problems described in Section 2.2.3, notwithstanding the weaker relationship between transformations and XML languages. Specifically, the document author should not be required to specify the document processing. Language processing information should be associated with the individual languages, instead of the documents, and transformation specifications are a partial language processing definition. Cocoon provides the only alternative transformation binding mechanism, which associates transformation specifications with URIs and URI patterns. The Cocoon binding separates the document from its processing, but the associations remain document specific, and document modifications can require extensive reconfiguration of the associated transformation pipelines. Consequently, there is still an implicit relationship between the document author and the final document processing.

The majority of transformation approaches allow namespace aware transformations that use namespace qualified constructs. However, there is currently no proposal for associating individual namespaces to reusable transformations that can be combined for processing a mixed namespace document. The lack of such proposals is a consequence of the current principal perception of transformations as an independent step of XML document processing and not as a way to define the processing of XML languages.

2.3.5 Interoperation with validation

As described in Section 2.1.2, an XML presentation model must define the interoperation between its individual processing components. The output of a transformation process $T \in \mathcal{T}_{L_1}^{L_2}$ is well defined, if its input document d_1 is a valid document of L_1. The output d_2 of T is guaranteed to be a valid document of L_2, if the transformation process behaves correctly and has no errors. The interoperation of the transformation and the validation processes is necessary, in order to ensure well defined processing behaviour and to assist the identification of implementation errors.

In addition to transformation processing steps, the W3C XML Pipeline approach also allows validation steps within the pipelines. Additionally, validation approaches that are implemented as transformations, such as the previously introduced Schematron, can be incorporated in all transformation pipeline approaches. More fine grained

transformation and validation interoperation occurs within integrated validation processing models. For instance, the result of any transformation in XDuce is a priori known to produce valid results, and there is no need for additional runtime validation. Similarly, the type checking approach described in [MSV00], ensures that the result of a transformation $\mathcal{T}_{L_1}^{L_2}$ will be a valid document of L_1, if the input is a valid document L_1.

2.3.6 Transformation summary

Transformations are important for an XML presentation processing model, and they have several applications in document styling, Web publishing and content adaptation. There are several generic transformation languages, where XSL-T is the most widely used, notwithstanding that it is relatively more complex than the alternatives. Transformation pipelines enable the combination of transformation steps to form more expressive, easier to develop and more reusable processing components.

Current approaches that bind transformation information to XML constructs are problematic, because they delegate the document processing responsibility to the document author. The lack of language based associations results in the lack of generic mixed namespace transformation models.

Content adaptation is a prominent application for transformations, but apart from implicit and ambiguous URI based associations, there are no generic approaches that adapt XML content to the variety of devices and user preferences that will inevitably be required for the Web.

Finally, the interoperation between the validation and transformation processes can be expressed by incorporating validation steps within transformation pipelines. Additionally, integrated validation approaches allow the validity to be a priori known, without requiring a separate runtime validation.

2.4 Presentation

A document user uses a Web browser, which runs on a *target device*, to interact with the presentation of an XML document. The presentation component is the interface between the XML data, the document user and the target device, and it is responsible for adequately presenting the XML document according to the document author intentions, the *target device capabilities* and the *document user preferences*.

The presentation component can natively support a finite set of XML languages, the *native presentation languages* set $\mathcal{L}_p$. As Figure 2.8 illustrates, the other processing model components interoperate for mapping the source document to its natively supported interpretation. In a similar manner to the transformation and validation components, the presentation component benefits from mixed namespace documents and an extensible native presentation language set. The former allows better $\mathcal{L}_p$ utilisation, and the latter is necessary to accommodate the ever increasing Web presentation requirements. Additionally, the presentation component must simultaneously fulfill the, possibly conflicting, document author requirements, document user requirements and target device capabilities.

Sections 2.4.1, 2.4.2 and 2.4.3 focus on the $\mathcal{L}_p$ set: its members, its extensibility and the corresponding mixed namespace support. Section 2.4.4 investigates both presentation and transformation approaches for adapting the document presentation, according

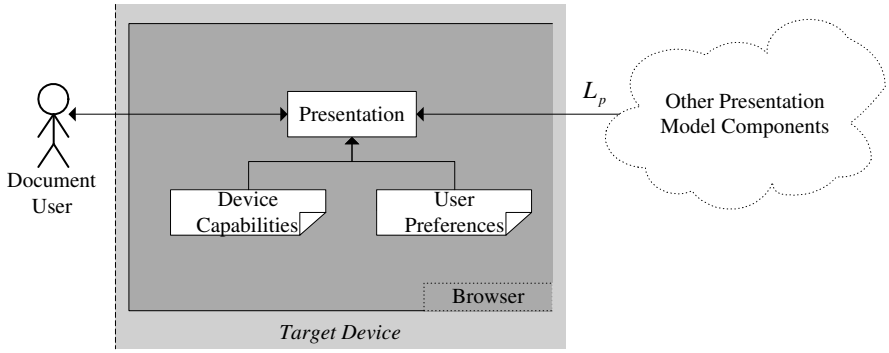

Figure 2.8: The presentation component as an interface between the target device, the document user and the document author.

to the user preferences and device capabilities. Finally, Sections 2.4.5 and 2.4.6 introduce several imperative scripting and declarative constraint approaches, which can enhance the functionality of the presentation component.

2.4.1 Native presentation languages set $\mathcal{L}_p$

The members of $\mathcal{L}_p$ significantly influence a presentation model's capabilities, because $\mathcal{L}_p$ represents its interface, and all presentation information must be encoded using the languages in $\mathcal{L}_p$. Before the advent of XML, the presentation model capabilities were restricted by a limited set of Web languages, and an $\mathcal{L}_p$ that covered the functionality of HTML and CSS was sufficient. However, XML content can require arbitrary complex presentation, and the members of $\mathcal{L}_p$ must cover a wide presentation functionality spectrum that is difficult to enumerate.

Existing W3C recommendations include a multitude of presentation oriented XML languages that are candidate members of an adequate $\mathcal{L}_p$, because they cover a wide range of presentation functionality. For instance, HTML and CSS have covered the majority of the document layout and hyper-linking uses, over the last decade. Extensive use does not in itself prove sufficient coverage of a presentation area, but it does give an indication of sufficient coverage for the most common requirements. However, the illustrated sufficiency has been maintained by continuous extensions that have resulted in complex and feature saturated languages, which, as mentioned in Section 1.3, are against the minimalistic nature of the Web.

XML allows the creation of purpose specific languages, and it can address the problem of overly complex languages, but XML languages are evolving in a similar way to HTML. For instance, SMIL 1.0 contained the core multimedia integration functionality in a thirty pages recommendation. The SMIL 2.0 recommendation[ABC+01b] attempted to cover most necessary Web multimedia functionality, and it resulted in a twenty fold recommendation size increase. Similarly, SVG started as simple vector graphics syntax, but evolved to a very broad language, which includes animation, text layout, streaming media and presentation of arbitrary XML content. Such languages are versatile, but they do not conform to the minimalistic nature of the Web and most include features that not all devices can support. As Allen states in [All04], the Web needs simple building blocks that can be combined to provide extended functionality.

Functionality categories	Hardy[Har04]	Allen[All04]	Birbeck[Bir04]
Layout	document layout	generic layout	styling, generic rendering
Multimedia	integration, animation	media support animation	generic rendering
Interaction	Interface components	Input events	dynamic document model, system module, communications.
Behaviour	code/data binding	binding constraints	Events, validation
Extensibility	extensible architecture	building blocks	generic presentation model
Adaptation	resolution independence	views	object broker

Table 2.3: Proposed Web content presentation building blocks

Consequently, there is an ongoing effort to partition existing languages into a set of loosely coupled modules.

Hardy[Har04], Allen[All04] and Birbeck[Bir04] propose three separate sets of the necessary presentational building blocks for the Web. As Table 2.3 illustrates, all proposals cover the same functionality categories: document layout, multimedia, interaction, behaviour specification, extensibility and adaptation for a variety of devices. The core difference between the three proposals is the granularity level of the proposed building blocks. For instance, [All04] describes individual presentation components, but [Bir04] describes a higher level generic presentation model. All proposals contain language examples, but they define the necessary functionality using sets of features, instead of specific XML languages. Such descriptions enable well defined functionality categories, which do not depend on language specific features.

None of the above proposals proves the sufficiency of either the functionality categories or their corresponding building blocks in Table 2.3. Sufficiency investigations for the freely evolving Web are difficult, but sets of use cases can assist the requirements identification, within sufficiently narrow categories. In the multimedia domain, [OGHR03] uses a set of use cases to define the requirements for a multimedia vocabulary for the Web. [Sch02] establishes the SMIL 2.0 time model sufficiently for the majority of the Web multimedia requirements. Use cases do not provide indisputable proofs of sufficiency, but they are a valid alternative for the Web, where sufficiency is impractical or impossible to prove.

2.4.2 Extensibility of the native presentation languages set

The transformation component enables the presentation of non native presentation XML languages (not in $\mathcal{L}_p$), if the $\mathcal{L}_p$ functionality is sufficient. The presentation component must be extensible, in order to allow $\mathcal{L}_p$ to evolve in parallel with the continuously increasing Web presentation requirements.

Plug-ins and *applets* are the currently prevalent method of extending the functionality of the presentation component. A plug-in is a device and browser specific component that fully controls the presentation of a new language or media type. An applet is a device independent component that controls a rectangular screen area and

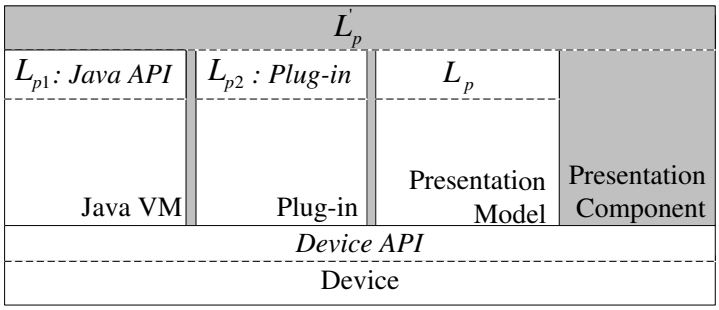

Figure 2.9: $\mathcal{L}_p$ extension using plug-ins and applets

runs on a Java virtual machine. Both approaches can provide arbitrary complex presentations, since, as Figure 2.9 illustrates, they are not restricted by the underlying presentation model. Plug-ins directly access the device API and add their "plug-in $\mathcal{L}_{p2}$" to the combined $\mathcal{L}'_p$. The applet approach exposes the generic Java API as a part of the $\mathcal{L}'_p$. Both applets and plug-ins implement their own presentation model that does not depend on other components or presentation models, and they can impede the interoperation between separate plug-ins, applets and the existing presentation model. Therefore, the presentation of documents that combine the languages in $\mathcal{L}'_p$ becomes problematic, because it requires a well orchestrated component interoperation.

Alternative extension approaches that integrate into the existing native presentation model can overcome the above interoperation issues, but they require generic and extensible presentation models. For instance, using CSS to define the style of new languages enables language interoperation on the foundation of the common CSS presentation model. However, as described in Section 2.3, CSS is not sufficiently generic, because it closely couples the structure and the presentation of documents, and it only covers a presentation functionality subset.

Scripting approaches, such as the *ECMAScript*[ECM99] and the *light–weight Web–based applications*[Bos04], do not directly extend the $\mathcal{L}_p$ set, but they allow presentation and processing model customisations. ECMAScript, which is the standardised version of JavaScript, is a scripting language that can manipulate XML data, using the DOM interfaces, and access runtime presentation information, through the browser API. The light–weight Web–based applications approach is a scripting alternative to applets, which is closely integrated to a hosting document. Both approaches extend the presentation component functionality by manipulating and combining its existing functionality. However, they do not have an explicit binding with XML constructs and, therefore, they cannot directly extend the $\mathcal{L}_p$. Moreover, the lack of a well defined browser API restricts their presentation processing model access. The light–weight Web–based applications approach proposes a custom processing model, which exposes a well defined interface, but it leads to the same problems with applets and plug-ins.

The *XML Binding Language* (XBL)[Hya01] and the *Rendering Custom Content* (RCC), which is a component of the current SVG 1.2 working draft, combine imperative scripting and declarative associations to enhance the presentation component functionality. XBL uses an object oriented model for associating behaviour with XML structures, in a similar way to the CSS styling information association. Specifically,

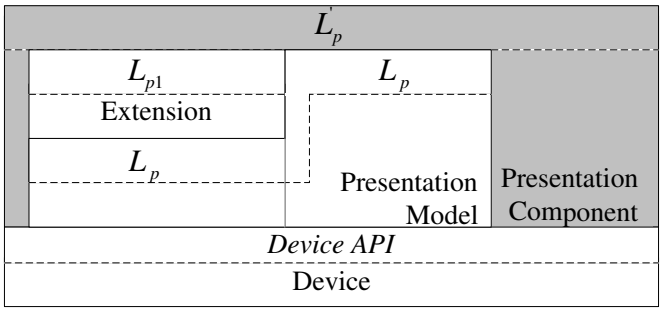

Figure 2.10: Extension with presentation model integration

XBL associates the name of an element and its position in a document to a set of properties, event handlers and "hidden content". The "hidden content" uses other XML languages to specify the presentational behaviour of an element. RCC uses a similar approach to XBL, but it is specific to the SVG presentation model. The SVG "hidden content", or "shadow tree" in the SVG terminology, solely consists of SVG elements that define the presentation of an element. RCC binding uses the qualified element names, instead of the positional XBL bindings. RCC does not explicitly support the XBL concept of object oriented scripting, and it is less generic than XBL, but it conforms to the SVG processing and presentation model. Both XBL and RCC are restricted by the existing presentation model, but their combination of binding, scripting and language reuse is the most promising current approach for an extensible $\mathcal{L}_p$.

Figure 2.10 illustrates how presentation model integration approaches, such as XBL and RCC, add new languages to a presentation model. The definition of the presentation of a new language uses the existing $\mathcal{L}_p$. The resulting $\mathcal{L}_p'$ seamlessly integrates both the original $\mathcal{L}_p$ and the introduced extension $\mathcal{L}_{p1}$. Further extensions can use the combined $\mathcal{L}_p'$ and allow recursive reuse of existing functionality.

2.4.3 Presentation of mixed namespace documents

A presentation component benefits from mixed namespace document processing, because it enables the combination of the individual $\mathcal{L}_p$ presentation building blocks into a coherent whole, and it also enables the seamless integration of $\mathcal{L}_p$ extensions. Moreover, there are existing $\mathcal{L}_p$ candidates that require mixed namespace processing, such as the *XML Linking language* (XLink)[DMO01]. XLink defines a syntax and a model for generic XML links that provide bidirectional, multiple source and multiple target linking. The "simple" XLink links are similar to the HTML links, and their syntax consists of the single `href` attribute of the XLink namespace. The `href` attribute can be attached to an element of any XML language, in order to specify the URL that the element is linked to. A document that uses XLink is a mixed namespace document, because it must use constructs from both the XLink and the content description language namespace.

In a similar manner to the other processing components, the lack of a generic

integration model impedes the presentation of mixed namespace documents. However, an $\mathcal{L}_p$ integration model can achieve generic integration by utilising the, yet undefined, presentation domain constraints in a more straightforward manner than a generic transformation and validation integration model. Specifically, the $\mathcal{L}_p$ languages are the interface of the presentation component, which has the well defined purpose of document presentation. As described in Section 2.4.2, prominent $\mathcal{L}_p$ extension approaches must integrate all extensions to a common underlying presentation model. Consequently, each presentation component implementation explicitly bounds the functionality of the $\mathcal{L}_p$ languages and of their extensions. The RCC proposal and the X-Smiles browser[PHV02], which are discussed below, utilise the presentation domain constraints for mixed namespace document presentation.

In contrast, *namespace assimilation* and *integration profiles* do not utilise the presentation domain constraints. Namespace assimilation approaches assimilate in a namespace the necessary constructs of other languages. For instance, SVG declares the SMIL animation constructs, within its own namespace. Namespace assimilation allows customized integration, according to the host language presentation model, but it does not comply with the modularity requirement and results in monolithic specifications and redundant constructs. Moreover, continuous updates are necessary to keep such specifications up to date. Integration profiles define how to integrate a fixed set of languages, and they do not necessarily introduce redundant constructs or require updates when one of their individual languages is updated. However, existing profiles, such as the XHTML + SVG + MathML profile [Mas02], do not define sets of integration rules, but combine the individual specifications into a profile specification. Therefore, they bear redundancy and must be updated, when the languages are updated. Additionally, current approaches focus more on the syntax, and not the presentation integration.

Namespace assimilation and integration profile approaches are problematic, because they do not utilise the presentation domain constraints, in order to propose a generic presentation integration model. Integration proposals that use the presentation domain constraints face a trade off between versatility and integration capabilities. [MMM04] proposes mixed namespace documents presentation on top of a common presentation framework. For instance, RCC enables the presentation of mixed namespace documents by mapping their constructs to the SVG presentation model. The common presentation model allows powerful integration, because it enables well defined interoperation between the individual languages. Such approaches require a well defined and sufficiently generic presentation model, which is difficult to establish, as discussed in Section 2.4.1.

The X-Smiles browser[PHV02] proposes more loose interoperation, in order to provide more generic integration. Each language namespace is associated with a language specific presentation module. A document subtree is presented by the instance of the presentation module that corresponds to its root node namespace. Each instance allocates a rectangular rendering area and can interoperate with other modules, if the subtree contains multiple namespaces constructs.

Figure 2.11 illustrates the DOM tree and the presentation layout of a mixed namespace document that contains XHTML, XForms and SVG constructs. The root of the tree is an XHTML node, and X-Smiles uses an XHTML presentation module instance to cover the browser rendering area. The XHTML presentation module interoperates with the XForms and SVG modules, for the presentation of the XForms and SVG subareas, using a box layout model and a set of predefined interfaces. The X-Smiles model successfully integrates nine W3C recommendations, but the box layout cannot cover all integration cases, and the interface based interoperation is problematic, because

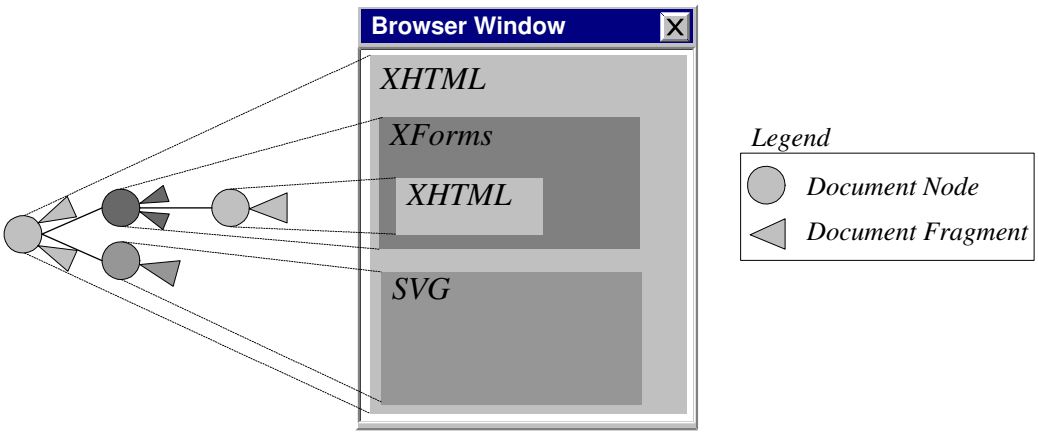

Figure 2.11: X-Smiles mixed namespace document presentation. The various colours represent the nodes and the presentation of different languages

interface extensions require component modifications. A viable alternative, illustrated in [Zil04], is property based component interoperation, where the introduction of new properties does not require existing component modifications, because they can ignore any unknown properties.

2.4.4 Presentation adaptation

The presentation of an XML document must fulfill the possibly conflicting requirements of the adaptation factors: the document author, the document user and the target device. The granularity of the $\mathcal{L}_p$ presentation information distributes the adaptation responsibility between the transformation and the presentation components. As illustrated in Figure 2.12, the more abstract the input of a component in relation to its output, the more adaptation freedom and responsibility it has. In 2.12 (a) the $\mathcal{L}_p$ interface between the transformation and presentation components lies in the middle of the overall adaptation range, and the adaptation capability/responsibility is divided equally between them. On the contrary, the $\mathcal{L}_p$ presentation information is more abstract in (b) and more precise in (c), and the adaptation capabilities/responsibilities are separated in a corresponding uneven manner. This section will describe both transformation and presentation adaptation approaches.

A component that performs content adaptation must be able to access the adaptation factor requirements. The document author requirements are directly available to all processing components, because they are either explicitly or implicitly contained within the XML documents. For instance, a style attribute can explicitly specify the font-size of a paragraph, and an `<h1>` XHTML element implies a larger and bolder font than the main body of a document. On the contrary, an alternative source must provide the document user and target device requirements. URIs can contain implicit device type information, such as “desktop”, “printer” and “mobile”, as illustrated in Section 2.3.3. CSS permits the use of separate presentation styles for separate device types. However, such approaches are not sufficiently generic, since the type of a device is only one of the several relevant adaptation factors.

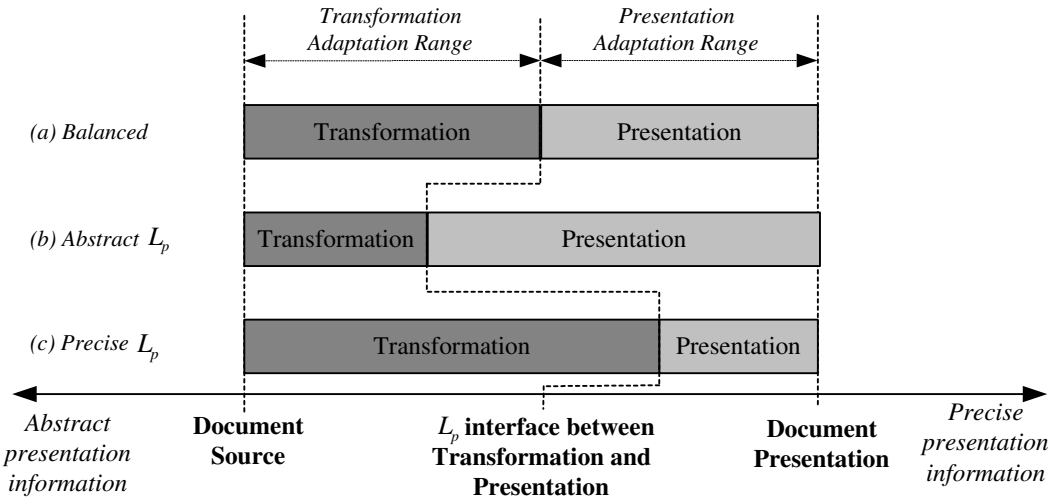

Figure 2.12: Adaptation responsibility/capability according to $\mathcal{L}_p$ granularity

The *Composite Capabilities/Preferences Profile* W3C recommendation[RHDS99] (CC/PP) is an RDF syntax that enables more precise requirements specification. A CC/PP profile is a collection of statements about user preferences, hardware and software capabilities. CC/PP profiles are "composite", because they can be composed out of a *default profile* and multiple *temporary* and/or *permanent* modification sub-profiles. A presentation model can use CC/PP to access the adaptation factor requirements, for a given target device, document and presentation session. For instance, a CC/PP default profile, which is associated with a target device, can contain the corresponding software and hardware capabilities. The target device can also be associated with a permanent sub-profile that describes software upgrades and session independent user preferences. Finally, the presentation session can be associated with a temporary sub-profile that contains temporary requirements, such as the user preferences for a specific document. The extensible CC/PP profiles can express any adaptation requirement, and their composite nature minimises the profile communication requirements, because it allows caching the most static sets of requirements. However, it does not provide a well defined resolution mechanism for conflicting statements, which is necessary for composing independently developed profiles.

As mentioned above, the adaptation capabilities of a component improve, if its input is sufficiently more abstract than its output. However, especially when documents are not authored with adaptation in mind, the adaptation range might not allow sufficiently powerful adaptation. The proposals described in [YW03] and [CMZ03] focus on adapting existing HTML content to limited resource devices, such as mobile phones. HTML describes the presentation of a document, but it does not convey higher level concepts, such as the relative content importance and semantic document partitioning, which are necessary for the proposed adaptation. The above proposals attempt to guess the original author intentions by a heuristic analysis of the HTML source, which takes into account the source indentation and the proposed screen positioning of the individual elements. The resulting documents are more adequate than the originals for

limited resource devices. However, such approaches are limited, because when the original author intentions are lost, heuristics can only estimate a limited subset of them, with questionable accuracy.

Device independent authoring aims to provide more powerful content adaptation by preserving the high level author intentions in the document. Existing approaches allow either the adaptation of generic XML content or of application domain specific content. For instance, CSS supports alternative presentations of generic XML content for various devices, because it allows the association of separate stylesheets with separate devices. As mentioned above, such a selection mechanism does not cover the variety of adaptation requirements; moreover, such stylesheets are document specific, do not allow functionality reuse and associate the presentation semantics with the document, instead of the XML languages. The proposal in [OH02] enables significantly more powerful adaptation, because it extends the adaptation requirements spectrum by associating CSS stylesheets and XSL-T transformation specifications with CC/PP profile queries.

Application domain specific adaptation approaches can only adapt a subset of XML documents, but they can exploit the properties of an application domain to offer powerful adaptation. For instance, the *adaptive grid*[JLS+03] provides high quality document layout, for a variety of page sizes. The document author provides information streams and sub-streams, which define the information sequence and grouping that the document user must perceive. The document author can also define a set of adaptive page layout templates, using numerical constraints on the page size and the visual component position. The layout engine solves the resulting constraints system and uses the most appropriate template for each page, which is also adapted to the corresponding information streams.

The adaptive grid approach is limited to adapting page layout specifications, according to a single group of adaptation requirements: the page dimensions. The highly constrained application domain enables more autonomous and powerful adaptation than the above generic approaches. Specifically, the layout engine is able to automatically evaluate the best template for each page, without the intervention of the document author or the document user. Moreover, the domain specific constraint definitions allow precise template adaptation, according to presentation time variables that cannot influence the adaptation processes in the above CSS and XSL-T based approaches. Finally, as opposed to the CSS stylesheets, the page templates are reusable and not document specific.

However, the adaptive grid is highly constrained and not adequate for adapting other information domain content, such as multimedia. As described in [OGHR03], multimedia content requires different adaptation approaches than text based media, because it uses different adaptation abstractions and processing models. For instance, typical text layout does not include temporal orchestration or media type and media transmission negotiation, which are central to a multimedia presentation. The above work[OGHR03] summarises the core requirements of a high level multimedia representation that preserves the necessary high level author intentions, for adapting the content to a variety of devices. The Cuypers presentation engine[OGC+01] precisely defines a high level multimedia content representation and a corresponding adaptation mechanism. It combines a generic prolog-based solver with multiple domain specific linear constraint solvers, and it iteratively transforms the original representation to increasingly more device specific representations, according to a set of predefined rules.

Both the adaptive grid and the Cuypers presentation engine use constraint systems

Requirements
XML compatible syntax
Point of execution specification
Predefined interfaces
Predefined but extensible processing model
Interoperation between independently developed code
Distributed, instead of centralised processing model
Interoperation with declarative approaches

Table 2.4: Imperative approaches requirements for XML presentation

to express the presentation interrelationships within a document. Constraints are a high level, declarative, tolerant and both device and processing model independent way to express presentation information. However, generic constraint solvers are complex and computation intensive processes that cannot be used for all devices that can access the Web. However, as Section 2.4.6 describes, restricted special purpose solvers can efficiently address the most common requirements for Web presentations. Constraints are high level constructs that can express the author presentation intentions, in a device independent manner, and provide a sufficient adaptation range for powerful adaptation. Section 2.4.6 will overview the field of constraint based document presentation for the Web.

2.4.5 Scripting

Scripting and other imperative presentation descriptions can reduce a presentation component's adaptation range, because they are more precise than declarative alternatives. However, as described in [Fou04], declarative techniques should be used when possible, but "scripting is here to stay", because it enables the $\mathcal{L}_p$ functionality to catch up with the evolving Web presentation requirements. Ultimately, any arbitrary presentation can be implemented in a generic imperative language, and there are proposals, such as [LFCH02], where an imperative-only approach controls the content presentation.

Many presentation approaches use imperative techniques, and all the $\mathcal{L}_p$ proposals, illustrated in Section 2.4.1, include code binding and event handling that require imperative descriptions. The introduced $\mathcal{L}_p$ extension approaches, such as applets, plug-ins and XBL, also use imperative specifications to define the behaviour of new constructs. Finally, the most prominent mixed namespace presentation approaches, illustrated in Section 2.4.3, are the interfaces of X-Smiles and the RCC/XBL proposals. The former use imperative Java interface implementations, while the latter use scripting for events processing. This section describes the existing imperative approaches and how they relate to the presentation of XML documents.

Table 2.4 summarises the requirements of imperative approaches that are adequate for generic XML document presentation. Specifically, they must conform to the Web design principles and fit within an XML processing model. Since XML parsing is part of an XML processing model, imperative specifications that can be embedded within XML documents must either use an XML or an XML compatible syntax. Both transformation and presentation components can use imperative specifications to modify

the document tree and to control the presentation, respectively. Therefore, an imperative specification must also explicitly or implicitly specify its *point of execution* and its corresponding interoperation with the XML processing components. A well defined processing model can assist the definition of the necessary interfaces and of the alternative points of execution, but it must be sufficiently extensible to cope with the evolving Web.

Additionally, mixed namespace document presentation requires interoperation between independently developed imperative specifications. Distributed processing techniques are more adequate for the Web than centralised ones, as described in [CLNL03]. For instance, a dependency resolution process, such as linking, must be able to locate the required code libraries using URIs, instead of local identifiers that are only valid within a single device. Finally, an imperative approach must be able to coexist and interoperate with declarative ones, so that the latter can be used whenever possible.

The JavaScript and ECMAScript approaches, which were introduced in Section 2.4.2, provide generic purpose scripting, and they are extensively used to customise and control the presentation of Web content. However, they do not fulfill the above requirements. Specifically, they do not have an XML compatible syntax, and they are often hidden within XML comments, to avoid harming the well formedness of documents. Since they are not XML specific, they do not define an XML processing model, a point of execution and the corresponding well defined interfaces, apart from the independently defined DOM interfaces. Additionally, they do not offer a binding mechanism that allows their interoperation with declarative approaches. Finally, they do not have a distributed processing model that enables the interoperation of independently developed code, and they require that all the code is included in or linked from a single document. Additional specifications, such as the DOM ECMAScript interfaces and the XHTML event handlers, provide mechanisms for using JavaScript and ECMAScript within XML processing, but they are not sufficient to fulfill all the above requirements.

XML-specific imperative approaches can fulfill the above requirements, but most current approaches focus on document validity and Web services, instead of document presentation. For instance, the *XML Objects* approach[KL02] associates Java classes to XML elements. However, the class association does not influence the presentation of the corresponding elements, because it solely provides the necessary interfaces for DOM tree manipulations that result in valid documents. Web services oriented approaches [FGK02, CLNL03] focus on Web services imperative definitions that associate application logic with URIs. Such associations allow their distributed interoperation, but they do not enable presentation behaviour binding to XML constructs.

The above approaches are not designed for XML presentation and only fulfill a subset of Table 2.4 requirements. Binding techniques, such as XBL, which was introduced in Section 2.4.2, can bridge such imperative approaches with an XML processing model. XBL explicitly separates the scripts that are executed before the presentation from the event handlers that are executed during the presentation. XBL decouples the scripting from the XML syntax, because the imperative specifications do not necessarily reside within the binding specifications. Moreover, XBL enables interoperation between declarative and imperative approaches, because the bindings are a combination of existing XML elements and scripting. As Figure 2.10 (page 31) illustrated, each binding results in an element that wraps the binding definition by extending the $\mathcal{L}_p$. New bindings can easily reuse existing ones by declaratively using their respective XML elements.

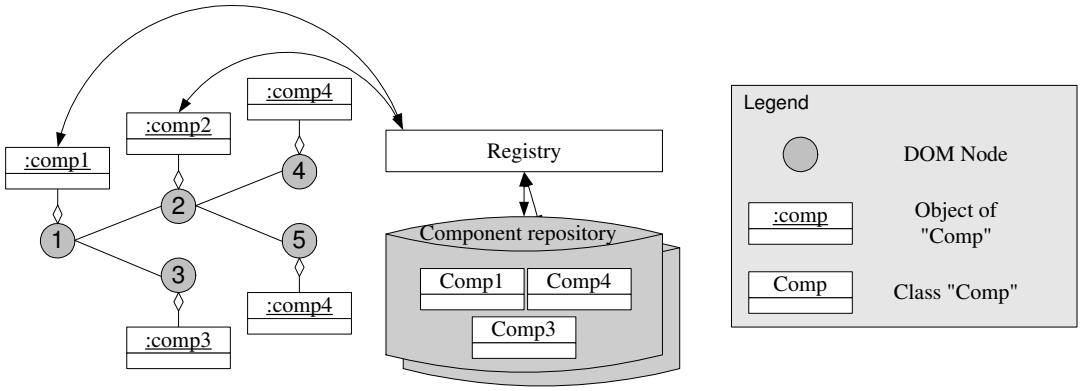

Figure 2.13: XVM relationship between DOM nodes and Java objects

XBL enables the use of existing scripting approaches for XML presentation by introducing a minimal processing model and a binding mechanism. It decouples the scripts from the XML syntax, explicitly defines the point of execution and allows interoperation between declarative and imperative approaches and between independently developed code. However, XBL does not define the necessary presentation model interfaces. Moreover there is no well defined document independent way to retrieve the necessary binding information.

The *XML Virtual Machine* (XVM)[LKSW04] is a generic purpose XML processing approach that explicitly defines a binding location mechanism. Specifically, XVM associates Java classes to XML elements, according to their qualified name, using a registry service that locates the appropriate class within a set of distributed class repositories. After the creation of the DOM tree, XVM attaches objects of the associated classes to the DOM element nodes. Each element's attached object is responsible for the processing of the respective element. The parent of an element, or a bootstrap process for the root node, is responsible for interoperating with the registry service to retrieve and instantiate the attached object.

Figure 2.13 illustrates the relationship between the DOM nodes and the Java objects. Each element is responsible for locating the necessary classes and attaching the respective objects to its children. For instance, the XVM bootstrap process attaches the `:comp1` object to the root node 1. Subsequently, `:comp1` interoperates with the registry to locate the implementations of the `Comp2` and `Comp3` classes, which are respectively associated with Node 2 and Node 3. `:comp1` fetches the implementations from the component repositories, instantiates the objects and attaches them to the child nodes. The process continues recursively, until all the necessary objects have been attached.

XVM is designed as a generic XML processing layer that can support more specific XML applications. XVM fulfils the distribution and point of execution requirements by its distributed processing model that separates between the object instantiation and subsequent processing. It does not explicitly cover the other requirements, but it provides the means to build applications that do. An example XVM application,

illustrated in [LKSW04], introduces a simplistic presentation model, where all the presentable objects implement a common interface. The root element interoperates with its descendants, using the common interface, to perform the layout and the presentation in a rectangular canvas. The example application fulfils the predefined presentation model interfaces and interoperation requirements.

2.4.6 Constraints

Several Web presentation proposals use *numerical constraints*, because they can express and resolve the possibly conflicting requirements of the adaptation factors. This section describes the fundamentals of numerical constraints, summarises requirements of constraint solvers for Web content presentations and overviews the existing proposals.

G. Badros provides an early but comprehensive literature review[Bad98] of the uses of constraints in interactive applications. Efficient constraint resolution has been an active research field for the last 50 years, and its applications span a wide variety of subjects. Constraint resolution is processor intensive, because there are no efficient generic purpose constraint solvers. However, there are several purpose specific approaches, which fulfill restrictive time complexity requirements.

$\mathcal{C}_{X_1,X_2,\ldots,X_n}$ and the more concise $\mathcal{C}_n$ will denote the set of all constraints over the variables $X_1, X_2, \ldots, X_n$. Every constraint $C \in \mathcal{C}_n$ is an expression of the form:

$$f(X_1, X_2, \ldots, X_n) \ op \ \ c$$

where f is a function over the variables $X_1, X_2, \ldots, X_n$, c is a constant and op is an operator where $op \in \{<, >, =, \leq, \geq\}$. There are several subsets of $\mathcal{C}_n$, according to the values that $X_1, X_2, \ldots, X_n$ and c can take. $\mathcal{C}_n^{\mathbb{R}}$ is the set of all constraints C where $X_1, X_2, \ldots, X_n$ and c are real numbers. Similarly, $\mathcal{C}_n^{\mathbb{Z}}$ is the set of constraints over integers.

A tuple $X = (x_1, x_2, \ldots, x_n)$ *satisfies* a constraint $C : f(X_1, X_2, \ldots, X_n) \ op \ \ c$ iff the expression $f(x_1, x_2, \ldots, x_n) \ op \ \ c$ is true. A constraint problem is a conjunction of constraints:

$$P = C_1 \wedge C2 \wedge \cdots \wedge C_m$$

$\mathcal{P}_{\mathcal{C}_n}$ is the set of all problems that consist of constraints $C_i \in \mathcal{C}_n$. A tuple $X = (x_1, x_2, \ldots, x_n)$ is a *solution* to the problem P iff X satisfies $C_i, \forall i \in [1, m]$. Finally, a *constraint solver* is a function that maps a problem to a set of solutions. For instance, a constraint solver for constraints in $\mathcal{C}_n^{\mathbb{R}}$ is a function $S : \mathcal{P}_{\mathcal{C}_n} \longrightarrow \wp(\mathbb{R}^n)$ that maps each problem in $\mathcal{P}_{\mathcal{C}_n}$ to a set of n-tuple solutions.

Adequate constraint resolution systems for interactive applications must be efficient and allow over and under constrained systems, as described in [BB98, Bad98]. Efficiency is important for presenting Web documents, because of the variety of the target device capabilities and the document users expectation for nearly instant document rendering. Since there are no efficient generic purpose solvers, it is necessary to restrict the types of constraints used for document presentation and to choose the optimal presentation attributes representation. The latter can influence the complexity of the required solver. For example, consider the two lines in Figure 2.14a, and a constraint C that requires that they have equal lengths. If the lines are defined using pairs of coordinates, then C will be the a polynomial equation:

$$(x_3 - x_1)^2 + (y_3 - y_1)^2 = (x_4 - x_2)^2 + (y_4 - y_2)^2$$

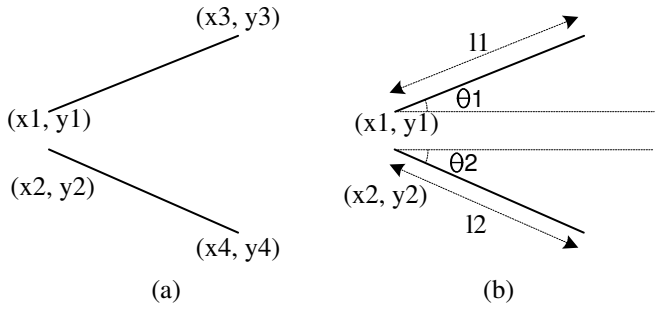

Figure 2.14: Alternate presentation attributes representations: line representation using a pair of coordinates (a) or starting coordinates, angle and length (b)

Alternatively, if the lines are defined using their starting coordinate, their angle and their length, as illustrated in 2.14b, C will be the simpler linear equation $l_1 = l_2$.

Efficient problem solution reevaluation is necessary for supporting user interaction and time dependent content, such as animations. Solution reevaluation must follow the principle of the "least astonishment"[Bad98] and only introduce the minimum possible modifications. For instance, dragging the 2.14a (x_3, y_3) point, within an interactive presentation, must result in modifying only x_3 and y_3 and no other variable, except if there is a constraint that requires so. Finally, a constraint resolution system for the Web must allow under and over constrained systems, in order to accommodate the possibly conflicting or incomplete requirements of the presentation adaptation factors.

The core differences between existing constraint solvers relate to their efficiency and generality tradeoff, because most solvers allow both under and over constrained systems. The most efficient solvers are the *local propagation solvers* (LPS), which consider a single constraint at a time. *Local propagation problems* are the problems that can be solved by an LPS. For instance, consider a local propagation problem:

$$P = \begin{cases} C_1 : & X_1 = 2 \\ C_2 : & X_1^2 + X_2 = 3 \\ C_3 : & X_1 + X_2 + X_3 = 7 \end{cases}$$

Figure 2.15a is a graph representation of the variable relationships expressed by the constraints in P. An LPS can solve P by identifying the value of the most restricted variable, which indicated by a greater number of edges, and propagating the solution to the less restricted ones. As Figure 2.15b illustrates, C_1 explicitly defines the value of X_1. Given that $X_1 = 2$ and using the constraint C_2, $X_2 = -1$. In a similar manner, $X_1 = 2, X_2 = 1 \stackrel{C_3}{\Longrightarrow} X_3 = 6$.

One way problems are the subclass of the local propagation problems, where each constraint is an assignment that only constrains the value of a single variable. One way problem solvers are faster than general LPS, because the corresponding graph is directed, which reduces the amount of possible constraint order permutations. For instance, if C_3 was $X_3 := 7$ - X_1 - X_2, it is explicit that C_3 must be used after the evaluation of X_1 and X_2.

Local propagation problems are only a subset of the general constraint problems, but LPS are efficient and do not introduce restrictive assumptions, such as constraint

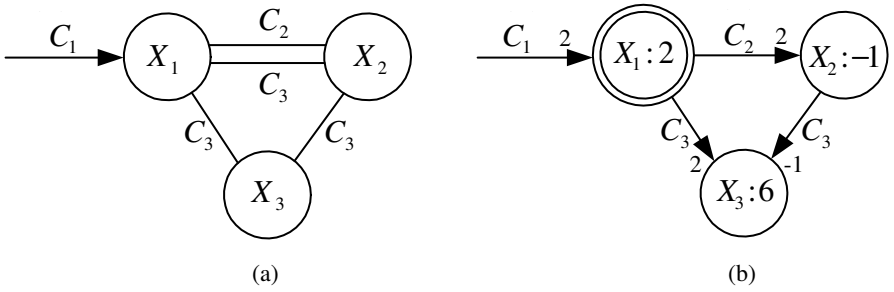

Figure 2.15: Graph representation of a local propagation constraint problem (a) and of its solution (b)

linearity. One way solvers impose even less restrictions, and constraints can use arbitrary complex expressions, without increasing the complexity of the solver. Consequently, several constraint based approaches for the Web are based on LPS. For example, the Constraint CSS (CCSS)[BBMS99] extends CSS with constraints that express relationships between style attributes, where font size constraints are resolved with a one way LPS. The constraint resolution processing overhead is negligible, and the resulting syntax is significantly more expressive than the original CSS syntax.

The *adaptive grid*[JLS+03] also uses one way constraints, to enable desktop publishing quality layout to Web documents for a variety of page sizes. It uses a set of layout templates, which include a set of constraints that enable page size specific template adaptation. Finally, the functional extensions for XML proposal[KST03] considerably extends the SMIL animation and SVG presentation model functionality by introducing one way constraints over the presentation attributes. The one way constraints, within the above work, are expressed as functional specifications of presentation attributes, which are reevaluated in real time.

The introduction of local propagation constraints can extend the functionality of a presentation model, but they can only express a limited set of problems. For example, consider the Figure 2.14b and a problem that requires both equal line lengths ($l_1 = l_2$) and a total length of 4 ($l_1 + l_2 = 4$). An LPS cannot produce the solution $l_1 = l_2 = 2$, because it is a *simultaneous constraint problem*, which requires the simultaneous consideration of multiple constraints. There are no computationally efficient generic solvers of simultaneous constraint problems.

However, *linear constraints* over real values can express the most common layout requirements. $\mathcal{CL}_n$ will denote the set of linear constraints, which is the subset of $\mathcal{C}_n$, where $C \in \mathcal{CL}_n$ iff $C \in \mathcal{C}_n$ and C is of a linear form:

$$a_1X_1 + a_2X_2 + \cdots + a_nX_n \ op \ \ c$$

where $a_1, \ldots, a_n$ are constants. The set of linear constraints over real numbers is denoted by $\mathcal{CL}_n^{\mathbb{R}}$ and $\mathcal{CL}_n^{\mathbb{R}} \subset \mathcal{C}_n^{\mathbb{R}}$.

The *Simplex* algorithm provides an efficient solver for $\mathcal{P}_{\mathcal{CL}_n^{\mathbb{R}}}$ problems, and it can

solve under-constrained problems by minimising a linear expression, the *objective function*. The Cassowary[BB98] and the QOCA[BMSX97] algorithms are incremental versions of the Simplex algorithm, which apply the "least astonishment" principle and also allow over constrained systems. Both Cassowary and QOCA fulfill the above requirements for Web document presentation, within the domain of linear constraints, and they form the basis of several presentation approaches. The previously introduced CCSS approach uses the Cassowary algorithm for solving linear problems to position presentable objects. The work in [BLM00] uses linear constraints within multiple layout templates, in a similar manner to the use of one way constraints in the introduced adaptive grid. Finally, the constraints extensions to SVG[BTM$^+$01] uses linear constraints for positioning SVG graphics. All the above approaches conclude that the resulting presentation models are efficient and that the use of constraints provides higher level presentation descriptions, which allow powerful adaptation to a variety of devices.

No efficient generic solver exists for nonlinear simultaneous problems, but there are several proposals that efficiently address the most relevant problem subsets for presenting Web documents. For instance, font size constraints require finite domain constraints in $\mathcal{C}_n^{\mathbb{Z}}$, because most platforms only display integer font sizes. Moreover, the selection of multiple layout templates requires disjunction of constraints, as opposed to the above definition of constraint problems as a conjunction of constraints. The backtracking techniques that are used in the field of logic programming, such as in Prolog, are specifically designed to handle finite domain problems and disjunctions. However, as [Bad98, LMS99] describe, even optimised backtracking algorithms are exponentially complex and not adequate for interactive applications.

However, both the finite domain constraints and constraint disjunctions that are commonly required for document presentation can be solved efficiently. A font size constraints observation[LMS99] is that they contain at most two variables and can be addressed by a polynomial complexity solver. Constraint conjunction problems can be also solved efficiently by an algorithm proposal[MMSB01] that eliminates the disjunctions by activating only one constraint of each disjunction at any time. Reevaluations that invalidate the active constraint trigger a mechanism that selects a new active constraint. A similar approach[HMM02] can address multiple cases of nonlinear simultaneous constraints by reducing nonlinear problems to linear ones, within a small range of values. Reevaluations that result in values out the specified range trigger the generation of a new approximation generation, which corresponds to the new value range.

2.4.7 Presentation summary

The presentation component is responsible for conveying an XML document according to the possibly conflicting requirements of the adaptation factors: the document user, the document author and the target device. A presentation component that is adequate for generic XML processing must be extensible, because it can only natively support a limited set of languages $\mathcal{L}_p$. Moreover, it must provide a presentation integration model, content adaptation and both imperative and declarative mechanisms, such as scripting and constraints, for customising the document presentation

The lack of well defined Web content presentation functionality boundaries impedes establishing the sufficiency of an $\mathcal{L}_p$ set. Moreover, combining the existing standardised XML languages does not result to an adequate $\mathcal{L}_p$, because they contain redundant

features and their sufficiency has not been established. A set of presentation functionality studies[Har04, All04, Bir04] derive $\mathcal{L}_p$ sets from abstract sets of features, but they neither investigate their sufficiency nor illustrate the derivation of the individual languages.

Both XBL and RCC are prominent binding approaches, which can form the foundation of an extensible $\mathcal{L}_p$ that supports mixed namespace documents and combines imperative and declarative processing descriptions. They both can reuse an existing presentation model, to seamlessly integrate language extensions within an existing $\mathcal{L}_p$, and they allow inter-language interoperation by mapping everything to a single presentation model. The X-Smiles interface-based interoperation alternative is generic, but it does not provide powerful integration, and the inevitable updates to its interfaces would require updating all their corresponding components.

ECMAScript and JavaScript imperative approaches are not designed for XML, but they fulfill a substantial subset of the XML imperative definition requirements, if they are used within the XML tailored bindings of XBL. The XML Virtual Machine (XVM) is a processing model specifically designed for XML. It does not directly provide a presentation solution but provides the foundation for applications that can fulfill all imperative definition requirements. In order to utilise the full potential of XBL/RCC and XVM, well defined and sufficient $\mathcal{L}_p$ and presentation model interfaces are necessary, but there currently are no such specifications.

The adaptation range of both the transformation and presentation components increases when their input is significantly more abstract than their output. Heuristic approaches can extend the adaptation range of a component, but they can only derive a very restricted set of the original document author intentions. Predefined stylesheet selection according to queries on generic requirement specifications, such as CC/PP profiles, can offer powerful adaptation. However, stylesheet selection is a one way process, which is not adequate for resolving conflicting requirements, and its interoperation with the presentation process is limited. Numerical constraints can convey the original author's intentions in several abstraction levels, resolve conflicting requirements and enhance the presentation component adaptation capabilities. Generic constraint solvers are complex, but there are efficient solvers for the majority of the required constraint systems for Web content presentation. However, there are no well defined ways for integrating constraint definitions to a $\mathcal{L}_p$ and for generically integrating the several individual solvers.

2.5 XML Browsers

This chapter has separated the XML document presentation process into four interoperating components, but most existing approaches are independent proposals and they are not combined within a Web browser. This section will summarise the support of the individual processing components within existing Web browsers, in order to assess the presentation functionality that is accessible by document authors and users.

There is a multitude of available Web browsers, but *Internet Explorer* (IE), the *Mozilla* family browsers, and *Opera* are currently the most widely used implementations, as illustrated in Figure 2.16. All three browsers support XML, but only IE and Mozilla focus on generic XML presentation. Opera natively supports XHTML and XML styling with CSS, but it does not support document transformations. Both Mozilla and IE support generic DOM manipulation, using JavaScript and XSL-T based

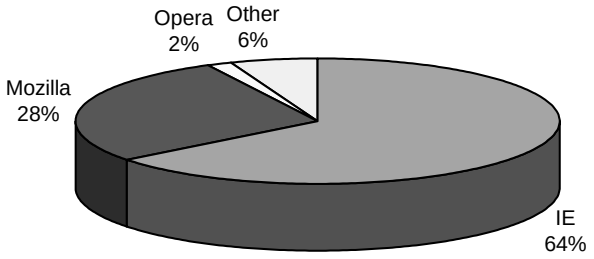

Figure 2.16: Browser user statistics, according to the approximate 2006 averages in `http://www.w3schools.com`

document transformations. IE supports additional imperative approaches that allow access to rich component libraries. Mozilla focuses on supporting multiple Web recommendations, such as SVG and XBL. None of the above browsers explicitly demonstrate a generic and customisable presentation model. However, IE and Mozilla define browser specific interfaces that enable imperative access to their internal presentation components, for dynamically initiating document manipulation processes, such as transformation and validation.

The *Amaya*, *X-Smiles* and*XEBRA*[THHH01] browsers are not as widely deployed, but they possess several distinctive characteristics. Amaya has been specifically developed by W3C for testing several of the Web recommendations. In a similar manner to Opera, Amaya neither supports XSL-T nor includes a generic mechanism for XML presentation. However, it supports XHTML, MathML, SVG and their combination based on integration profiles. It thus can provide insights into the profile based processing of mixed namespace documents. X-Smiles supports a variety of XML languages and mixed namespace documents, according to the simplistic integration model introduced in Section 2.4.3. XEBRA stands out because of its simplicity, and it is based on interpreted LISP programs that process XML documents, using a set of loosely coupled components. A XEBRA processing example[THHH01] illustrates that a fourteen line long specification is sufficient for defining an HTML browser, which combines an existing HTML parser, an XSL-T transformer and an XSL-FO renderer.

Table 2.5 summarises the processing properties of the above browsers. The Web design principles require *simple* and *modular* Web browsers. Only X-Smiles and XEBRA can be considered as simple, because their architectures are straightforward, their components interoperation is well defined, and they are relatively straightforward to extend. The extent to which modularity enhances a browser's functionality depends on its underlying presentation model and the interoperation between its modules. All introduced browsers are modular, but only X-Smiles and XEBRA are modular in a way that directly benefits the document users and authors.

As Section 2.4.1 described, the sufficiency of an $\mathcal{L}_p$ is not well defined. However, the variety of languages and the extensibility mechanisms that a browser supports can indicate its potential for supporting a sufficient $\mathcal{L}_p$. IE, Mozilla and X-Smiles support a wide range of XML languages and also support plug-ins and other platform specific extension approaches. Additionally, Mozilla and X-Smiles extensions can respectively

	IE	Mozilla	Opera	Amaya	X-Smiles	XEBRA
Simple					••	••
Modular	•	•	•	•	••	••
Rich $\mathcal{L}_p$	••	••			••	**P**
Extensible $\mathcal{L}_p$	•	••			••	**P**
Non DTD validation	•				•	
Transformation	••	••			•	••
Dynamic pres. model						
Scripting	••	••	•		•	••
Pres. component interfaces	••	••			•	••
Constraints						
GMNS presentation		•			••	
GMNS validation						
GMNS transformation						
Generic content adaptation						

Table 2.5: Properties of current browser implementations. GMNS stands for "Generic Mixed Namespace". • signifies possession of a property. •• identifies a property that is central to and enhances the functionality of an implementation. **P** identifies the potential to acquire a property.

use the XBL and the X-Smiles interfaces, which allow their integration with the underlying presentation model. XEBRA includes only a minimum set of components and does not provide a presentation model that can incorporate additional components. However, a combination of additional components and a generic integration layer can potentially lead to a sufficient and extensible $\mathcal{L}_p$. On the contrary, Amaya and Opera do not offer similar generic extension mechanisms, and their current $\mathcal{L}_p$ is significantly restricted.

A generic presentation model must not dictate a predefined processing sequence (as described in Section 2.1.1), but none of the approaches above provides such a generic processing model. However, IE, Mozilla and XEBRA allow dynamic imperative access to their processing components, which can allow to create transformation pipelines, customise the presentation model and potentially orchestrate the interoperation between the validation and transformation processes. Apart from Amaya, all introduced browsers offer several forms of scripting, and the IE, Mozilla and XEBRA internal component interfaces allow the introduction of custom presentation models. In contrast, X-Smiles, which both supports JavaScript and has well defined component interfaces, does not allow such presentation model customisations. Finally, none of the above browsers have a presentation model that supports constraint based presentation.

There is minimal support for generic mixed namespace processing that is not based on profiles. Mozilla supports XBL, which allows generic presentation of mixed namespace documents, but they ultimately have to be mapped, through the XBL bindings, to either a single namespace document or to a profile-based mixed namespace document. The simplistic integration model of X-Smiles is the only generic mixed namespace processing candidate. The lack of generic mixed namespace transformation or validation models can be attributed to the lack of namespace based bindings to validation and transformation processing information.

Finally, most Web browsers include minimal adaptation capabilities, such as choosing alternative fonts, page sizes and stylesheets, according to the type of the target device. However, there is no explicit generic content adaptation model for either their presentation or their transformation components.

The above browser discussion should not be taken as an evaluation of their relative usefulness and applicability, but, as an indication of their adequacy for generic XML presentation. For instance, XEBRA illustrates some prominent design concepts, but it is not a complete browser and does not appear in any browser usage statistics. On the other hand, Opera is not a prominent generic XML content presentation approach, but it is very efficient, renowned for its CSS implementation and used by a considerable minority of Web users.

Summarising, current browsers offer rich and extensible sets of functionality, but not within a generic XML presentation framework. The lack of such a framework leads to problematic interoperation between $\mathcal{L}_p$ extensions and to platform specific imperative presentation model customisations. Moreover, generic mixed namespace processing support, explicit content adaptation and declarative presentation specification constructs, such as constraints, are missing.

2.6 Discussion

The lack of generic XML presentation processing models impedes a thorough review of the XML presentation literature, because the set of related research areas is not well defined. This chapter identified the relevant research areas by outlining the top level XML presentation processing components: parsing, validation, transformation and presentation. Subsequently, it investigated the relevant existing approaches and the core issues that they must address: mixed namespace document processing, processing information binding and adaptation. Document parsing is well defined, but there is a multitude of incompatible alternative validation, transformation and presentation approaches. Their corresponding mixed namespace processing, processing information binding and adaptation are neither well defined nor adequate for the Web.

Most XML document processing approaches use inline processing instructions, which delegate the document processing specification responsibility to the document author. Namespace URI based associations, such as the RDDL and XBL proposals, are more adequate for the Web, but the current lack of a generic integration models impedes their wide deployment. A generic integration model is necessary for their deployment, because it allows the inference of a document's processing from the processing of its individual constructs. Language integration profiles and namespace assimilation are only short term solutions, because they do not fulfill the Web design requirements, and their enumeration becomes exponentially complex, under the continuous introduction of new languages. Existing integration models do not cover document transformation, and they are not sufficiently generic and powerful. NRL enables the independent validation of a mixed namespace document's subtrees, but it does not define the necessary construct relationships to avoid erroneous validation cases. The X-Smiles and XBL/RCC models do not provide sufficient inter-language interoperation and a sufficiently generic underlying processing model, respectively.

The transformation and presentation processing components share the responsibility to adapt a document, according to a variety of user preferences and device capabilities. CC/PP provides a generic and extensible adaptation requirements representation,

Processing issues	Resolution directions
Component interoperation.	Dynamic interoperation, within a generic processing model.
No generic integration models.	Extend prominent approaches, such as NRL, to incorporate the necessary construct associations.
No adequate distributed binding.	Combine URI associations with a distributed location mechanism, such as RDDL.
No generic processing adaptation.	Fine grained adaptive processing information associations. Definition of presentation domain constraints. Sufficiently generic constraint solvers.
No well defined presentation functionality set.	Use case based functionality investigation. Set of minimalistic languages that cover the identified functionality.
$\mathcal{L}_p$ extensibility. No adequate imperative approaches. No integrated constraint mechanisms.	Well defined presentation model and interfaces. Scripting and constraints binding, using adequate technologies, such as XBL or XVM. Well defined $\mathcal{L}_p$ integration model.

Table 2.6: Summary of the identified XML processing issues and their corresponding resolution directions

but most adaptation approaches use imprecise sources of adaptation requirements, such as URL address implications or limited sets of device types. The proposed CC/PP based stylesheet selection is more powerful, but it focuses on the document as a whole, instead of allowing the separate adaptation of its individual constructs. The most powerful adaptation approaches focus on constrained areas, and they are not adequate for the Web. However, declarative presentation specification techniques, such as constraints, can enhance the adaptation capabilities of a generic presentation component. Generic constraint support is complex and inefficient, but there are efficient solvers for the most common Web layout problems.

Modern Web browsers implement a rich set of functionality and natively support a variety of XML languages. However, the lack of a well defined set of the necessary presentation functionality for a generic processing model impedes its development. A generic processing model is essential for reusing the rich existing functionality, towards generic XML content presentation, and for developing a generic functionality extension mechanism.

2.6.1 Current issues and resolution directions

The literature review in this chapter identified the core processing problems for presenting XML documents and the core directions towards addressing them, which are summarised in Table 2.6.

The core processing issues are the lack of well defined component interoperation and of generic integration models. The former is essential for document processing that does not use inadequate static processing sequences. Moreover, the processing components require an integration model, because it provides the foundation for well defined processing of mixed namespace documents that use an open set of independently developed languages. The current adaptation profile and namespace assimilation approaches are

not adequate for the Web. An adequate proposal can reuse the prominent integration approaches, such as the NRL, and also incorporate the RDF's concept of well defined associations to the XML authoring model.

The remaining processing issues relate to the binding, adaptation and presentation of XML documents. An adequate binding mechanism, which allows the processing an open set of languages, must be distributed and follow the Web design principles. The widely used document based binding is neither distributed nor compliant with the Web design principles. URI based binding, such as in RDDL, is more extensible and distributed, and it can form the basis of an adequate binding mechanism.

The most powerful adaptation approaches are not adequate for an open set of languages, but their principles can be applied for generic document processing. The most prominent generic adaptation approach is the CC/PP based stylesheet selection, but its document wide stylesheets are not sufficiently fine grained. A well defined set of presentation processing domain constraints can enable more precise adaptation and the application of powerful adaptation approaches, such as constraint systems.

Generic scripting and constraint approaches require a well defined presentation model that must rely on a well defined and extensible $\mathcal{L}_p$. The sufficiency of the individual application domain functionality proposals is not well established, but use case based investigations are an adequate alternative method for defining the sufficient presentation functionality, within an application domain. An $\mathcal{L}_p$ definition can be based on a set of such use case based investigations, which cover all the necessary application domains, and a derivation of the corresponding set of minimal languages. An extensible $\mathcal{L}_p$ set is necessary for a dynamic information medium, such as the Web, but commonly used extension technologies, such as plug-ins and applets, are problematic. A well defined presentation model can be the foundation of an adequate $\mathcal{L}_p$ extension mechanism that can provide the necessary interfaces for taking full advantage of prominent extension technologies, such as XBL and RCC.

2.6.2 Concluding remarks

There are several approaches for the individual areas of XML document presentation processing; however, most existing approaches do not follow the Web design principles, and there is no generic XML presentation processing model that is adequate for the Web. An adequate model must define the parsing, validation, transformation and presentation of mixed namespace documents, using a distributed binding model, according to a variety of preferences and capabilities. Most binding approaches are document based, because of the lack of a generic integration model. Profiles and namespace assimilation are only short term integration solutions, and the more generic NRL and NVDL approaches do not define the necessary construct relationships. CC/PP provides an adequate adaptation requirements representation, but most existing adaptation approaches use imprecise adaptation requirements sources. The proposed CC/PP based stylesheet selection is powerful, but more fine grained stylesheet applications are necessary. Finally, there is currently a wide spectrum of presentation functionality, but the lack of a well defined $\mathcal{L}_p$ set results to the absence of a generic presentation model.

This chapter reviewed the existing XML presentation processing literature, identified the core XML presentation processing problems and established the core directions toward addressing them. The literature review followed the introduced top level processing model, which allowed the identification of the necessary processing components and their necessary functionality and interoperation for generic document processing.

Specifically, it introduced an original way of addressing XML processing: the various forms of processing, such as validation and transformation, are not independent, but they represent the individual parts of a common processing model. Each introduced processing component is important in itself, but its interoperation with all other components is also important. The next chapter proceeds to providing the necessary definitions for investigating the restrictions of the presentation domain and for unambiguously expressing our hypothesis.

Chapter 3

Definitions and the hypothesis

The vision behind the Web is that anyone can make available any form of information and link to everything available in the Web, as described in Section 1.3. A processing entity must be able to both locate the necessary information and derive its adequate interpretation, within its specific context. For instance consider the example document illustrated in Listing 3.1, which is duplicated from Section 1.4.1. An XML browser requires machine processible presentation descriptions of each document construct and of the relationships between the individual constructs. A human might be able to interpret a document, according to the corresponding language specifications and the descriptive construct names. However, the typical interaction with Web information occurs through the interface of a browser. Consequently, machine processible descriptions of Web information are necessary and the main future direction for the Web is to make its information more machine processible, as Berners-Lee has described in [BL98a].

Generic processing descriptions of an information domain are feasible, if there is a core set of information constructs, which have well defined processing and can be combined to describe every possible processing of all information domain data. Specifically, a *grounded* document[BL02a] is a document that exclusively contains constructs of such a core set. The processing of a *non-grounded* document, which may contain additional constructs, can be described as a combination of the core constructs.

For instance consider the XML presentation domain and a corresponding core set of XML languages, which include all necessary presentation functionality for presenting XML documents. A document that only uses the above languages is a grounded document, and a document that contains at least an element or an attribute of another language is not grounded. The presentation of a non grounded document can be described using a grounded document, since the core set of languages can describe any conceivable presentation. XML transformations can map any XML document to its presentation by transforming its constructs to their corresponding core set presentation interpretation.

The SMIL, CSS, XSL-FO, SVG, XForms, XBL and XML Events languages can be considered as the core set of languages for Web browser functionality, because they cover the key Web functionality of existing Web browsers[Har04]. A document that uses SVG and SMIL constructs to describe an animated vector graphic is a grounded document. In contrast, a document that describes a tabular layout of animated vector graphics, using the `http://layout.org` language illustrated in Listing 3.1, is not a grounded document. However, the presentation of such a non grounded document

```
1  <?xml version="1.0"?>
2  <l:table xmlns:l="http://layout.org/"
3        xmlns:f="http://furniture.org/">
4   <l:row>
5    <l:column>
6     <f:table material="iron">
7       An office table
8     </f:table>
9    </l:column>
10   <l:column>
11    <f:table material="wood">
12      A kitchen table
13    </f:table>
14   </l:column>
15  </l:row>
16 </l:table>
```

Listing 3.1: XML document with namespaces

can be described using the languages in the core set. For instance, a CSS stylesheet can describe the tabular layout of the `http://layout.org`, using the CSS box layout functionality.

The unrestricted nature of both the Web information and of its processing impedes the development of a complete core set of processing constructs. The document presentation domain does not constrain the processed information, but, as a specific type of processing, it constrains the corresponding processing domain. Such constraints are not well defined, and existing presentation technologies do not utilise them for generic Web content presentation. If the set of XML presentation processing domain constraints can be well defined, they can form the foundation for establishing the corresponding core functionality set and developing a generic presentation processing model.

The previous chapters introduced the necessary XML, Web, and XML presentation processing background. This chapter establishes the foundation for the remainder of this thesis by defining the necessary presentation processing terminology, defining the scope of this thesis and stating our hypothesis, in sections 3.1, 3.2 and 3.3, respectively.

3.1 XML presentation processing definitions

The study of document presentation processing requires unambiguous definitions of the necessary terms, in order to investigate the constraints of the presentation domain and to communicate the ideas of this thesis. However, most of the accustomed Web terminology is not well defined and has context dependent interpretations. For instance, Table 3.1 illustrates four separate interpretations of the term *semantics*, according to the context of its use. This section defines the necessary unambiguous presentation processing terminology.

3.1.1 XML documents and languages

Within the context of the Web, the term *document* refers to any Web resource description, which can use any existing or future data representation. This thesis only focuses

Context	Example	Interpretation
Core XML	The inherent rich *semantics* of XML.	Structured information and use of meaningful names.
XML Schemas	Schemas define the *semantics* of XML languages.	Syntactical constraints with optional data type information.
RDF	The *semantic* Web.	General purpose machine processible information, in the form of graphs.
Ontologies	*Semantically* rich representation.	Ontological extension of RDF that allows inferences and clever data queries.

Table 3.1: Context dependent interpretations of the term *semantics*

on XML documents, because, as described in Section 1.4.3, XML is sufficiently generic to represent any information, it is adequate for the Web and its standard representation provides the foundation for a generic document processing model.

XML documents (D): D represents the set of all well-formed XML documents.

XML languages ($\mathcal{L}$): $\mathcal{L}$ represents the set of all XML languages

An XML document encodes information using several components, such as elements, attributes, text, comments, processing instructions and document type declarations. The elements and attributes of an XML document define the tree structure of the represented information and express the relationship between a document and its corresponding XML languages. The term *XML constructs* will refer to the set of all XML elements and attributes, which can be defined as optionally qualified XML names.

XML constructs (Σ): The set of all *XML constructs* Σ includes all qualified or unqualified names of XML elements and attributes:

$$\Sigma = (URI \cup \epsilon) \times S$$

where URI is the set of URIs, S is the set of all non-qualified XML names and ϵ is an null URI

The relationship between documents and languages with their constructs can be expressed by a set of functions. Functions $docConstructs()$ and $langConstructs()$ will map a document and a language, respectively, to their corresponding set of XML constructs. Whether a construct $\sigma \in \Sigma$ is an element or an attribute depends on either its usage within a document or its definition within a language, because XML elements and attributes share the same alphabet Σ. Consequently, the element and attribute specific functions $docConstructs_a()$, $langConstructs_a()$, $docConstructs_e()$ and $langConstructs_e()$ are necessary for separating between attribute and element constructs.

Document constructs functions (*docConstructs*):
$docConstructs : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs(d)$ is the set of all the XML constructs in d.
$docConstructs_e : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs_e(d)$ is the set of all the XML constructs that appear as elements in d.
$docConstructs_a : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs_a(d)$ is the set of all the XML constructs that appear as attributes in d.

Language constructs functions (*langConstructs*):
$langConstructs : \mathcal{L} \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs(L)$ is the set of all the XML constructs that are defined by L.
$langConstructs_e : D \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs_e(L)$ is the set of all the XML constructs that are defined as elements by L.
$langConstructs_a : D \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs_a(L)$ is the set of all the XML constructs that are defined as attributes language L.

XML namespaces are the standard mechanism for the unique identification of document constructs, which, as described in Section 2.2.3, is the necessary foundation for information associations in an open environment, such as the Web. Therefore, namespaces are necessary for associating XML documents to their corresponding XML languages and for any other document construct based association. This thesis focuses on the sets of *namespace qualified documents* $D_Q \subset D$ and namespace bound languages $\mathcal{L}_Q \subset \mathcal{L}$, where all element constructs have an associated namespace URI. Attribute constructs do not require qualification, because unqualified attributes are uniquely identifiable within the context of their parent element: "The combination of the attribute name with the element's type and namespace name uniquely identifies each unqualified attribute"[BHL99].

Namespace qualified XML documents (D_Q): The set of all *namespace qualified XML documents* $D_Q \subset D$ is the subset of XML documents where $\forall d \in D_Q, \forall \sigma = (uri, s) \in docConstructs_e(d), uri \neq \epsilon$

Namespace bound XML languages ($\mathcal{L}_Q$): The set of all *namespace bound XML languages* $\mathcal{L}_Q \subset \mathcal{L}$ is the subset of XML languages where $\forall L \in \mathcal{L}_Q \ \forall \sigma = (uri, s) \in langConstructs_e(L), uri \neq \epsilon$

As described in Section 2.1.1, a presentation processing model must allow the processing of mixed namespace XML documents, which can contain constructs of multiple XML languages. The relationship between an XML document d and its corresponding XML languages $\mathcal{L}_d$ is only well defined for the uniquely identifiable constructs of namespace qualified documents and namespace bound languages.

Document's languages ($\mathcal{L}_d$): For a namespace qualified document $d \in D_Q$, the set of its languages is the subset of the XML bound languages $\mathcal{L}_d \subset \mathcal{L}_Q$, where
$\forall \sigma \in docConstructs_e(d), \exists L \in \mathcal{L}_d$ where $\sigma \in langConstructs_e(L)$
and
$\forall \sigma = (uri, s) \in docConstructs_a(d), uri \neq \epsilon, \exists L \in \mathcal{L}_d$ where $\sigma \in langConstructs_a(L)$.

3.1.2 XML semantics

A language author designs an XML language, in order to define the necessary syntax for conveying the data of an information domain. Within this thesis, the term *XML semantics* will refer to a language author's intended usage and interpretation of an XML language. An unambiguous XML language must have a single precisely defined usage and interpretation; consequently, it must have a *single precisely defined* semantics.

XML semantics ($\mathcal{I}$): For each XML language $L \in \mathcal{L}$, there is a single precisely defined semantics $I \in \mathcal{I}$, which represents the language intended usage and interpretation by its author. The set $\mathcal{I}$ is the set of all XML language semantics.

The presentation semantics are the subset of XML semantics that *are* a presentation, as opposed to XML semantics that *can be associated* with a presentation. For instance, the tabular layout language, illustrated in Listing 3.1, can be associated with presentation semantics, because the primary interpretation of its constructs is their corresponding tabular presentation layout. On the contrary, the furniture description language must be associated with non presentation semantics, because its primary interpretation relates to generic information about furniture. Nevertheless, its constructs can also be associated with a furniture information presentation. The subsequent section will further clarify the above separation, using the concept of presentation languages and presentation documents.

Presentation semantics ($\mathcal{I}^P$): The set of all *presentation semantics* is the subset of XML semantics $\mathcal{I}^P \subset \mathcal{I}$, that define a language's interpretation according to the presentation of its constructs to the document user.

3.1.3 Presentation languages and documents

The *presentation semantics* definition enables the definition of *presentation languages* and *presentation documents*, as the sets of languages that are associated with presentation semantics and of namespace qualified documents that only use presentation language constructs, respectively. The presentation documents must be namespace qualified, in order to ensure their well defined association with the corresponding languages.

Presentation languages ($\mathcal{L}^P$): The set of all *presentation languages* is the subset of XML languages $\mathcal{L}^P \subset \mathcal{L}$, where each $L \in \mathcal{L}^P$ is associated with presentation semantics $I \in \mathcal{I}^P$.

Presentation documents (D^P): The presentation documents subset of namespace qualified XML documents is the set $D^P \subset D_Q$, where $\forall d \in D^P$ the set of document languages $\mathcal{L}_d$ contains only presentation languages: $\mathcal{L}_d \subset \mathcal{L}^P$.

For example, consider the document illustrated in Listing 3.1, which uses the two XML languages L_1 and L_2 that are associated to the namespaces `http://layout.org` and `http://furniture.org`, respectively. L_1's language author should associate L_1 to a semantics definition I_1, which defines its interpretation as a tabular information presentation. On the contrary, the semantics I_2, which are associated to language L_2, should define its interpretation based on the various furniture characteristics.

Therefore, $I_1 \in \mathcal{I}^P$ and $I_2 \notin \mathcal{I}^P$, because I_1 defines the L_1 interpretation, according to its tabular layout presentation to the document user. I_2 defines the L_2 interpretation, according to the corresponding abstract furniture information. Therefore, L_1 is a presentation language ($L_1 \in \mathcal{L}^P$) and L_2 is not a presentation language ($L_2 \notin \mathcal{L}^P$). The document in Listing 3.1 is not a presentation document, because it contains the `f:table` element construct of L_2, which is not a presentation language. If the `f:table` elements where removed, the document would become a presentation document, because it would only contain constructs of the presentation language L_1.

The above example is based on the semantics that the language authors "should" provide, because the interpretation of a language only depends on its associated semantics and not on its construct names or the common understanding of their interpretation. For instance, the language author of L_2 could interpret the `f:table` element as the presentation of a letter "T", which is either brown, if the material is wood, or grey, if the material is iron. Such an interpretation converts L_2 to a presentation language. Nevertheless, while such an interpretation is not invalid, it does not follow the common human understanding of the semantics of L_2 constructs.

The presentation documents definition does not relate to the feasibility of presenting an XML document. Specifically, there are multiple methods to present all XML documents, such as presenting their source, their DOM tree, or their mapping to a set of natively supported constructs. However, only the presentation documents have a primary interpretation that *is* their presentation, as opposed to an interpretation that *can be associated* with a presentation.

3.1.4 Document processing

The defined XML semantics represent the abstract notion of a language author's intentions and not a physical representation of processing information. The term *processing model semantics* will refer to the interpretation of a semantics definition, within the context of a processing model. $\mathcal{I}_P$ is the set of all processing model specific implementations of all XML language semantics that can be implemented for the processing model P. An XML processing model P defines how to process an XML document, according to the corresponding processing model specific semantics of its languages.

Processing model semantics ($\mathcal{I}_P$): $\forall I \in \mathcal{I}$, iff there is a semantics implementation I_P of the semantics I for the processing model P, then $I_P \in \mathcal{I}_P$

Processing models ($\mathcal{P}$): $\forall d \in D_Q$, where for all the languages in $\mathcal{L}_d = \{L_1, \ldots, L_n\}$ there are the respective processing model's P semantics $\{I_{P1}, \ldots, I_{Pn}\}$, P defines how to locate, combine and apply these semantics to interpret d, according to the processing model specific interpretation of the languages in $\mathcal{L}_d$. $\mathcal{P}$ denotes the set of all processing models.

For instance, the processing semantics of an XML language can consist of a schema and a transformation specification. The latter can be the presentation interpretation of the language, for an application with a set of natively supported languages $\mathcal{L}_p$, by mapping all its constructs to their equivalent $\mathcal{L}_p$ constructs. The corresponding processing model would be responsible for locating and using the above semantics, for both validating a document and for transforming it into its presentation interpretation.

XML *document processing* can be defined, in respect to a processing model, as the location, combination and application of the processing model semantics that correspond to a document's languages. A *Web browser* can be informally defined as a combination of one or more processing model implementations that process presentation documents, a set of arbitrary mechanisms for the presentation of non-presentation documents and a set of additional mechanisms for the presentation of non-XML documents. As described in the next section, non-presentation and non-XML documents processing is outside the scope of this thesis. Therefore, a formal definition of the term *Web browser* is not necessary, because this thesis only focuses on the defined *processing models* and *document processing*.

Document processing: The processing of a document $d \in D_Q$ by a processing model $P \in \mathcal{P}$, when there are P specific semantics for all languages in $\mathcal{L}_d$, is the location, combination and application of the above semantics by an implementation of the processing model P.

3.2 The scope of this thesis

The presentation of namespace qualified XML documents (D_Q) is separated into the presentation of the presentation documents (D^P) and the non-presentation documents ($D_Q - D^P$). There is a document $d \in D^P$ for every conceivable information presentation, because D^P includes all possible uses of all presentation languages. Documents in $D_Q - D^P$ include constructs of non-presentation languages, and their presentation can be context dependent. The presentation of any non presentation document can be defined as a presentation document, since D^P includes all possible presentations. Consequently, studying the presentation of XML documents requires to firstly study the presentation of documents in D^P.

This thesis will primarily focus on presentation documents, because their processing provides the foundation for presenting all XML documents, covers all possible presentations and allows the utilisation of the presentation domain constraints. Specifically, the presentation domain constraints cannot be used for the generic presentation of XML documents, because their presentation can be context dependent. XML documents can describe any information that can be used in the context of any application, and it is difficult, or even impossible, to define the corresponding constraints. Nevertheless, several of the proposals in this thesis also apply to the processing of non-presentation XML documents.

> *This thesis focuses on XML processing models and the processing model semantics that are necessary and sufficient for processing presentation documents, according to the Web design principles.*

Presentation documents are namespace qualified documents; unqualified documents are outside the scope of this thesis. As stated in Section 3.1.1, such a restriction is necessary for developing a generic processing model for the Web.

The remainder of this thesis focuses on the domain specified by the above definition. In order to avoid cluttering this thesis with excessive term qualifications, unless stated otherwise, the terms *XML document*, *XML language* and *semantics* will refer to the terms *presentation XML document*, *presentation XML language* and *presentation semantic*, respectively.

3.3 The hypothesis

Most user initiated processing of Web information results in information presentation by either directly presenting the information or presenting the result of another form of information processing, such as a query. XML presentation languages are the successors of HTML, which focuses on human consumable hypertext, and they should provide the means for information presentation, through the interface of a browser. The wide applicability of information presentation, within the Web, and the relationship of XML to HTML necessitate an XML processing model for presentation XML documents.

Presentation document processing is an important problem, but, as Chapter 2 concluded, current approaches are neither sufficient for the variety of XML documents nor adequate for the Web. The lack of well defined presentation domain constraints, impedes the creation of generic XML processing models, even if existing approaches and browsers provide a wealth of functionality. Specifically, there are several prominent approaches and ways to combine them for individual aspects of document processing. However, there is no generic processing architecture that utilises the constraints of the presentation domain, in order to allow the combination of the individual approaches, towards a generic processing model for presentation documents.

Hypothesis: *The presentation document processing domain is sufficiently restrictive to allow the development of generic processing models, which are adequate for the Web and can process a significant subset of current and future presentation documents. The combination and extension of existing technologies, in a way that utilises the presentation document domain constraints, can form the basis of such generic processing models.*

The remainder of this thesis focuses on proving the above hypothesis, using a preprocessing framework and a preprocessing architecture proposal. Specifically, the next chapter proposes a presentation document preprocessing framework, which identifies the necessary presentation document validation and transformation functionality for the Web. The preprocessing framework development is solely based on applying the Web design principles to the scope of this thesis; consequently, it acts as both a foundation for building and as a measure for evaluating preprocessing architectures. Chapters 5 to 9 introduce individual proposals for each preprocessing sub-model, and Chapter 10 combines all the proposals to the XMLPipe preprocessing model. XMLPipe is a generic processing model that combines and extends existing validation and transformation approaches, in order to allow XML language integration and presentation adaptation, according to the adaptation requirements: the target device capabilities and the document author and user requirements.

XMLPipe utilises the presentation domain constraints to propose a generic integration model that covers a significant subset of the presentation documents. The proposed integration model allows the combination of independently developed processing semantics for validating and transforming presentation documents.

XMLPipe solely focuses on presentation document validation and transformation and does not include the presentation component that is responsible for rendering a document's presentation. However, as stated in the Chapter 1, the enhanced processing functionality, achieved by exploiting the presentation domain restrictions, allows the extrapolation of our observations to cover the whole spectrum of the hypothesis.

3.4 Concluding remarks

Well defined terminology and concepts are fundamental for a sound discussion of XML processing models and for an investigation of the presentation domain constraints. However, most of the commonly used Web and XML terminology is either underdefined or ambiguous. This chapter defined the necessary XML processing terminology for the rest of this thesis and used it to define the scope of this thesis and our hypothesis.

The introduced terminology focuses on namespace bound documents and languages, because namespace qualified constructs are necessary for a well defined relationship between a document and its corresponding languages. Such a relationship is essential for deriving the interpretation of a document from the semantics of its individual constructs.

The introduced definitions establish the separation and relationship between several ambiguous concepts. Specifically, the abstract notion of semantics describes the intended interpretation of a language's constructs by its language author. In contrast, the processing model semantics are processing model specific instances of a language's semantics. The subsets of presentation semantics and presentation documents also allow the unambiguous separation between the processing of presentation language constructs and the generic presentation of any language construct. Additionally, the definition of presentation constructs according to their associated semantics, as opposed to the implicit language semantics, clarifies the separation between a language and its semantics.

Document presentation is an essential form of information processing for the Web, and the processing of presentation documents is the foundation of presenting any XML document. Section 3.2 restricted the scope of this thesis to the processing of presentation documents. Section 3.3 expressed our hypothesis, which focuses on the feasibility of generic presentation models for the Web, when they utilise the constraints of the presentation domain.

The following chapters proceed to the support of our hypothesis, using the foundation set by the previous chapters' literature review and this chapter's definitions. The hypothesis support will focus on the validation and transformation processing steps. Specifically, this thesis will firstly introduce a preprocessing framework that defines the core preprocessing components and their corresponding requirements. Subsequently, it will proceed to individual proposals for all preprocessing components and their combination into the XMLPipe preprocessing model, which provides the foundation for supporting our hypothesis.

Chapter 4

A preprocessing framework

Existing XML processing approaches offer a wealth of functionality, but the lack of a generic processing model impedes their combination towards generic processing of presentation documents. Additionally, current Web browsers expose a rich set of natively supported languages $\mathcal{L}_p$, but the browsers are neither adequate for generic content adaptation nor provide mixed namespace validation, transformation and presentation.

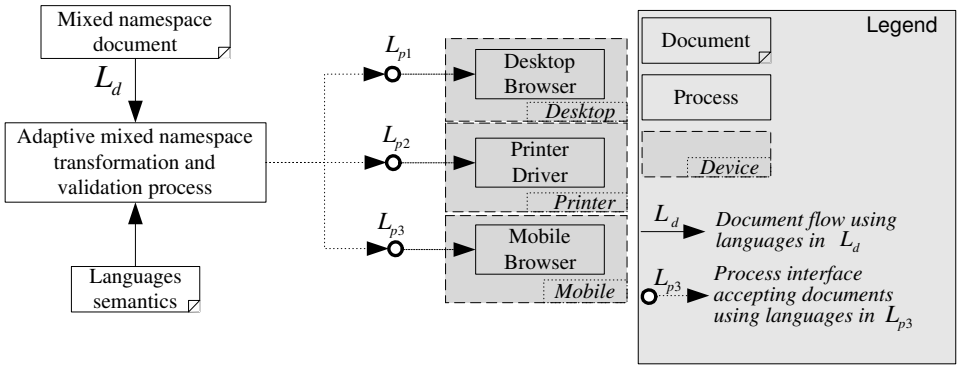

Figure 4.1: A preprocessing approach

Preprocessing approaches do not cover all presentation document processing functionality, because they only provide document validation and transformation. However, they allow the reuse of existing browsers, within a generic processing model for XML. As Figure 4.1 illustrates, an XML preprocessor can offer the necessary validation and transformation functionality to provide generic adaptation, validation and transformation of mixed namespace documents. For instance, the *adaptive mixed namespace transformation and validation process* can process a mixed namespace document d, according to the semantics of its languages, and transform it to its optimal interpretation, for several device and browser combinations.

Preprocessing approaches are beneficial for introducing a new document processing paradigm: they do not require the time consuming development of a complete presentation architecture; they do not require the document users to migrate to alternative Web browsers, and they can reside on both the server and the client side. The remainder of this thesis focuses on supporting our hypothesis, using a preprocessing architecture.

Our hypothesis refers to the development of generic processing models that are adequate for the Web. The last chapter's term and scope definitions defined the boundaries of the presentation document processing domain, but they did not provide an adequacy benchmark for generic XML processing models. A preprocessing framework that defines all necessary preprocessing functionality can provide the foundation for evaluating preprocessing approaches. Such a framework must identify the individual subproblems that a preprocessing approach must solve, their corresponding requirements and their composition towards a generic preprocessing model.

This chapter identifies the top level requirements of a generic preprocessing model for XML and refines them into a framework that consists of the core processing components and their associated requirements. Section 4.1 summarises the core problems addressed during the definition of the framework requirements. Section 4.2 proceeds to the identification of the top level framework entities, and Section 4.3 proceeds to include the necessary additional entities. Finally, Section 4.4 refines the expressed entities requirements into the top level framework architecture and into the requirements of its individual components. The resulting separation of the preprocessing issues sets the context for the subsequent discussion and evaluation of our proposed preprocessing model: the XMLPipe preprocessing model.

4.1 Towards a generic preprocessing framework

Underdefined and unrestricted areas, such as the processing of XML documents, impede the expression of an indisputable set of requirements, which is necessary for a sound preprocessing framework. The proposed framework avoids unnecessary assumptions by only using the widely accepted Web design principles and the requirements of the preprocessing external entities, within the context of the constraints and definitions of Section 3.2.

The requirements of the external entities, such as the document author and document user, are essential for formulating a generic preprocessing framework. However, the lack of existing generic XML processing models, which conform to the Web design principles, impedes the objective expression of such requirements and the proof of their correctness. Current human requirements are adapted to the existing processing models; threfore, traditional requirements identification methods, such as statistical analysis of actual human requirements, can lead to deficient processing models. For instance, the Web applications position paper[Fou04] of the Mozilla foundation and the Opera organisation, which serve a significant Web user subset, condemns extensive namespace usage, because it confuses document authors. Additionally, it suggests that XML languages should include all necessary foreign language XML constructs that can be used in conjunction with them. However, as Section 2.4.3 described, such namespace assimilation approaches result in monolithic specifications and redundant constructs, and they are not adequate for the Web. Consequently, they are not adequate for generic XML processing.

The proposed framework eliminates such obsolete requirements, because it does not use actual human entities requirements. In contrast, it uses well established Web, XML and software design principles to support the set of external entities requirements it uses. Such an approach may lead to an incomplete or subjective set of requirements, but it ensures that the resulting framework will be sufficiently generic for the Web.

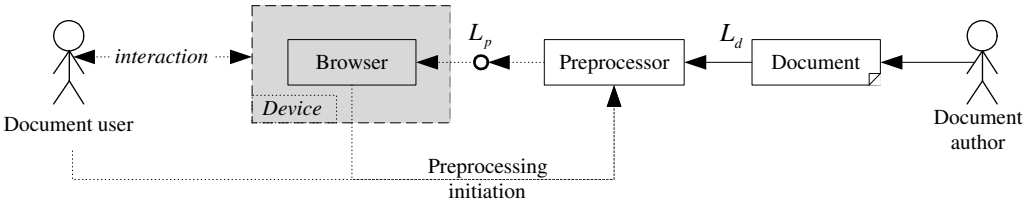

Figure 4.2: Preprocessing framework: top level entities

Document author requirements and assumptions
Single presentation XML document input
Document associated with a URL
Open set of languages
Well defined integration model
Human oriented language semantics descriptions
Authoring validation
Inline presentation control capability

Table 4.1: Document author requirements and assumptions

4.2 Top level entities

This section initiates the investigation of the external entities requirements. In order to avoid introducing unnecessary assumptions, it only addresses the top level entities that are explicitly related to a preprocessing model. Their investigation will indicate the necessary additional entities. The subsequent collective study of all identified requirements and the Web design principles will conclude with a concise set of preprocessing model requirements.

As illustrated in Figure 4.2, the top level entities are the document author, the document user and the documents user's browser and device. The document author produces a document d that uses the languages in $\mathcal{L}_d$. The document user interacts with the document presentation, using a browser that exposes an interface, which consists of the languages in $\mathcal{L}_p$. Either the browser or the document user can initiate the document preprocessing.

4.2.1 Document author

Table 4.1 summarises the assumptions and requirements of the document author. According to the scope of this thesis, the document d is a presentation XML document. An XML document can be the aggregation of multiple documents, since XML documents can represent any information. As described in Section 2.3, assuming that a preprocessing model only processes a single document at a time does not harm its generality. Additionally, a document author must associate that single document with a URL, because URLs are necessary for Web information identification and location.

Document user requirements and assumptions
Only provides the document URL and an optional set of preferences
Expects an adequate presentation for the user preferences and the capabilities of the browser and the device
Efficient processing
Presentation consistency

Table 4.2: Document user requirements and assumptions

As described in Section 1.4.1, according to the least power Web design principle, each XML language must cover a single data representation domain. A fixed set of languages can cover all necessary presentation functionality, but, according to the modularity and the test of independent invention principles, the document author must be free to choose any set of existing or future presentation XML languages. Therefore, a preprocessing model must support an open set of languages. As described in Section 2.4.3, namespace assimilation and integration profiles are not adequate for the Web and for an open set of languages. In contrast, generic authoring and processing of mixed namespace documents requires a generic integration model, which defines how to combine the individual language constructs and processing semantics.

In a similar manner to a browser, a document author cannot a priori know the semantics of all XML languages. A well defined mechanism for locating human oriented descriptions of language semantics is necessary, to allow the wide deployment of the multitude of existing and future languages. Additionally, exhaustive testing and debugging are not applicable for documents that combine an open set of languages and that can be processed according to a multitude of adaptation requirements. Therefore, a preprocessing model must offer a document validation mechanism that is independent of the adaptation requirements. Such a mechanism is essential for performing the *authoring validation*, which was introduced in Section 2.2.

Document authors span the spectrum between the two extremes of *designers* and *structuralists*, as described in [BBMS99]. Structuralists focus primarily on the conveyed information and its high level structure. Designers focus more on the aesthetics of the information presentation and require precise control over the document presentation. Consequently, a processing model that is adequate for the variety of document authors must allow, but not require, fine grained inline specification of both document processing and adaptation.

4.2.2 Document user

Table 4.2 summarises the assumptions and requirements of the document user. The *document user* uses a browser, which runs on a device, to interact with the document presentation. A preprocessing model must require the minimum user input, because Web content should be available to everyone, and the document user assumptions must be the minimum necessary. Specifically, the document user must provide the document URL, in order to identify a document, and an optional set of user preferences, for personalising the document presentation. The preprocessing model is responsible for

Device/Browser adaptation requirements
Adaptation for a variety of devices
Adaptation for a variety of $\mathcal{L}_p$ sets
Adaptation for a variety of browser integration models

Table 4.3: Device and browser adaptation requirements

retrieving all the necessary information for generating an optimal document representation, according to the browser capabilities, device capabilities and user preferences.

The processing of a document may vary according to its intended usage, such as for browsing or for printing. However, such variations are subproblems of document processing for a diverse set of capabilities and requirements. For instance, document processing for printing is interchangeable to document processing for a printer target device, as illustrated in Figure 4.1.

Finally, timely and consistent information processing are necessary qualities for most information media. The interactive nature of Web information presentation and the nearly instant presentation of the widespread HTML and XHTML documents necessitate efficient document processing. Consistency is necessary for ensuring that a document's presentation remains the same when the presentation parameters, such as the user preferences and the browser capabilities, remain the same.

4.2.3 Target device and browser

Table 4.3 summarises the necessary adaptation requirements for supporting the multitude of Web devices and browsers. Specifically, an adaptation mechanism must accommodate the wide range of devices that can access the Web. In a similar manner, there is a multitude of browsers, which can natively support a wide range of languages and integration models. Generic document preprocessing must take into account the $\mathcal{L}_p$ set and the integration models that a browser supports.

4.3 Additional entities

Both the browser and the document preprocessor can natively support only a limited set of XML languages. However, according to the aforementioned document user and author requirements, a document can use an open set of languages, and neither the document user nor the document author have to provide the necessary processing and adaptation information. This section introduces the *language author* and the *semantics author* external entities, in order to study the language creation process and to investigate the location of all necessary processing semantics. Figure 4.3 incorporates the *language author* and the *semantics author*, within a more fine grained illustration of the interactions between the external entities and a preprocessor.

The language author creates a language and, as Section 2.2.3 described, must provide its authoritative interpretation. Specifically, the language author must provide a language's semantics, which, within the scope of this thesis, is identical to its interpretation. In contrast, as Figure 4.3 illustrates, the semantics author is responsible for providing the processing model specific implementations of a language's semantics.

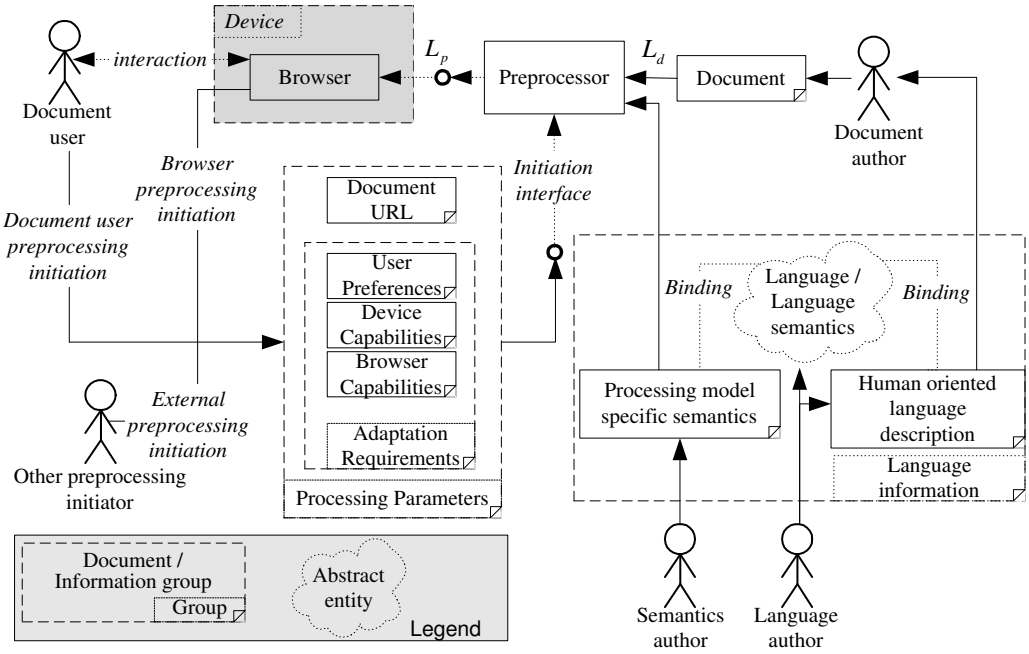

Figure 4.3: Preprocessing framework: all external entities

Language authors may also provide processing model semantics, but separate semantics authors are necessary for a processing model that conforms to the Web design principles. Specifically, a language author that is solely responsible for a language's processing information becomes a central point of control/failure, which is against the Web decentralisation principle. Additionally, according to the test of independent invention, the language author would have to provide the processing model specific semantics for all existing and future processing models. Such an assumption is not realistic for a freely evolving information system, such as the Web.

Each language corresponds to exactly one language author, who is responsible for all the processing model independent resources that are central to a language: its namespace URI, the administration of the corresponding Web space if the URI is a URL, and the human oriented language description. In contrast, there can be multiple semantics authors for each language, and they are responsible for both providing the necessary processing model semantics and associating them with the language.

Table 4.4 summarises the requirements of the language and semantics authors. Any dependencies between separate languages or semantics implementations share the drawbacks of integration profiles, and they are against the principle of independent invention, because they can restrict the languages that can be combined in a document. Consequently, language authors require a method to independently define XML languages and, in a similar manner, the semantics authors require a method to independently define processing model semantics. The independent languages and semantics definitions necessitate a well defined method to associate high level language descriptions and processing model semantics to the corresponding languages. Such associations must enable the location of all necessary information, by both the document authors and the

Entity	Requirements
Language author	Independent definition of languages
	Well defined human oriented description binding
Semantics author	Independent semantics development
	Well defined processing semantics binding
	Multiple processing semantics binding
	Multiple processing models binding

Table 4.4: Language and semantics authors requirements

Adaptation requirements representation requirements
Composite representation
Conflicting requirements resolution
Extensible representation

Table 4.5: Requirements of an adaptation requirements representation

processing models. According to the test of independent invention, the semantics implementation associations must allow several implementations for multiple processing models, as well as multiple implementations for the same processing model.

A preprocessing model must generate an optimal document interpretation, according to the preprocessing adaptation factors: the user preferences and the browser and device capabilities. As Figure 4.3 illustrates, document preprocessing can be initiated by the document user, the browser (as a response to a document user interaction with the browser) or another external entity, such as a Web server that preprocesess a document. A well defined preprocessing initiation method requires a well defined adaptation requirements representation, because the preprocessing initiation entity must provide all necessary adaptation information.

Table 4.5 summarises the adaptation information representation requirements. As described in Section 2.4.4, requirement sets composition is necessary for efficiently communicating the requirements and for combining the requirements of several independent sources: the browser, the device and the document user. Since the requirement sources are independent, adaptation requirement conflict resolution is necessary. Finally, extensibility is necessary, because the unrestricted nature of the Web devices and browsers impedes the development of a fixed set of adaptation parameters.

4.4 Framework architecture and requirements

The previous sections described the requirements of a preprocessing model's external entities. Each identified requirement relates to the design of either the processing model as a whole or its individual components. Table 4.6 groups all external entities requirements, according to their corresponding subset of preprocessing functionality. This section describes the development of a preprocessing framework that has a well defined set of components and requirements, based on the above external entities requirements.

Functionality	Entity	Requirement
Assumptions		Single input XML document
		URL identifiable/retrievable document
Global		Web design principles
	Document user	Efficient processing
		Presentation consistency
Int. model	Document author	Open set of languages
		Generic integration model
		Low and high level presentation specification
	Language author	Independent definition of languages
	Semantics author	Independent development of semantics
Binding	Document author	Human oriented semantics descriptions
	Document author/user	No necessary inline presentation information
		No necessary inline adaptation information
	Document user	Well defined way to provide adaptation requirements
	Language author	Well defined human oriented description binding
	Semantics author	Well defined processing semantics binding
		Well defined way to specify semantics
		Multiple semantics specifications
		Multiple processing models
Validation	Document author	Authoring validation
Adaptation	Document user	Adaptation according to browser capabilities, device capabilities and user preferences
	Device/Browser	Adaptation for a variety of devices
		Adaptation for a variety of $\mathcal{L}_p$ sets
		Adaptation for a variety of integration models
	Preferences representation	Well defined representation
		Composite representation
		Extensible representation
		Conflicting requirements resolution

Table 4.6: Assumptions and external entities requirements, grouped according to their corresponding functionality subsets

4.4.1 Validation

As described in Section 2.2, a well defined validation process enables authoring validation and assists the development of document processing components, because it suppresses the requirement for custom validation logic. Figure 4.4 illustrates the preprocessing framework validation. A generic validation model must incorporate several schema languages, because of the test of independent invention and because no existing validation approach is more expressive than the others, as described in Section 2.2.1. Therefore, the validation model must allow the instantiation of several *schema validators* that correspond to the individual validation approaches.

The *validation driver* component is responsible for driving the validation process, and it delegates all validation requests to the *integration model validation driver*, because the validity of a mixed namespace document depends on its integration model. The integration model validation driver drives all relevant schema validators, according to the corresponding integration model.

All validation components must be able to process document portions, in addition

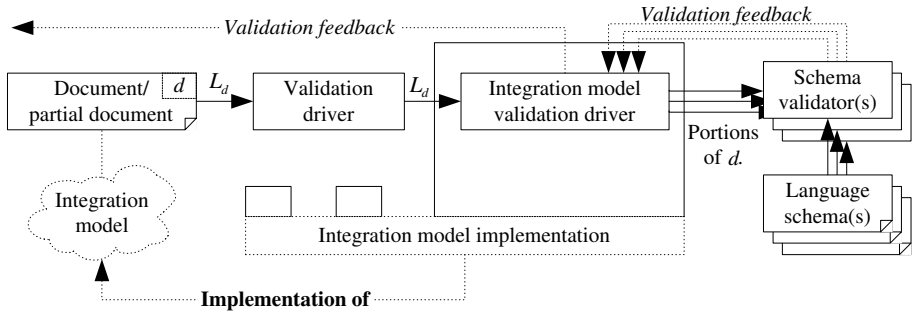

Figure 4.4: Preprocessing framework: Validation module

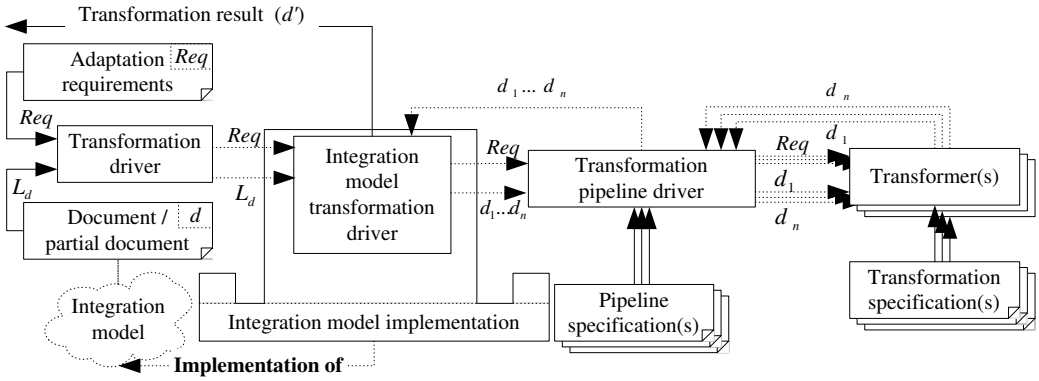

Figure 4.5: Preprocessing framework: Transformation module

to complete documents. The individual portions of a mixed namespace document can correspond to separate languages; therefore, they can require separate schema validator instances. Both the validation driver and the integration model validation driver must be able to process individual document portions, because d can be a partial document, during the interoperation between the transformation and validation models. Specifically, a transformation process must be able to validate its input, and transformation processing can also apply to individual document portions, as the subsequent transformation model investigation will illustrate.

4.4.2 Transformation

Figure 4.5 illustrates the top level preprocessing framework transformation model, which transforms a document d to its optimal representation d', according to a set of adaptation requirements Req. Well defined methods for independently developing transformation specifications, integrating a multitude of transformation technologies and processing document portions are necessary, because of the requirement for independent processing semantics and the test of independent invention principle.

In a similar manner to the validation model, the *transformation driver* drives the transformation of d by delegating the transformation request to the *integration model transformation driver*, which is specific to the integration model of d. The requirement for multiple transformation technologies also covers transformation pipelines support, since a pipeline is a transformation. However, built-in transformation pipelines are beneficial, because they provide the foundation for combining multiple transformation technologies. Consequently, the integration model transformation driver delegates all transformation requests, for the separate document portions $d_1 \dots d_n$, to the *transformation pipeline driver*, instead of directly calling the individual *transformers*. Subsequently, the *transformation pipeline driver* calls the necessary transformers, according to the corresponding *pipeline specification*.

A transformation model, which fulfils the document user's adaptation requirements, must have access to the set of adaptation requirements *Req*. Moreover, it must provide well defined methods to use the adaptation requirements, within the transformation pipelines and the transformation specifications, in order to enable transformations, which are sensitive to the adaptation requirements.

There is no reason to assume that a document transformation is a one step process, where its output requires no further processing. A recursive transformation mechanism is necessary for multiple-step transformations. Moreover, it is beneficial for processing naturally recursive languages and for reusing the functionality of existing XML languages. For instance, consider the documents illustrated in Listing 4.1. The semantics of the `n1:p` construct, in `authorsOut.xml`, is to present its content as a paragraph. Moreover, the semantics of the `n2:description` construct, illustrated in `authors.xml`, is to present the contents of all its `n2:dItem` constructs as a list of paragraphs that follow the content of the `n2:name` construct. Recursive transformations allow reusing the `n1` constructs for defining the `n2:description` semantics. For instance, a transformer that corresponds to the `n2` namespace can map `authors.xml` to `authorsOut.xml`. Subsequently, the transformer that corresponds to the `n1` namespace can transform `authorsOut.xml` to its most appropriate representation, according to set of adaptation requirements.

Additionally, consider the transformation of the `n3:imp` construct, illustrated in `import.xml`, which must insert the referenced document (`d.xml`)[1] in its place. The processing model must recursively apply the `n3:imp` transformation for processing all `n3:imp` occurrences, in both `import.xml` and the imported `d.xml`.

4.4.3 Binding

A preprocessing model must automatically locate the necessary validation and transformation semantics, because it cannot natively support an open set of XML languages and no external entity is required to provide document processing information. Therefore, a preprocessing model must include a binding model that specifies how to organise, distribute and locate the necessary processing model semantics. Figure 4.6 illustrates the top level organisation of the processing semantics and the interoperation between the binding process and the other preprocessing components.

The processing semantics must be associated with XML languages and not with

[1] `d.xml` is not a well formed XML document, because it contains two root elements. However, within this example, it is used as a text file, which results in a well formed document by substituting the `n3:imp` construct.

```
<n1:doc>
  <n1:section
    name="D. author">
  <n1:p>
The entity that
creates an XML
document.
  </n1:p>
  <n1:p>
The document author
has a set of
requirements.
  </n1:p>
  </n1:section>
</n1:doc>
```

authorsOut.xml

```
<n2:description>
  <n2:name>
D. author
  </n2:name>
  <n2:dItem>
The entity that
creates an XML
document.
  </n2:dItem>
  <n2:dItem>
The document author
has a set of
requirements.
  </n2:dItem>
</n2:description>
```

authors.xml

```
<n2:description>
  <n2:name>
D. author
  </n2:name>
  <n3:imp ref="d.xml"/>
</n2:description>
```

import.xml

```
<n2:dItem>
 <n3:imp ref="d1.xml"/>
</n2:dItem>
<n2:dItem>
 <n3:imp ref="d2.xml"/>
</n2:dItem>
```

d.xml

Listing 4.1: Documents that can benefit from recursive transformations

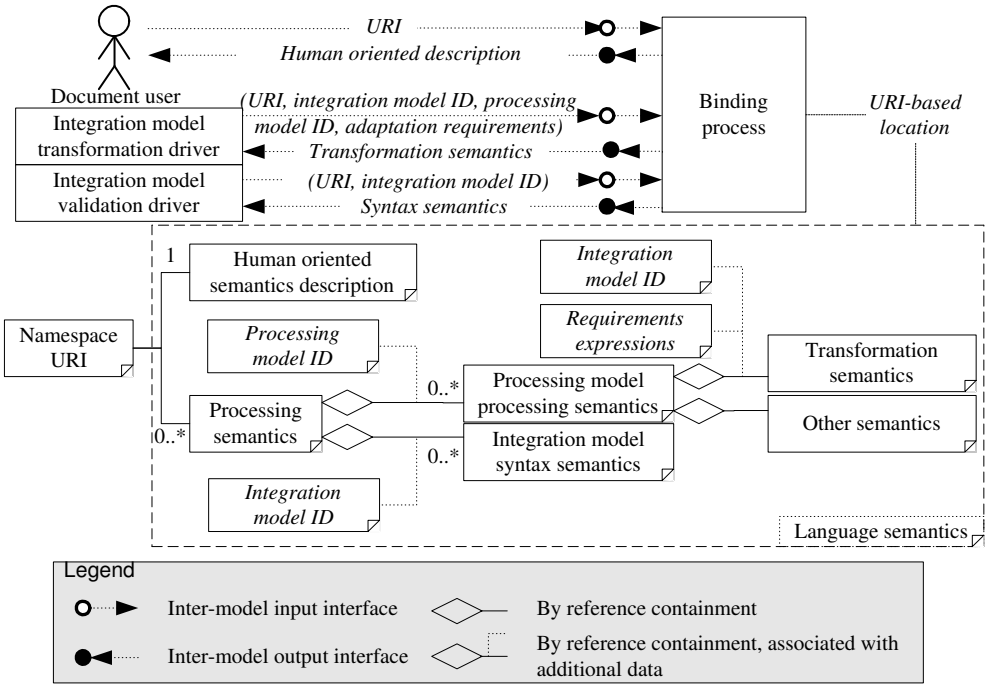

Figure 4.6: Preprocessing framework: Binding module

documents, because the document author is not required to provide processing information and document based processing association is problematic, as described in Section 2.2.3. Therefore, the semantics organisation must be based on the namespace URIs, since they uniquely identify namespace bound XML languages. Namespace URIs must also be the foundation of the principal semantics location mechanism, because they ensure the wide availability of independently developed processing semantics for both document authors and processing models.

A preprocessing model, in addition to its principal semantics location mechanism, must support alternative location mechanisms, in order to avoid central points of failure. For instance, consider that the principal semantics location mechanism for XHTML used its namespace URL, for locating all necessary XHTML processing semantics. If a processing model did not support an alternative location mechanism, the failure of either a Web server or a client's connectivity could prohibit the presentation of XHTML documents, for the duration of the failure. Consequently, the proposed semantics organisation, illustrated in Figure 4.6, accommodates alternative location mechanisms by associating each namespace URI to multiple semantics specifications.

The document user and both the integration model validation and transformation drivers must provide a language's URI to the binding process, because it is the basis of the principal semantics location mechanism. Specifically, the document user uses the binding process to retrieve the human oriented language descriptions. The integration model validation driver requires a set of syntax semantics. In addition to the corresponding namespace URI, it must specify the document's integration model, because separate integration models can require different validation processing semantics.

Finally, the integration model transformation driver, in addition to a language's URI, must provide the integration model, the processing model and the adaptation requirements, in order to retrieve the optimal transformation semantics. In a similar manner to the validation semantics, the transformation semantics can be integration model specific. Additionally, they are also processing model specific, because the transformation outcome depends on the way that a model processes a document. Moreover, a preprocessing model must enable the use of separate transformation specifications for separate sets of adaptation requirements, because document transformation can be adaptation requirements dependent.

4.4.4 Integration model and overall processing

An integration model, which is adequate for a preprocessing approach, must define how to validly nest language constructs, within a mixed namespace document. Additionally, it must define how to combine the processing semantics of the individual languages, in order to process a mixed namespace document. Consequently, the document authoring process, the validation process driver and the transformation process driver must be integration model specific.

A generic integration model cannot use predefined inter-language relationships, between either language constructs or processing semantics. Such relationships do not allow the independent definition of an open set of languages, because they share the problems of integration profiles. As described in Section 2.1.1, integration profiles are against the test of independent invention and exponentially difficult to enumerate for the continuously expanding set of languages. Well defined processing APIs and well defined sets of languages can form the integration basis for a restricted set of languages, such as the $\mathcal{L}_p$ of a presentation component that covers a well defined range

of functionality. However, they are not adequate for the integration of an open set of languages, because they are against the minimalistic nature of XML[BL02a] and they require the enumeration of all possible integration scenarios, for a multitude of abstraction levels. Consequently, a generic integration model must define the necessary construct relationships, without using predefined integration profiles, APIs or sets of languages.

Finally, a preprocessing model must define the necessary orchestration of its individual components, for performing its two principal applications: authoring validation and document transformation, according to a set of adaptation requirements. Specifically, a preprocessing model must define the necessary algorithms for both processing applications and the necessary components interoperation for implementing those algorithms. The latter must also ensure the validity of each transformed document or document portion, in order to assist the development of the individual transformers.

4.4.5 The complete preprocessing framework

Table 4.7 summarises all the requirements that a preprocessing model for presentation XML documents should fulfill. Specifically, it combines the requirements of the external entities, initially summarised in Table 4.6, and the additional requirements introduced in Section 4.4. The proposed XML preprocessing framework consists of the Table 4.7 requirements and the three modules, illustrated in Figures 4.4, 4.5 and 4.6.

The proposed preprocessing framework can be applied for both developing and evaluating generic XML preprocessing models. The preprocessing framework development did not use arbitrary assumptions, but it was solely based on the Web design principles and the external entities requirements, within the boundaries set by the scope of this thesis. Consequently, either the absence of any component illustrated in Figures 4.6, 4.5 and 4.4, or the lack of compliance to any of the Table 4.7 requirements indicates that a processing model is not sufficiently generic for the Web, within the scope of this thesis. Moreover, the proposed component separation assists the development of preprocessing models, because it divides the problem of generic XML processing to smaller and more manageable subproblems with well defined requirements.

4.5 Discussion

The remainder of this thesis will use a preprocessing model to prove our hypothesis. Specifically, it will illustrate that the development of *generic* processing models that are *adequate* for the Web is feasible, when utilising the constraints of the presentation domain. However, the definitions of Section 3.2, which describes the domain of presentation document processing, are not sufficient for identifying whether a preprocessing approach is adequate for the Web.

This chapter proposed a preprocessing framework, which established the functionality subsets of a preprocessing model and their corresponding requirements. Such a framework is essential for proving our hypothesis, because it provides the means to identify the adequacy of a preprocessing approach for the Web. Additionally, the identified functionality subsets assist both the development of new preprocessing models and the independent investigation of their components.

Functionality	Requirements
Assumptions	Single presentation XML document
	URL identifiable/retrievable document
Global	Web design principles
	Efficient processing
	Presentation consistency
Validation	Authoring validation
	Multiple validation technologies
	Validation of documents and document portions
Int. model validation	Independent definition of schemas
	Well defined combination of independent schemas
	Orchestration of the individual validators
Transformation	Multiple transformation technologies
	Transformation of documents and document portions
	Built-in transformation pipelines
	Adaptation requirements sensitive transformation components
	Well defined way to use adaptation requirements in both the pipelines and the transformation specifications
Int. model transformation	Independent transformation definitions
	Well defined combination of independent transformations
	Orchestration of the individual transformers
	Recursive transformation processing
Int. model	Open set of languages
	Low and high level presentation information in the document
	Independent definition of languages
	Well defined combination of separate languages' constructs
	No predefined construct relationships
	No use of predefined exhaustive APIs or sets of languages
Int. model processing	No language specific interoperation between processing components
	Well defined authoring validation and adaptation algorithms
	Well defined component orchestration to implement those algorithms
	Ensure valid transformation input
Binding	No necessary presentation information in the document
	No necessary adaptation information in the document
	Well defined organisation, distribution and location of human oriented descriptions and processing model semantics
	URI-based semantics association and principal location mechanism
	URI based human oriented semantics location
	Validation semantics location and organisation based on the language URI and the integration model
	Transformation semantics location and organisation based on the language URI, the integration model, the processing model and the adaptation requirements
	Alternative location mechanisms
Adaptation	Well defined adaptation requirements representation
	Adaptation for a variety of $\mathcal{L}_p$ sets
	Adaptation for a variety of browser integration models
	Composite representation
	Extensible representation
	Conflicting requirements resolution

Table 4.7: Preprocessing framework requirements

4.5.1 Evaluation

A sound preprocessing framework must use indisputable requirements and assumptions, in order to provide the foundation for evaluating the adequacy of preprocessing approaches for the Web. The proposed framework is based on the Web design principles and the requirements of the external entities. The soundness of the Web design principles is well established, because they have been fundamental to the success of the Web. There is no reliable source for the requirements of the external entities, because human requirements are adapted to the accustomed processing practices, and they can lead to deficient processing models. Therefore, the identified requirements are derived from assumptions that are based on the Web design principles. The correctness and sufficiency of the requirements cannot be proven. However, since they are based on the Web design principles, they should reflect the adequacy of a preprocessing model for the Web.

The derivation of the proposed framework, from the identified requirements and assumptions, is also sound. Specifically, the framework consists of the identified functionality areas, their corresponding requirements and Figures 4.4, 4.5 and 4.6. These three figures define the necessary interoperation between the individual components of each functionality area. The proposed functionality areas and their requirements are derived by grouping the initial set of requirements and assumptions. Additionally, the proposed set of components and their interoperation, illustrated in the three figures, are a direct consequence of the grouped requirements and the observations of the existing processing approaches, described in the literature review (Chapter 2). Consequently, the processing framework is as sound as its initial requirements.

The defined component interoperation and requirement grouping enable the independent evaluation and proposal of the individual preprocessing components. For instance, the adequacy of a validation model can be evaluated by the extent to which it fulfils the validation requirements of Table 4.7 and it fits within Figure 4.4. In a similar manner, the same requirements and figure can be used as a guideline for developing a new validation model.

A framework that is adequate for evaluating preprocessing approaches must provide a *sufficient* set of requirements. The proposed framework attempts to cover all the spectrum of XML preprocessing, within the scope of this thesis, but sufficiency is neither adequate nor provable. Specifically, there are neither existing and indisputable sets of preprocessing requirements nor existing generic processing models. Consequently, there is no benchmark for comparing the identified requirements and processing components. Additionally, a preprocessing model contains a multitude of processing components, and it can combine several existing technologies. Exhaustive coverage of all the necessary functionality, at a sufficiently fine grained level to ensure the sufficiency of a preprocessing model, would result to a prohibitive multitude of requirements. Consequently, the proposed framework provides an indication of, and does not prove, the adequacy of a preprocessing model.

In a similar manner, a framework that can be used for evaluating the adequacy of preprocessing approaches, must only require the *necessary* preprocessing functionality. If this is the case, each unfulfilled requirement would indicate a deficiency of a preprocessing model. However, the *necessity* of the introduced requirements cannot be proven, because, as stated above, the framework requirements are sound, but their correctness cannot be proven. Nevertheless, all framework requirements and components are inferred from the Web design principles. Consequently, the lack of adherence of a

preprocessing model to the proposed framework is a good indication, but not a proof, of its inadequacy for the Web.

As a consequence of the above evaluation, the proposed preprocessing framework is derived from a sound set of requirements, but its requirements cannot be proven to be either necessary or sufficient. Nevertheless, it is based on the Web design principles and provides an indication of the adequacy of preprocessing approaches, which is essential to proving our hypothesis. Moreover, it assists the development of original preprocessing proposals.

4.5.2 Evaluation of existing approaches

The literature review of Chapter 2 indicated the core strengths and weaknesses of existing XML processing approaches. However, their evaluation according to the preprocessing framework is beneficial, because its provides a straightforward way to evaluate the adequacy of processing approaches for the Web.

Integration profiles and namespace assimilation approaches are widely used. However, they only address a document's syntax, instead of its processing. Moreover, they do not comply with most framework requirements, because they do not fulfill the simplicity, modularity and least power Web design principles, they are not adequate for an open set of languages and they require predefined language associations. X-Smiles proposes a more loose integration model that allows an open set of languages and does not require their close interoperation. However, the looser integration necessitates the introduction of predefined interfaces, for the interoperation of the individual language processors, and prohibits the development of a well defined validation process. NRL and NVDL are the most prominent integration approaches, and they fulfill the framework requirements, if their context features are not used (because their use is based on predefined language combinations). Therefore, the NRL and NVDL concepts of subtree partitioning and namespace based association of processing semantics can provide the foundation for a generic integration model.

Adaptive document processing requires a well defined representation of the adaptation requirements and a well defined document adaptation mechanism. CSS stylesheets allow the application of a different style, according to the device type. Moreover, the adaptive Cocoon processing uses URI device implications to infer the type of a device. However, neither approach fulfils the framework requirements. CC/PP provides a composite extensible representation of adaptation requirements, and it fulfils all the framework requirements, apart from the requirement for a conflict resolution mechanism. Consequently, CC/PP can form the the basis of a generic adaptation mechanism.

The proposal in [OH02] uses CC/PP profile queries to control a document's processing, and it is the most powerful generic adaptation mechanism. Device independent authoring approaches can offer more powerful adaptation, but they can only be used in highly constrained domains and they are not adequate for an open set of languages and a variety of integration models. In contrast, the above CC/PP based adaptation combines both an adequate representation of adaptation requirements and the potential to support a multitude of languages and integration models.

No existing transformation approaches fulfill the framework requirements of mixed namespace and adaptive transformations. Specifically, there is no existing generic way to transform mixed namespace documents, according to the semantics of their individual constructs. The above CC/PP based adaptation approach enables the adaptive

transformation of a document, but it cannot individually address the document subtrees.

There is a multitude of existing schema languages and validation models, but they can only fulfill the framework requirements within the context of NRL or NVDL. As described above, NRL and NVDL provide the only prominent methods to validate mixed namespace documents, according to independently defined schemas.

The preprocessing framework requires URI based association and primary location of semantics. Most schema bindings, such as in the XML Schema and NRL, use URI based associations. However, their location mechanisms are either document specific or not well defined. RDDL is the only generic binding method that uses URIs for both semantics association and location. The preprocessing framework also requires secondary location mechanisms. XVM provides a distributed location mechanism that can use several repositories to store URI associations. However, it is not based on the URIs of the individual resources.

Most of the aforementioned approaches do not cover their corresponding framework requirements. However, there is also no existing generic method to combine them into a more powerful document processing approach. Consequently, the existing XML processing approaches do not provide an adequate model for the generic processing of presentation XML documents.

4.6 Summary

This thesis will use a preprocessing model to prove our hypothesis, because preprocessing approaches can provide generic validation and transformation, without the transition and implementation difficulties of complete processing approaches. The proof of our hypothesis requires a well defined method to evaluate the adequacy of the proposed preprocessing model. This chapter proposed a preprocessing framework that enables the evaluation of preprocessing approaches.

The proposed framework defines the components of a preprocessing model, their interoperation and their corresponding requirements. Specifically, a preprocessing model consists of a validation model, a transformation model, an integration model and a binding model. The validation and transformation models are responsible for validating and transforming a presentation document. Both models are specific to an integration model, because it defines how to combine the processing semantics of the individual languages, in order to process a mixed namespace document. The binding model is responsible for locating and retrieving all necessary processing semantics.

The proposed framework can neither be proven correct nor complete. However it provides an adequate method to evaluate preprocessing approaches, in order to prove our hypothesis. Specifically, there is no well defined benchmark for comparing the framework requirements, because there are currently no generic processing models for presentation documents. Traditional requirement identification methods, such as statistical analysis, are not reliable, because Web users are accustomed to the existing practices. Nevertheless, all identified requirements and components are based on the well established Web design principles. Consequently, they can provide a good indication of the adequacy of a preprocessing model for the Web.

The investigation of existing processing approaches, according to the proposed framework, illustrated that there are no adequate approaches for the generic processing of presentation documents. The subsequent chapters will introduce the necessary

processing models to cover all aspects of a preprocessing model.

Chapter 5

XMLPipe integration model

This chapter initiates the XMLPipe preprocessing model investigation by describing the XMLPipe integration model, because an integration model provides the foundation of a preprocessing model. Specifically, it defines the interpretation of a mixed namespace document, according to the interpretation of its constructs. Such a well defined interpretation is necessary for authoring and processing a document, according to the semantics of its individual languages.

The XMLPipe integration is based on three observations, which apply to a significant subset of presentation languages. Firstly, each language has a subset of constructs, the processing of which is independent of their context. The term *handled constructs* will refer to such constructs. Secondly, the processing of document subtrees that are rooted at handled constructs is also independent of their context. Thirdly, the handled constructs can be classified into distinct categories. For each handled construct rooted subtree, the category of its handled construct defines its relationship with its context.

There are currently no generic integration models. Section 5.1 investigates the underlying reasons for the lack of such integration models. It also identifies the most prominent current approaches, which can provide the foundation for the XMLPipe integration model. Section 5.2 investigates the three handled construct observations, and Section 5.3 describes how they can enable the processing of presentation documents. Section 5.3 does not provide the details of document transformation and validation, because they depend on the transformation and validation model, respectively. The corresponding Chapters 7 and 8 will apply the XMLPipe integration model to precisely define the XMLPipe document processing.

5.1 Integration model considerations

Well defined information interpretation requires well defined interpretation of the individual information pieces and of their relationships, as described in Section 2.2.2. Consequently, well defined document processing requires that the processing of the document constructs is well defined and that their processing relationships are also well defined.

Language specifications provide all the necessary information for processing single namespace presentation documents. For instance, consider the presentation document in Figure 5.1. The XHTML specification defines the presentation of all XHTML constructs and their dependencies with their context and their contents. The illustrated DOM tree summarises the presentation dependencies of the first `xhtml:p` construct.

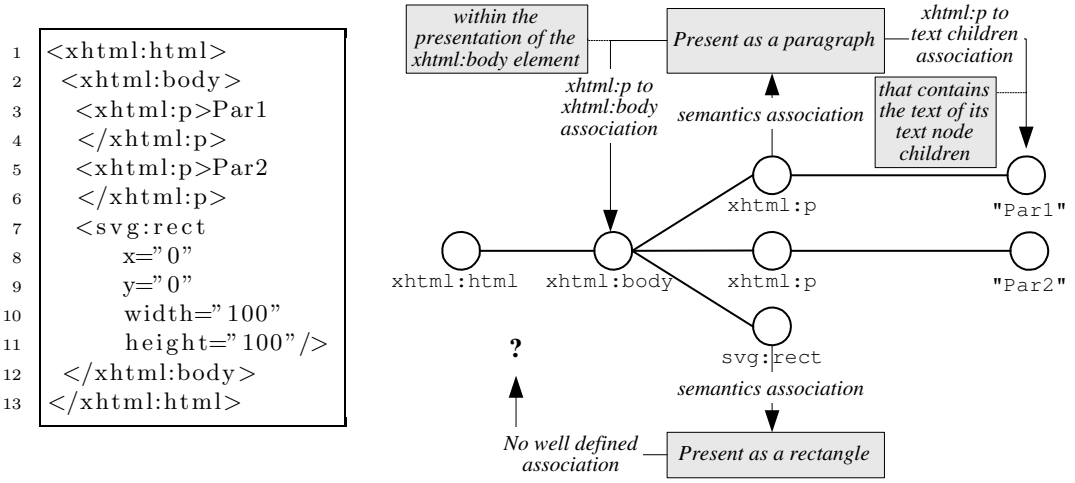

Figure 5.1: Processing associations between language constructs

Specifically, `xhtml:p` must be presented as a paragraph, within the boundaries of its `xhtml:body` parent element. The presentation of `xhtml:p` also depends on its descendants, because it must contain the contents of all its text node children.

However, the language specifications are not sufficient for processing mixed namespace documents. Specifically, the ancestors and the descendants of a mixed namespace document construct σ do not necessarily belong to the same language as σ. Consequently, the presentation dependencies between σ and its context/contents are not well defined. For instance, consider the `svg:rect` element, which is illustrated in line 7 of Figure 5.1. The SVG recommendation specifies that it must presented as a rectangle. However, neither the XHTML nor the SVG specifications define how to orchestrate its presentation with the presentation of its ancestor `xhtml:body` element.

Integration profiles and predefined integration constructs are inadequate methods to provide the missing processing associations between languages. Specifically, integration profiles, such as the XHTML+SVG+MathML profile, are not adequate for an open set of languages, as described in Section 4.4. Predefined integration constructs are XML constructs with well defined processing associations between their context and their contents. They can enable the processing of mixed namespace documents, but such fixed sets of constructs are against the principle of independent invention. Moreover, they do not comply with the XML authoring model, which is not compatible with such well defined associations between its constructs, as described in Section 2.2.2.

NRL and NVDL are the most prominent integration approaches for validation, but they do not define the necessary processing relationships for presentation documents. Specifically, they attempt to provide generic integration, which also covers non-presentation documents. Therefore, they cannot utilise the constraints of the presentation domain, in order to define the presentation associations between language constructs. Consequently, they can result to erroneous validation of presentation documents, as illustrated in Listing 2.1 (page 19).

Therefore, a generic integration model for presentation documents can adopt the subtree separation concept of NRL and NVDL, but it must also define the presentation relationships between language constructs. Such relationships cannot be based on

```
1  <n1:doc>
2   <n1:title>
3     Document title
4   </n1:title>
5   <n1:section>
6    <n1:title>
7      Sec1
8    </n1:title>
9    <n1:p>Par1</n1:p>
10   <n1:p>Par2</n1:p>
11   <n4:box>
12    <n1:p>Par3</n1:p>
13    <n1:p>Par4</n1:p>
14   </n4:box>
15  </n1:section>
16 </n1:doc>
```

(a)

```
1  <xhtml:html>
2   <xhtml:head><xhtml:title>
3         Document title
4   </xhtml:title></xhtml:head>
5   <xhtml:body>
6    <xhtml:h1>Document title</xhtml:h1>
7    <xhtml:h2>1. Sec1</xhtml:h2>
8    <xhtml:p>Par1</xhtml:p>
9    <xhtml:p>Par2</xhtml:p>
10   <xhtml:table
11     border="1"><xhtml:tr><xhtml:td>
12    <xhtml:p>Par3</xhtml:p>
13    <xhtml:p>Par4</xhtml:p>
14   </xhtml:td></xhtml:tr></xhtml:table>
15  </xhtml:body>
16 </xhtml:html>
```

(b)

Listing 5.1: Handled constructs example

predefined integration profiles or integration constructs. In contrast, the integration model must utilise the constraints of the presentation domain, in order to define generic relationships that cover all presentation languages.

5.2 Handled construct observations

The XMLPipe integration model is based on three observations, which apply to a significant subset of the presentation languages. This section will describe these observations and provide the foundation for the subsequent XMLPipe integration model proposal.

5.2.1 Handled constructs

The first observation is that each presentation language contains a subset of constructs, where their processing can be *defined* independently of their context, within a document. The term *handled constructs* will refer to such constructs. For instance, consider Listing 5.1(a), which represents a mixed namespace document that contains a title, a section, two top level paragraphs and two additional paragraphs, which are enclosed in a box. Listing 5.1(b) represents an adequate interpretation of 5.1(a) for an XHTML browser. The placement of the source `n1:p` constructs (Listing 5.1(a), lines 9, 10, 12, 13) depends on their ancestors. However, their processing is context independent. Specifically, the transformation of both the first two `n1:p` elements (lines 9, 10), which are enclosed in an `n1:section` element, and the last two `n1:p` elements (lines 12, 13), which are enclosed in an `n4:box` element, always results in an `xhtml:p` construct (Listing 5.1(b) lines 8, 9, 11, 12). Consequently, `n1:p` is a handled construct.

Not all presentation language constructs are handled constructs. For instance, the processing of the two `n1:title` elements differs according to their context. The first occurrence (line 2) defines the document title, but the second occurrence (line 6) defines

a section title. Consequently, their XHTML interpretations (Figure 5.1(b), lines 2–6 and 7) are substantially different. Additionally, the processing of an `n1:title` element that occurs within a `n4:box` is not well defined, because its association with its context is undefined. Consequently, `n1:title` is not a handled construct, because its processing definition depends on its context.

Language authors must explicitly specify the handled constructs of a language, because they are central to the XMLPipe integration model. Functions $langConstructs^{HC}$, $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$ will map a language to its set of handled constructs, and they are a subset of the binding model interface, which will be described in Chapter 9. Specifically, $langConstructs^{HC}$ returns all the handled constructs of a language. $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$ return only the element and attribute handled constructs, respectively. They are necessary, because the XMLPipe integration model treats elements and attributes differently.

Handled constructs function ($langConstructs^{HC}$):
$langConstructs^{HC} : \mathcal{L} \rightarrow \wp(\Sigma)$ is a function where,
$\forall L \in \mathcal{L}$, $\sigma \in langConstructs^{HC}(L)$ iff $\sigma \in langConstructs(L)$ and σ is a handled construct.

Element handled constructs function ($langConstructs_e^{HC}$):
$langConstructs_e^{HC} : \mathcal{L} \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}$,
$langConstructs_e^{HC}(L) = langConstructs^{HC}(L) - langConstructs_a(L)$.

Attribute handled constructs function ($langConstructs_a^{HC}$):
$langConstructs_a^{HC} : \mathcal{L} \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}$,
$langConstructs_a^{HC}(L) = langConstructs^{HC}(L) - langConstructs_e(L)$.

5.2.2 Handled construct rooted subtrees

A consequence of the existence of handled constructs is that the processing of a single namespace subtree that is rooted at a handled construct is also independent of its context.

Corollary 1 *Consider a valid $d \in D_Q$, where $\mathcal{L}_d = \{L\}$, and that the syntax of L enforces the processing relationships between its constructs. If d' is a subtree of d that is rooted at a handled construct $\sigma \in langConstructs^{HC}(L)$, the processing of d' can be defined independently of its context in d.*

Proof:
In order to prove the above corollary, it is sufficient to prove that the processing of all constructs of d' can be defined independently of the context of d'. Consider that Σ_L and Σ'_L represent the handled and non-handled constructs of L, respectively:

$$\Sigma_L = langConstructs^{HC}(L)$$
$$\Sigma'_L = langConstructs(L) - \Sigma_L$$

Each node of d' can correspond to either a handled construct or a non-handled construct. By definition, the processing of all nodes that correspond to a handled

construct $\sigma \in \Sigma_L$ can be defined independently of their context. In contrast, the processing definition of nodes that correspond to a non handled construct $\sigma \in \Sigma'_L$ may depend on their ancestor nodes.

Consider that the processing definition of a node n, which corresponds to construct σ, depends on an ancestor n', which corresponds to a construct σ'. Since the syntax of L enforces the processing relationships between its constructs, it must require that σ can only occur as a descendant of σ'. If n' is an ancestor of the subtree d', the syntax of L must also constrain the root of d' to be a descendant of σ', when it contains σ at the place that corresponds to n. Consequently, if the processing definition of a node in d' depends on a ancestor node of d', the syntax of the root of d' depends on its context. The d root is a handled construct, and, by definition, its processing cannot depend on its context. Therefore, the processing definition of every non handled construct of d' does not depend on the context of d'.

Consequently, the above proves Corollary 1, because the processing of all d' constructs and of d' as a whole can be defined independently of its context.

Most handled constructs are elements, because attributes are closely related to the context of their parent elements. The processing of an attribute must *relate* to its context, because it cannot exist without a well defined parent element. Nevertheless, its processing might be *defined* independently of its context. Therefore, all above observations apply interchangeably to both elements and attributes.

For instance, consider the document illustrated in Listing 5.2(a). According to the semantics of the XLink language, *any* element that contains an `xlink:href` attribute (lines 9 and 12) must be interpreted as a link to the specified attribute URL. Additionally, consider a preprocessing application that adapts Listing 5.2(a) for an XHTML browser that does not support XLink links. It must enclose all elements that contain `xlink:href` attributes into an XHTML anchor element `xhtml:a`, as illustrated in Listing 5.2(b) (lines 8 and 12). The adaptation of the `xlink:href` attributes involves their parent element, because it must be enclosed within the `xhtml:a` element. However, their processing can be defined independently of their context, because it is interchangeable for both `n1:p` and `n3:box` elements. Consequently, the `xlink:href` attribute can be considered as a handled construct.

5.2.3 Handled constructs classification

The final observation is that the valid occurrence of handled construct rooted subtrees depends on the type of their root construct. For a significant subset of the presentation languages, their handled constructs can be classified into three types: *content oriented constructs* (*COC*), *functionality oriented constructs* (*FOC*) and *structure modification constructs* (*SMC*).

Most presentation language handled constructs are *content oriented constructs*, and they introduce presentable content. Specifically, the semantics of a *COC* describe the introduction of a well defined and presentable piece of information, at a place that corresponds to its position in a document. For instance, consider the `n1:doc`, `n1:p` and `n4:box` elements, illustrated in Listing 5.2(a). If their corresponding semantics are that they introduce a document, a paragraph and a box, at the place that they occur in a document, they are all *COC* handled constructs. Additional *COC* examples, within widely used languages, are the `html`, `p` and `em` XHTML elements and the `svg` and `rect` SVG elements.

```
1  <n1:doc>
2   <n1:title>
3    Document title
4   </n1:title>
5   <n1:section>
6    <n1:title>
7     Sec1
8    </n1:title>
9    <n1:p xlink:href="uri1">Par1
10   </n1:p>
11   <n1:p>Par2</n1:p>
12   <n4:box xlink:href="uri2">
13
14
15    <n3:imp ref="pars.xml"/>
16
17
18   </n4:box>
19  </n1:section>
20 </n1:doc>
```

(a)

```
1  <xhtml:html>
2   <xhtml:head><xhtml:title>
3         Document title
4   </xhtml:title></xhtml:head>
5   <xhtml:body>
6    <xhtml:h1>Document title</xhtml:h1>
7    <xhtml:h2>1. Sec1</xhtml:h2>
8    <xhtml:a href="uri1">
9      <xhtml:p>Par1</xhtml:p>
10   </xhtml:a>
11   <xhtml:p>Par2</xhtml:p>
12   <xhtml:a href="uri2">
13    <xhtml:table
14       border="1"><xhtml:tr><xhtml:td>
15     <xhtml:p>Par3</xhtml:p>
16     <xhtml:p>Par4</xhtml:p>
17    </xhtml:td></xhtml:tr></xhtml:table>
18   </xhtml:a>
19  </xhtml:body>
20 </xhtml:html>
```

(b)

Listing 5.2: Handled constructs classification example

The *functionality oriented constructs* do not introduce a separate piece of presentable information, but they amend the presentation of their context, in a context independent way. As described in the previous section, an element that contains an `xlink:href` attribute (Listing 5.2(a), lines 9 and 12) must be interpreted as a link to the specified attribute URL. Consequently, the `xlink:href` attribute amends the presentation of its parent element, in a context independent way, because it converts it to a link, independently of which element its parent is. Therefore, the `xlink:href` attribute is a *FOC*.

The classification of a construct as a *COC* or as a *FOC* depends on both its semantics and its syntax. For instance, the XHTML anchor element `xhtml:a` introduces the same presentation functionality with the `xlink:href` attribute. The former converts its content to a link, and the latter converts its parent element to a link. However, `xhtml:a` is a *COC*, because it introduces a well defined piece of information, since it contains the linked content.

The final commonly observed category of handled constructs are the *structure modification constructs*. *SMC* are constructs that the primary purpose of their associated semantics is to modify the document tree. For instance, the semantics of the `n3:imp` element, illustrated in Listing 5.2(a) (line 15), is to import the referenced document. Consequently, it is an *SMC*, because its semantics describe a modification of the document tree. Most *SMC* constructs can also be classified as *COC* or *FOC*, according to their exact semantics. However, the *SMC* category is necessary, because document validation must process such constructs separately, since the introduced modifications can alter the validity of a document.

The classification of a language's handled constructs is essential for the processing

of presentation documents. The symbols *COC*, *FOC* and *SMC* will represent the functions that map a language to its corresponding subsets of handled constructs. *COC*, *FOC* and *SMC* are a part of the binding model interface.

Handled construct classification functions (*COC*, *FOC*, *SMC*):
The functions
$COC : \mathcal{L} \rightarrow \wp(\Sigma)$, $FOC : \mathcal{L} \rightarrow \wp(\Sigma)$ and $SMC : \mathcal{L} \rightarrow \wp(\Sigma)$ are defined as follows:
$\forall \sigma \in langConstructs^{HC}(L)$:
$\sigma \in COC(L)$, if σ is a *COC*
$\sigma \in FOC(L)$, if σ is a *FOC*, and
$\sigma \in SMC(L)$, if σ is an *SMC*.

5.2.4 Valid nesting of subtrees

For the majority of presentation languages, the valid nesting of handled construct rooted subtrees depends on the classification of their root handled constructs.

Specifically, *SMC* rooted subtrees can usually occur at any place in a document, because tree modification functionality can apply to any context. For instance, consider any document d. Any subtree d' of d can be retrieved by an external source. Consequently, d' can be substituted by an *SMC* construct that imports d', such as the `n3:imp` construct in 5.2(a). Therefore, such an *SMC* construct can validly occur at any place in a document.

FOC rooted subtrees can also occur at any place, because presentation amendments are always applicable in a presentation document. Specifically, since a *FOC* amends the presentation of its context, it should occur as a descendant of a *COC*, which are the only constructs that introduce well defined pieces of presentable information. However, a *FOC* might also validly occur under another *FOC*, so that they both amend the presentation of their context. Additionally, a *FOC* can also occur as a descendant of an *SMC*, because, after the *SMC* processing, the new context of the *FOC* construct will either be a *COC* or another *FOC*. Consequently, *FOC* rooted subtrees can validly occur at any place in a document. For instance, the `xlink:href` attribute, illustrated in Listing 5.2(a), is a *FOC*. `xlink:href` can be added to any element of Listing 5.2(a), because all information can be presented as a link.

The nesting of *COC* rooted subtrees is more restrictive. Specifically, only constructs that allow the introduction of presentable content can be parents of *COC* constructs, because they introduce a presentable piece of information. For instance, consider that the semantics specification of the `n1:doc` construct, illustrated in Listing 5.2(a), states that is must contain exactly one title and a sequence of sections. `n1:p` and `n4:p` are *COC* constructs, and they can occur as children of the `n1:section` element, because a document section can be the host of arbitrary content. In contrast, they cannot validly occur under the `n1:doc` element, because they will not belong to a document section, and they will contradict the `n1:doc` semantics: it must only contain a title and a sequence of sections. In a similar manner, an XHTML paragraph cannot be validly placed between lines 4 and 5 of Listing 5.2(b), because it would introduce presentable content outside the body of the document.

5.3 Handled constructs based integration

The aforementioned observations enable the partitioning a single namespace presentation document into a collection of autonomous information entities with well defined associations. The autonomous information entities are the handled construct rooted subtrees, because their processing can be defined independently of their context. The associations between the handled construct rooted subtrees are well defined, because the classification of their root handled constructs defines their relationship and valid integration with their context.

All observations are solely based on the existence of handled constructs and their classification, and they do not depend on language specific features. Consequently, they can be extended to mixed namespace documents. This section derives the XMLPipe integration model by extending the aforementioned observations to mixed namespace presentation documents.

5.3.1 Valid mixed namespace documents

A document tree can be processed autonomously, if it is rooted at an element handled construct that is not a *FOC*. Specifically, consider a single namespace document tree d, which is rooted at a handled construct σ of a language L. According to Corollary 1, the processing of d can be defined independently of its context. If σ is an attribute, d cannot occur outside the context of a parent element. If σ is a *FOC*, the processing of d may require to access its context. In contrast, the processing of most subtrees that are rooted at an element *COC* or *SMC* does not require access to their context. Consequently, d can be processed autonomously, if σ is a non *FOC* element handled construct and d is valid, according to the syntax of L.

If the nesting observations in Section 5.2.4 are followed, the insertion of a handled construct rooted subtree into a document that can be processed autonomously, results in a document that can also be processed autonomously. Specifically, consider that d can be processed autonomously and d' is rooted at a handled construct σ', of a language L'. d'' will denote the result of inserting d' at either a place in d where content is expected, if σ' is a *COC*, or at any place in d, otherwise. d'' can be processed autonomously, because d can be processed autonomously, the processing of d' is well defined, and the processing relationship between d' and its context is also well defined. Specifically, if σ' is a *FOC*, the processing of d' will amend the presentation of its context, within d'', in a well defined way. If σ' is an *SMC*, its processing will modify the document tree in a well defined way, independently of its context in d''. Finally, if σ' is a *COC*, the relationship to its parent is well defined, because it introduces presentable content at a place where presentable content is expected. Consequently, d'' can be processed autonomously.

The XMLPipe integration model uses the above tree composition to ensure that the processing of valid mixed namespace documents is well defined. Specifically, a valid mixed namespace document, according to the XMLPipe integration model, is composed out of a collection of handled construct rooted trees. Its root construct must be a non-*FOC* element handled construct and all *COC* rooted subtrees must only occur at places where content is expected.

The symbol $\mathcal{V}_{\mathcal{L}_1}^{Im_X}$ will denote the set that contains all valid documents, according to the XMLPipe integration model, which combine the constructs of an open set of

presentation languages $\mathcal{L}_1$. The definition of $\mathcal{V}^{Im_X}$ uses the $\underset{Im}{\overset{+}{\leftarrow}}$ operator, which defines the set of valid tree compositions, according to an integration model Im. Im_X will represent the XMLPipe integration model. The definition of $\mathcal{V}^{Im_X}$ formulates the XMLPipe integration model, but it does not indicate how to author and process valid mixed namespace documents. The following two sections will addresses these applications of the proposed integration model.

Valid tree composition ($\underset{Im}{\overset{+}{\leftarrow}}$): For all $d_1, d_2 \in D_Q$, $d_1 \underset{Im}{\overset{+}{\leftarrow}} d_2$ is the set of all documents that can be produced by placing d_2 at a valid place within d_1, according to the integration model Im.

Valid XMLPipe integration model composition ($\underset{Im_X}{\overset{+}{\leftarrow}}$): For all $d, d' \in D_Q$, where d' is rooted at a construct σ',

$$d \underset{Im_X}{\overset{+}{\leftarrow}} d' = \begin{cases} \emptyset & , \text{ if } \sigma' \notin \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} langConstructs^{HC}(L) \\ \{d_1, \ldots, d_n\} & , \begin{array}{l} d_1, \ldots, d_n \text{ result from placing } d' \text{ at any place in } d \\ \text{and } \sigma' \in \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} (FOC(L) \cup SMC(L)) \end{array} \\ \{d_1, \ldots, d_n\} & , \begin{array}{l} d_1, \ldots, d_n \text{ result from placing } d' \text{ at a place in } d \text{ where} \\ \textit{content is expected} \text{ and } \sigma' \in \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} COC(L) \end{array} \end{cases}$$

XMLPipe valid documents ($\mathcal{V}^{Im_X}$): For a set of languages $\mathcal{L}_1$, the set of the XMLPipe integration model valid documents $\mathcal{V}^{Im_X}_{\mathcal{L}_1}$ is defined as follows:

$$d \in \mathcal{V}^{Im_X}_{\mathcal{L}_1} \text{ iff } \begin{cases} \mathcal{L}_d = \{L\},\ d \text{ is rooted at } \sigma \in (COC(L) \cup SMC(L)) \cap langConstructs^{HC}_e(L) \text{ and } d \text{ is a valid tree of } L \\ OR \\ \exists d_1, d_2, \text{ where } d_1 \in \mathcal{V}^{Im_X}_{\mathcal{L}_1},\ \mathcal{L}_{d_2} = \{L_2\},\ d_2 \text{ is rooted at } \sigma_2 \in langConstructs^{HC}(L) \text{ and } d_2 \text{ is a valid tree of } L_2, \text{ so that } d \in d_1 \underset{Im_X}{\overset{+}{\leftarrow}} d_2 \end{cases}$$

5.3.2 Mixed namespace document authoring

According to the XMLPipe integration model, a document author must be aware of the $\mathcal{V}^{Im_X}$ definition, the syntax of an open set of languages, their handled constructs and the places where content is expected. This information allows the combination of a set of independently developed presentation languages into a valid mixed namespace document. Specifically, the syntax and the handled constructs of a language are the necessary information for creating valid single namespace trees that are rooted handled constructs. The classification of the handled constructs and the identification of places

where content is expected enable the valid integration of the individual subtrees into a mixed namespace document. Consequently, document authors can use the XMLPipe integration model to create valid presentation documents, which use an open set of languages, without relying on predefined integration profiles.

For instance, consider the example document illustrated in Listing 5.2(a). The symbols L_1, L_3, L_4 and L_{xlink} will denote the languages that correspond to the namespace prefixes `n1`, `n3`, `n4` and `xlink`. Additionally, consider that all four languages have a well defined set of handled constructs:

$$COC(L_1) = \{\texttt{doc}, \texttt{p}\}, FOC(L_1) = SMC(L_1) = \emptyset$$
$$SMC(L_3) = \{\texttt{imp}\}, COC(L_3) = FOC(L_3) = \emptyset$$
$$COC(L_4) = \{\texttt{box}\}, FOC(L_4) = SMC(L_4) = \emptyset$$
$$FOC(L_{xlink}) = \{\texttt{href}\}, COC(L_{xlink}) = SMC(L_{xlink}) = \emptyset$$

The `n1:doc` and `n1:p` constructs of L_1 are COC, because they introduce a document and a paragraph, respectively. Both are well defined pieces of presentable information and their presentation is independent of their context. In a similar manner, the `n4:box` construct of L_4 is also a COC. The `xlink:href` construct of L_{xlink} is a FOC, because its processing influences its parent element in an element independent way (it converts it to a link). The `n3:imp` construct of L_3 is an SMC, because it modifies the document tree by importing external content at its place. Finally, the `n1:section`, `n1:p` and `n4:box` elements denote the places where content is expected.

A document author can create 5.2(a) by combining its individual subtrees, according to the $\mathcal{V}^{Im_X}$ definition. The document authoring process can begin by creating the `n1:doc` handled construct, the title and the first section, according to the L_1 syntax. Subsequently, the `n4:box` element can be introduced as a child of the `n1:section` element, because `n4:box` is a COC and `n1:section` denotes a place where content is expected. Finally, the document author can validly introduce the `xlink:href` and `n3:imp` constructs at any document position, because they are FOC and SMC, respectively.

5.3.3 Mixed namespace document processing

A document author can obtain the necessary information to author a document by the corresponding language specifications. However, an integration model specific process, such as validation or transformation, must be able to access machine processible representations of all necessary information: the handled construct information and the places where presentable content is expected.

As described in Section 5.2, the functions COC, FOC, SMC, $langConstructs^{HC}$, $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$ are a part of the XMLPipe binding model interface, and they map a language to its handled constructs information. The symbol *ISemantics* will represent the set of all *XMLPipe integration model semantics*, which contains the necessary information for implementing the above functions. Specifically, an *ISemantics* member contains 5-tuples of qualified construct name sets, which correspond to the COC, SMC, FOC, element and attribute handled constructs. The XMLPipe binding model, which is described in Chapter 9, will define how the integration model semantics are retrieved for each language.

None of the aforementioned functions map a language to its corresponding information of places where content is expected. Such information is very closely related to the

XMLPipe integration model semantics (*ISemantics*): *ISemantics* represents the set of all XMLPipe integration model semantics. Each member of *ISemantics* is a 5-tuple of sets of qualified names that correspond to the *COC*, *SMC*, *FOC*, element and attribute handled constructs of a language, respectively:

$$ISemantics = \wp(\Sigma)^5$$

If $is \in ISemantics$ and $is = (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5)$, then

$$\forall \sigma \in \Sigma i : \begin{cases} \sigma \text{ is a } COC \text{ construct} & , \text{if } i = 1 \\ \sigma \text{ is a } SMC \text{ construct} & , \text{if } i = 2 \\ \sigma \text{ is a } FOC \text{ construct} & , \text{if } i = 3 \\ \sigma \text{ is an element construct} & , \text{if } i = 4 \\ \sigma \text{ is an attribute construct} & , \text{if } i = 5 \end{cases}$$

syntax of a language, and it requires more complex structures than a simple enumeration. We will propose two alternative methods to identify the places where presentable content is expected: a heuristic method and an explicit identification method.

Consider a construct σ of a language L. The heuristic method is to assume that σ denotes a place where content is expected, if the syntax of L allows the occurrence of a $\sigma' \in COC(L)$ as a child of σ. The rationale behind this method is that if a construct can be the parent of *COC* constructs of the same language, it should also be a valid parent of *COC* constructs of other languages. Nevertheless, a heuristic approach can produce incorrect results. For instance L might require a specific $\sigma' \in COC(L)$ as a child of σ, instead of any arbitrary *COC* construct. Additionally, if a language does not contain *COC* constructs, the proposed heuristic method cannot infer the places where content is expected.

The explicit identification method requires that the schema of a language includes a predefined construct at all places where arbitrary *COC* constructs can occur. The introduction of such a predefined construct is not against the principle of independent invention, because it must only occur within schemas that are specific to the proposed integration model. Such an approach is more precise than the heuristic method, but it requires that all document languages are associated with integration model specific schemas. A processing model can combine both approaches by using the explicit identification method, when such schemas are available, and the heuristic approach, as a fallback mechanism.

An integration model specific component can use the above information to partition a valid mixed namespace document into a collection of handled construct rooted subtrees, which enable its processing. Specifically, consider a $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ that is rooted at a construct σ of a language L. d consists of a top level single namespace tree d' and a collection of subtrees $d_1 \ldots d_n$, which are rooted at handled constructs $\sigma_1 \ldots \sigma_n$ that belong to other languages: $\forall i, \sigma_i \in \mathcal{L}_d - L$. According to the XMLPipe integration model, a valid mixed namespace document is composed out of valid single namespace trees, which are inserted at valid places within their context. Consequently, d' is a valid single namespace document, which is rooted at a handled construct, and $d_1 \ldots d_n$ occur at valid places, according to the classification of their root constructs $\sigma_1 \ldots \sigma_n$.

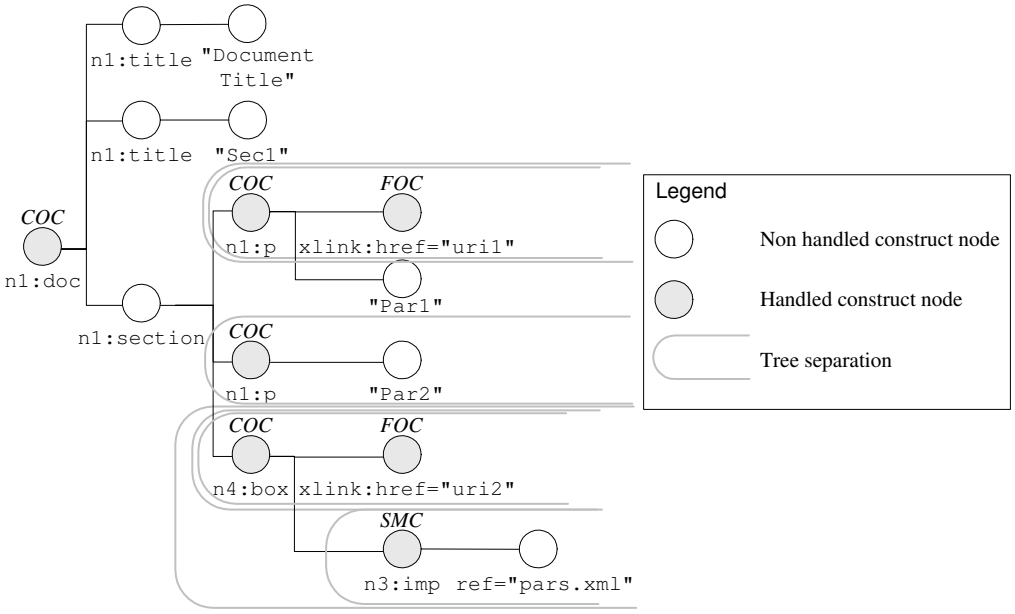

Figure 5.2: XMLPipe integration model subtree separation example

An integration model specific component can process d by combining the independently developed semantics of each document language, if they are adequate for subtree processing. Specifically, if the processing semantics specification for L defines the processing of handled construct rooted trees, it enables the independent processing of d'. The processing of d is well defined, if $d_1 \dots d_n$ also have well defined processing: all of d' and $d_1 \dots d_n$ will have well defined processing, and their processing relationships will be well defined by the classification of the $\sigma_1 \dots \sigma_n$ handled constructs. In a similar manner to d, $d_1 \dots d_n$ are also valid XMLPipe documents. Therefore, a recursive application of the same subtree partitioning as above can show that their processing is well defined. Consequently, d can be processed according to the processing semantics of its individual languages.

Each integration model specific process can use different methods to separate a mixed namespace document into its individual subtrees and to combine the processing semantics of each language. Such methods are closely related to the nature of each process. Therefore, this section will not describe the details of the validation and transformation of mixed namespace documents. The subsequent Chapters 7 and 8 will propose the XMLPipe validation and transformation models, respectively, and they will provide the corresponding document processing details. Nevertheless, the high level concepts of the document processing are common. Specifically, an integration model specific process must separate a mixed namespace document into its individual subtrees, process them independently and combine the processing results.

The subtree separation consists of traversing a mixed namespace document and creating a separate subtree, for each handled construct σ. If σ is a non-FOC element, it can be the root of the corresponding subtree. Otherwise, the subtree root must be the parent of σ, because attributes can only exist within their parent element and the

processing of *FOC* rooted subtrees may require access to its context. For instance, Figure 5.2 illustrates the resulting subtrees for the XML document in Listing 5.2(a). The `xlink:href` attribute is a *FOC* construct. Consequently, its corresponding subtrees also include its parent elements `n1:p` and `n4:box`. All other handled constructs, such as the `n3:imp`, `n4:box` and `n1:p`, are non-*FOC* element handled constructs. Consequently, they become the root constructs of their corresponding subtrees.

After the subtree separation, an integration model specific process processes each subtree separately, according to the semantics of the corresponding language. Finally, it combines the results of each subtree processing into an output document.

5.4 Discussion

The proposed integration model enables the generic processing of mixed namespace documents by utilising the constraints of the presentation domain. Specifically, it is based on three handled construct observations that are specific to presentation documents. The observations have been made to single namespace documents, and their extrapolation allowed the definition of valid mixed namespace documents, as a composition of valid single namespace subtrees. The proposed definition of valid documents enables the generic authoring and processing of mixed namespace documents, because it only requires the independently specified semantics of each language, instead of fixed integration profiles.

The preprocessing framework specifies several requirements that relate to the integration model, but this section will only address a subset of them. Most relevant requirements are specific to the integration model specific validation and transformation processes. Consequently, these requirements will be discussed in the subsequent chapters, after describing the XMLPipe validation and transformation models. This section will focus on evaluating the top level integration model concepts, which are independent of the individual processes.

Firstly, the XMLPipe integration model allows the integration of an open set of languages, which can contain both high and low level presentation information, because it imposes minimal restrictions on the processed languages. Specifically, it only requires that each language has a set of handled constructs. All presentation languages have at least a handled construct, because the processing definition of their designated root construct must always be independent of its context. Consequently, the XMLPipe integration allows the integration of all presentation XML languages.

Moreover, the preprocessing framework requires a well defined way to author and process mixed namespace documents, which does not require predefined relationships between the document languages. The introduced concept of handled constructs enables the authoring and processing of mixed namespace documents by individually addressing their single namespace subtrees, because it enables the separation of a document into autonomous information entities. No predefined relationships are necessary, because each subtree is processed independently. The handled constructs classification establishes the construct relationships that are not covered by the individual language specifications. These relationships solely depend on the classification of the handled constructs, and not on predefined associations. Therefore, the use of handled constructs enables the processing and authoring of documents that combine a set of independently developed languages, because it eliminates the requirement for predefined relationships.

The classification of the handled constructs into a *predefined* set of three categories should not be considered as a restriction of the XMLPipe integration model. In contrast it represents a balanced trade-off between integration generality and power. More fine grained construct relationships could enable more precise integration, but they would cover a narrower integration spectrum. In contrast, more abstract or no classification, such as in NRL, could allow a wider integration spectrum. However, it would not provide the necessary foundation for the well defined processing of presentation documents. The proposed three categories, do not significantly constrain the used languages, but they establish the necessary construct relationships for validating and transforming presentation documents, as the subsequent chapters will illustrate.

Finally, the XMLPipe integration model is significantly more powerful than existing approaches. It adopts the two most prominent generic integration approaches and enhances them by incorporating the presentation domain specific observations of handled constructs. Specifically, the proposed document processing and authoring address the individual document subtrees separately, in a similar manner to NRL and NVDL. Moreover, the classification of the handled constructs provides well defined associations between the individual constructs, in a similar manner to the explicit associations of RDF. The XMLPipe integration model is the only approach that allows authoring and processing of mixed namespace documents, in a way that is adequate for the Web, because it is the only integration approach that fulfils the preprocessing framework requirements.

5.5 Summary

This chapter introduced the XMLPipe integration model, which establishes the foundation of XMLPipe. Specifically, it utilises the common characteristics of the majority of presentation languages, in order to extend the NRL and NVDL concept of subtree separation. The resulting integration enables the processing and authoring of mixed namespace documents, according to the independently developed semantics of their individual languages. The use of independently developed semantics is essential for a generic preprocessing model, such as XMLPipe, that processes mixed namespace documents that combine an open set of languages.

An integration model is necessary for processing mixed namespace documents, because it defines the necessary construct relationships, which are not covered by the language specifications. Specifically, presentation language specifications define the presentation processing of their individual constructs and of the construct relationships, which is necessary for presenting a document. However, the processing relationships in mixed namespace documents are not always well defined. Integration profiles can define them, but they are not adequate for an open set of languages. NRL and NVDL are prominent integration approaches that are adequate for an open set of languages, but they provide generic integration and do not specify the necessary relationships for presentation document processing.

The XMLPipe integration model introduces the necessary associations, by extrapolating three observations of the structure of single namespace presentation documents to mixed namespace documents. Firstly, each presentation language has a set of handled constructs. The processing of all handled constructs can be defined independently of their context. Secondly, the processing of a handled construct rooted subtrees can also be defined independently of their context. Thirdly, the handled constructs can

be classified into three categories. The classification of the root construct of a subtree specifies the presentation relationship between the subtree and its context. The proposed integration model extends these observations to define the set of valid mixed namespace documents.

According to the definition of valid mixed namespace documents, document authors can create mixed namespace documents, knowing only the syntax of the individual languages and the corresponding handled constructs information. The integration model specific processing of mixed namespace documents depends on the nature of the processing, and it differs between document validation and transformation. However, the high level process remains the same, and it consists of separating the document into its individual subtrees, processing them, according to their corresponding language semantics, and combining the processing results.

The XMLPipe integration model fulfils the relevant preprocessing framework integration requirements. Consequently, it sets an adequate foundation for the validation and transformation of presentation documents, in a way that is adequate for the Web. However, the individual proposals of integration model specific processes will be postponed until Chapter 7. The following chapter will describe the XMLPipe adaptation model, because it sets the necessary foundation for proposing the XMLPipe transformation model.

Chapter 6

XMLPipe adaptation model

An adaptation model defines the representation and processing of the adaptation requirements. A well defined adaptation model is essential for investigating the processing of presentation documents, because all transformation processing components and their corresponding transformation semantics depend on the adaptation requirements, as described in Section 4.4.

This chapter describes the XMLPipe *adaptation model*. The adaptation model discussion begins with the definition of the XMLPipe *adaptation profiles*, which define a representation of adaptation requirements, in a similar manner to the CC/PP profiles. Adaptation profile composition is necessary, because it enables the efficient transmission of the adaptation requirements, from separate adaptation requirement sources (e.g. the browser, device and document user), as described in Section 4.3. Consequently, the XMLPipe adaptation model provides a well defined method to compose independently specified adaptation profiles. It also defines a method to choose the optimal transformation specification for a document subtree, over a set of alternatives. This method provides the core adaptation model functionality, because it allows the well defined adaptive transformation of document subtrees. The composition of adaptation profiles and the optimal transformation selection require a well defined mechanism for querying the adaptation requirements. The XMLPipe adaptation model introduces the concept of the *adaptation expressions*, which are a declarative method to express the necessary adaptation profile queries.

The next section describes the main advantages and disadvantages of CC/PP, which is the most prominent existing representation of adaptation requirements. Subsequently, Section 6.2 will define the XMLPipe adaptation profiles and expressions, which adopt the core CC/PP concepts. Finally, Sections 6.3 and 6.4 will describe the profile composition process and optimal transformation selection processs, respectively.

6.1 Adaptation considerations

Section 2.4.4 introduced the Composite Capabilities/Preference Profiles (CC/PP) recommendation, which is the most prominent existing method for representing adaptation requirements. The CC/PP representation is a two level hierarchy of RDF-based attribute and value pairs. Each CC/PP attribute is uniquely identified by a corresponding URI. The CC/PP recommendation does not include a typing mechanism, but the CC/PP attribute data types can be defined by RDF Schemas, which, in turn, use the XML Schema data types.

CC/PP provides an extensible representation of adaptation requirements. Specifically, the URI qualified attributes allow an extensible representation, because they ensure unique and context independent identification of all CC/PP attributes. These two properties are essential for the unambiguous introduction of new attributes, because they eliminate the conflicts between independently defined CC/PP attributes.

Composite representations are beneficial and adaptation requirement queries are necessary, as described in this chapter's introduction. However, CC/PP complicates the support of the necessary adaptation functionality. CC/PP allows profile composition, but it does not provide a well defined mechanism to resolve conflicting values. Such a mechanism is essential for the composition of independently defined profiles. Moreover, any RDF query language can be used for expressing queries to the RDF-based CC/PP representation. An adaptation model that uses CC/PP queries must incorporate support for a multitude of relevant technologies: RDF, RDF Schema, XML Schema data types and one or more RDF query languages. The support of these technologies is not prohibitive, but it could be considered as unnecessarily complex for an adaptation model.

6.2 Adaptation profiles and expressions

This section will define the XMLPipe representation of adaptation requirements and adaptation requirement queries. The introduction of their individual concepts will follow a rather non-intuitive sequence, in order to avoid forward references, within their definitions.

The XMLPipe adaptation model uses a simpler representation and data typing method than CC/PP, in order to avoid the inherent complexity of supporting all the relevant RDF technologies, which are necessary for enabling CC/PP profile queries. However, it will adopt the CC/PP concept of URI identified adaptation attributes, because they allow an extensible representation. The remainder of this thesis will use the term *adaptation terms*, as opposed to the CC/PP term *attributes*, in order to disambiguate between references to *XML attributes* and *adaptation attributes*.

Specifically, an XMLPipe adaptation term is a combination of a URI and a local name. *Terms* will represent the set of all adaptation terms. A pair of an adaptation term and a data value expresses an adaptation requirement, as a user/browser capability or a user preference. For instance, consider that the `uri1:supported` and `uri2:noImages` adaptation terms, where the former corresponds to the set of technologies that a browser supports and the latter corresponds to the preferred use of images within a presentation. The pair

$$(\texttt{uri1:supported}, \{\texttt{http://www.w3.org/1999/xhtml}\})$$

expresses the capability of a browser to render XHTML documents. The pair

$$(\texttt{uri2:noImages}, \texttt{true})$$

expresses the preference for a presentation that does not contain images.

Adaptation terms set (*Terms*): The set of all adaptation terms *Terms* is the set of all pairs of a URI and an XML local name:

$$Terms = URI \times S$$

where *URI* is the set of all URIs and S is the set of all XML local names.

Typed adaptation terms, where each term is associated to its corresponding set of acceptable values, are beneficial, because they enable the validation of term–value pairs and assist the evaluation of adaptation requirement queries. The XMLPipe adaptation model associates each adaptation term with a member of a predefined collection of data types. Each data type specifies a term's acceptable set of values, a set of valid operations (such as multiplication of numerals) and a set of conversion functions (such as converting a numeral to a string). *TermTypes* will represent the set of all XMLPipe *adaptation term types*. Each type $Type \in TermTypes$ will represent a set that contains all valid data values. Moreover, the *Numeric* and *Boolean* symbols will correspond to the *numeric* and *boolean* data types. The separate definitions of *Numeric* and *Boolean* are necessary, because they will be used in the subsequent proposal of the optimal transformation selection process.

Adaption term types (*TermTypes*): The set of all *adaptation term types* *TermTypes* contains all adaptation term data types

Numeric data type (*Numeric*): The *numeric type* $Numeric \in TermTypes$ is the XMLPipe numeric data type, and its acceptable values are real numbers:
if $v \in Numeric$ then $v \in \mathbb{R}$

Boolean data type (*Boolean*): The *boolean type* $Boolean \in TermTypes$ is the XMLPipe boolean data type, and it contains the values `true` and `false`:
$Boolean = \{\texttt{true}, \texttt{false}\}$

Adaptation requirement queries are necessary for the profile composition and optimal transformation selection, as Sections 6.3 and 6.4 will illustrate. The term *adaptation expression* will refer to the XMLPipe declarative mechanism for querying the values of adaptation terms. Specifically, an adaptation expression is a function that maps a tuple of values to a result value. The set $\mathcal{F}^{Type'}_{\{Type_1,\dots Type_n\}}$ will represent the set of all adaptation expressions that map an n-tuple of values, which belong to an n-tuple of term types $Type_1, \dots Type_n$, to a value of type $Type'$. The convenience set $\mathcal{F}^{Type_2}_{(Type_1)^n}$ will represent the subset of expressions that map n values of a type $Type_1$ to a value of type $Type_2$, and it will be extensively used in the following sections. Finally, the set $\mathcal{F}$ will represent all XMLPipe adaptation expressions.

For instance, if $f(x_1, x_2) = x_1 + x_2$ and $f'(x_1) = (x_1 == 2)$, then $f \in \mathcal{F}^{Numeric}_{(Numeric)^2}$ and $f' \in \mathcal{F}^{Boolean}_{(Numeric)^1}$. Both both f and f' belong to $\mathcal{F}$.

Adaptation expressions from a value tuple to a value ($\mathcal{F}^{Type'}_{\{Type_1,\dots Type_n\}}$): The set $\mathcal{F}^{Type'}_{\{Type_1,\dots Type_n\}}$ contains all XMLPipe expressions of the form
$f : Type_1 \times Type_2 \times \cdots \times Type_n \rightarrow Type'$

Adaptation expressions from a single type value tuple to a value ($\mathcal{F}^{Type_2}_{(Type_1)^n}$): $\mathcal{F}^{Type_2}_{(Type_1)^n} = \mathcal{F}^{Type_2}_{\{\underbrace{Type_1, \dots, Type_1}_{n}\}}$

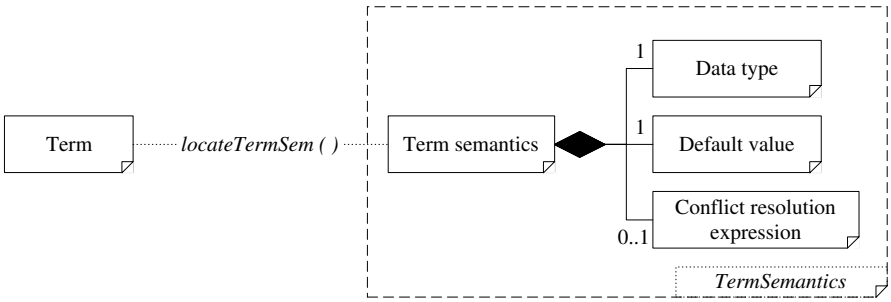

Figure 6.1: Adaptation term semantics

All adaptation expressions ($\mathcal{F}$): The set $\mathcal{F}$ includes all XMLPipe expressions:

$$\mathcal{F} = \bigcup_{\forall n \in \mathbb{N}, Type', \forall Type_1 \ldots Type_n} \mathcal{F}^{Type'}_{\{Type_1, \ldots, Type_n\}}$$

where $\mathbb{N}$ is the set of natural numbers {1, 2, ...}

An extensible representation of adaptation requirements, which does not rely on a fixed set of adaptation terms, necessitates well defined processing semantics, for each adaptation term. The above term type, term value and adaptation expression definitions allow the definition of the adaptation term semantics. Specifically, *TermSemantics* will represent the set of all *adaptation term semantics*. Each member of *TermSemantics* is a triplet of a term data type, a valid default value and conflict resolution expression. As described above, a data type is necessary for the validation of term–value pairs and the evaluation of adaptation requirement queries. Additionally, a default value is beneficial for an extensible representation, because it ensures the well defined evaluation of adaptation expressions, when the value of a referenced term is not specified. Finally, the conflict resolution expression is a binary adaptation expression (which uses two input values) that assists the resolution of conflicting value specifications, as Section 6.3 will illustrate.

Term semantics (*TermSemantics*): The set of all term semantics *TermSemantics* contains all valid triplets of a term type, a default value and a conflict resolution expression:

$$TermSemantics = \bigcup_{\forall Type \in TermTypes} \left(\{Type\} \times Type \times \left(\{\epsilon\} \cup \mathcal{F}^{Type}_{(Type)^2} \right) \right)$$

A well defined method to locate a term's semantics, is also essential for an extensible representation, because it ensures that a preprocessing model can retrieve all necessary adaptation term information. Figure 6.1 illustrates the relationship between a term and its semantics. Function *locateTermSem* is a part of the binding model interface (described in Chapter 9), and it maps a term to its corresponding semantics. For instance, consider that τ corresponds to the `uri1:supported` term, and that

$locateTermSem(\tau) = ts$, where $ts = (SetOfStrings, \{\}, x_1 \cup x_2)$. The semantics of τ specify that its data type is a set of strings, its default value is an empty set of strings and its default conflict resolution mechanism is to combine the conflicting sets of strings.

The convenience functions $termType()$, $termDefault()$ and $termResolve()$ map an adaptation term to its data type, default value and resolution expression, respectively. These functions are necessary for a more concise method to refer to the individual components of a term's semantics.

Term semantics location function (*locateTermSem*): The term semantics location function $locateTermSem : Terms \rightarrow TermSemantics$ maps an adaptation term to its corresponding semantics:
$\forall \tau \in Terms$, $locateTermSem(\tau)$ returns the semantics associated with τ, according to the XMLPipe binding model.

Term semantics convenience functions (*termType*,*termDefault*,*termResolve*): For every term $\tau \in Terms$, where $locateTermSem(\tau) = (Type, v, f)$, the utility functions are defined as follows

$$\begin{array}{ll} termType : Terms \rightarrow TermTypes, & termType(\tau) = Type \\ termDefault : Terms \rightarrow \bigcup_{\forall Type \in TermTypes} (Type), & termDefault(\tau) = v \\ termResolve : Terms \rightarrow \mathcal{F}, & termResolve(\tau) = f \end{array}$$

The association between a term and its semantics, enables the definition of an adaptation statement, which expresses an adaptation requirement. Specifically, an *adaptation statement* is a pair of an adaptation term and a valid data value, according to its semantics. *Statements* will represent the set of all XMLPipe adaptation statements. Moreover, an *adaptation profile* is a set of adaptation statements, and it can express the set of adaptation requirements that influence the processing of an XML document. *Profiles* will represent the set of all adaptation profiles.

Adaptation statements (*Statements*): The set of all adaptation statements *Statements* includes all pairs of a term and a value of its corresponding type.

$$Statements = \bigcup_{\forall \tau \in Terms} (\{\tau\} \times termType(\tau))$$

Adaptation profiles (*Profiles*): The set of all adaptation profiles *Profiles* includes all sets of adaptation statements: $Profiles = \wp(Statements)$

For example, consider the two example profiles in Listings 6.1 and 6.2, which describe an XHTML browser running on a desktop computer and a WML mobile, respectively. Each profile (`profile`) consists of a set of statements (`statm`) that express the relevant adaptation factor requirements. Each statement specifies the namespace URI and local name of an adaptation term (`ns`, `name`), and it includes its corresponding value. For instance, line 2 of Listing 6.1 specifies that the target device is a "desktop".

```
1 <profile name="desktop1">
2   <statm ns="xmlPipeUri" name="deviceType">desktop</statm>
3   <statm ns="xmlPipeUri" name="supported">
4     <item>http://www.w3.org/1999/xhtml</item>
5     <item>http://www.w3.org/2001/XMLSchema-instance</item>
6     <item>xmlPipeUri/Profile/Mime/image/jpeg</item>
7     <item>xmlPipeUri/Profile/Mime/video/mpeg</item>
8   </statm>
9 </profile>
```

Listing 6.1: Adaptation profile for a desktop device

```
1 <profile name="mobile1">
2   <statm ns="xmlPipeUri" name="deviceType">mobile</statm>
3   <statm ns="xmlPipeUri" name="supported">
4     <item>http://www.wapforum.org/DTD/wml_1.1.xml</item>
5     <item>xmlPipeUri/Profile/Mime/image/wbmp</item>
6   </statm>
7     <statm ns="xmlPipeUri" name="maxImageX">96</statm>
8     <statm ns="xmlPipeUri" name="maxImageY">100</statm>
9 </profile>
```

Listing 6.2: Adaptation profile for a mobile device

In a similar manner, line 2 of Listing 6.2 specifies that the corresponding device is a "mobile". Lines 3–8 of Listing 6.1 specify that the target browser can render XHTML, JPEG and MPEG content. The aforementioned semantics location functions allow the location of all necessary term semantics, without additional external input. For instance it is sufficient to use the namespace and local name of the `maxImageX` and `supported` terms, in order to retrieve their semantics:

$$termType(\texttt{xmlPipeURI}, \texttt{maxImageX}) = Numeric$$
$$termType(\texttt{xmlPipeURI}, \texttt{supported}) = SetOfStrings$$

6.3 Profile composition

The efficient transmission of the user preferences and browser/device capabilities necessitates a well defined mechanism to combine independently developed adaptation profiles, as described in Section 4.3. The concepts of adaptation terms and adaptation expressions, which have been established in the previous section, can form the basis of such a mechanism. This section will describe the *XMLPipe profile composer* that is responsible for combining multiple adaptation profiles and resolving their conflicts.

6.3.1 Profile composition observations

The mobile adaptation profile (Listing 6.2) can be partitioned into the profiles illustrated in Listings 6.3, 6.4 and 6.5. The profile in Listing 6.3 can be provided by the mobile manufacturer, and it represents the static capabilities of the described mobile device: the device type, Wireless Markup Language (WML) support and the maximum rendered image size. The vendor of a Wireless Bitmap (WBMP) rendering upgrade,

```
<profile name="mobileDefault">
  <statm ns="xmlPipeUri" name="deviceType">mobile</statm>
  <statm ns="xmlPipeUri" name="supported">
    <item>http://www.wapforum.org/DTD/wml_1.1.xml</item>
  </statm>
  <statm ns="xmlPipeUri" name="maxImageX">150</statm>
  <statm ns="xmlPipeUri" name="maxImageY">150</statm>
</profile>
```

Listing 6.3: Default mobile profile

```
<profile name="mobileCustom">
  <statm ns="xmlPipeUri" name="supported">
    <item>xmlPipeUri/Profile/Mime/image/wbmp</item>
  </statm>
</profile>
```

Listing 6.4: Software upgrade profile

```
<profile name="mobileUser">
    <statm ns="xmlPipeUri" name="maxImageX">96</statm>
    <statm ns="xmlPipeUri" name="maxImageY">100</statm>
</profile>
```

Listing 6.5: User preferences profile

which adds WBMP rendering capability to a mobile browser, can provide the profile illustrated in Listing 6.4. Finally, a user that wishes to override the default image size capabilities can provide the profile illustrated in Listing 6.5.[1]

The composition of the three adaptation profiles must result to the initial mobile profile (Listing 6.2). Specifically, the composition process must resolve the conflict between the `supported` statements, in the first two listings, and between the `maxImageX` and `maxImageY` statements, in the first and third listings. The WBMP support statement in Listing 6.4 introduces a software upgrade, and the resulting profile must specify support for both WML and WBMP. Consequently, the conflict resolution mechanism must combine the values of the two `supported` statements. In contrast, the user defined image dimensions are meant to override the default device statements. Consequently, the conflict resolution mechanism must favour the statements of Listing 6.5.

The resolution of conflicting adaptation statements can depend on several factors. Specifically, the resolution of conflicting image dimension specifications is *entity dependent*, because the document user statements override the device defaults. The conflict resolution of the `supported` statements is *term dependent*, because the term `supported` expresses a cumulative property: the introduction of new technologies extends the existing set of supported technologies. *Statement dependent* conflict resolution can be beneficial for overriding existing entity dependent or term dependent resolution mechanisms. For instance, a document user who wishes to retrieve an image-free document must be able to introduce a `supported` statement that overrides the `supported` term

[1]The document user can explicitly provide such adaptation statements, but no exposure to the adaptation profile syntax is necessary. For instance, a graphical user interface to document processing, such as a browser, can automatically derive the user preferences profile from the its internal configuration.

conflict resolution, in order to remove the WBMP support statement. Consequently, the illustrated profile partitioning demonstrated that the resolution of conflicting adaptation statements can be entity dependent, term dependent or statement dependent.

A generic conflict resolution mechanism must not depend on predefined sets of either adaptation requirement partitions or requirement specifying entities. Specifically, the optimal partitioning of a profile is a design decision that can be specific to each case of adaptation requirements specification. For instance, the partitioning of a particular set of adaptation requirements might depend how they are distributed, if it aims to optimise their transmission efficiency. Additionally, the preprocessing of a document can be influenced by a fixed set of external entities: the document user, the browser and the device. However, the specification of their adaptation requirements may involve additional entities. For instance, the browser capabilities might be specified by a combination of a default profile and several software upgrade profiles.

A generic conflict resolution mechanism can avoid such predefined sets and methods by using binary adaptation expressions and a predefined ordering guideline. Specifically, within the context of a well defined ordering guideline, binary adaptation expressions, between a preceding and a conflicting value, can resolve the majority of statement conflicts. For instance, if default statements occur before custom statements, a binary expression that always returns the conflicting value can resolve the `maxIntegerX` conflicts. Additionally, an order independent union, between the preceding and conflicting values, can resolve the conflicts between the `supported` statements. The sufficiency of such binary expressions cannot be proven, because of the unlimited variety of methods to partition adaptation requirements and resolve conflicting statements. Nevertheless, they cover several conflict resolution scenarios and they are adequate for a generic conflict resolution mechanism, because they do not rely on fixed requirement partitioning methods.

6.3.2 XMLPipe composite profiles

A profile composition mechanism that is based on an ordering guideline requires an ordered set of adaptation statements. However, the introduced adaptation profiles alone are not adequate, because they are *sets* of adaptation statements. The XMLPipe conflict resolution will use the concept of *composite profiles*. A composite profile is a *sequence* of *composite statements* and of external references to other *composite profiles*. A composite statement is similar to an adaptation statement, but it can also include a conflict resolution expression. *CProfiles* will represent the set of composite profiles, and *CStatements* will represent the set of composite adaptation statements.

A composite profile provides the necessary foundation for profile composition, because it allows arbitrary partitioning of the adaptation requirements, provides ordered adaptation requirements and includes statement-specific expressions for conflict resolution. Specifically, the preprocessing initiation entity can use the composite profile external references to arbitrarily partition a composite profile, into a sequence of sub-profiles. Subsequently, the preprocessing initiation entity must only transmit a subset of the sub-profiles. The preprocessing implementation can choose the most efficient method to retrieve the remainder of the referenced sub-profiles. Moreover, both inline and referenced composite statements are ordered, because a composite profile is a *sequence* of statements and external references. Finally, each composite statement can contain an optional statement specific expression for conflict resolution.

Composite statements (*CStatements*): *CStatements* is the set of all composite statements and it contains all pairs of an adaptation statement and an optional binary adaptation expression, which maps two values of the corresponding term type to a value of the same type.

$$CStatements = \bigcup_{\forall \tau \in Terms} \left((\{\tau\} \times termType\,(\tau)) \times \left(\mathcal{F}^{termType(\tau)}_{(termType(\tau))^2} \cup \{\epsilon\} \right) \right)$$

Composite profiles (*CProfiles*): The set of all *composite profiles CProfiles* contains all composite statement *sequences*.

$$CProfiles = \bigcup_{\forall n \in \mathbb{N}} (CStatements)^n$$

6.3.3 XMLPipe profile composition

The XMLPipe *profile composer* component is based on a predefined ordering guideline and three prioritised levels for the specification of conflict resolution expressions: a default resolution expression, adaptation term specific expressions and statement specific expressions, in ascending priority order. The profile composer maps the composite profile, which is provided by the preprocessing initiation entity, to a conflict-free adaptation profile, that can be used by other XMLPipe components.

An ordering guideline, which is adequate for an extensible adaptation model, must not depend on predefined categories of adaptation terms. The XMLPipe ordering guideline avoids such dependencies by using the abstract concept of document processing relevance. Specifically, the composite statements must be ordered in ascending relevance to the document processing. For instance, consider a set of adaptation requirements, where both browser and device capabilities consist of their default and custom portions. The browser can be considered as more relevant to the document processing than the device, and custom adaptation requirements are more relevant than default ones:

default device statements → custom device statements → default browser statements → custom browser statements → user preferences

The combination of such an ordering guideline with three prioritised levels of conflict resolution enable the resolution of most conflicting statements. Specifically, the default XMLPipe conflict resolution always favours a newly introduced value, because it must be more relevant to the document processing, according to the above ordering guideline. A term conflict resolution expression, which is specified in a term's semantics, overrides the default conflict resolution, in order to allow term-specific conflict resolution. Statement specific resolution expressions have the highest priority, in order to allow the preprocessing initiation entity to override both the default and the term-specific resolution mechanisms.

The XMLPipe profile composer implements the function *ccompose*, which uses the three conflict resolution levels to convert a composite profile into an adaptation profile.

ccompose iterates through the sequence of ordered composite statements and adds them to the output adaptation profile. If a source composite statement does not conflict with a previous output adaptation statement, *ccompose* adds it to the resulting profile. If it conflicts with a previous statement and it includes a conflict resolution expression, *ccompose* evaluates the expression over the two conflicting values, and uses its result for the output profile. Otherwise, *ccompose* uses the term specific resolution expression, if it exists. If there is no term specific resolution expression, *ccompose* applies the default conflict resolution and substitutes the old statement, in the output adaptation profile, with the conflicting composite statement, in the source composite profile.

Profile composition (*ccompose*): The profile composition function $ccompose : CProfiles \rightarrow Profiles$, maps a composite profile $cpr \in CProfiles$ to its corresponding adaptation profile pr:

function $ccompose(cpr) \rightarrow pr$
  Let $cpr = (((\tau_1, v_1), f_1), ((\tau_2, v_2), f_2), \ldots, ((\tau_n, v_n), f_n))$
  Let $pr = \emptyset$
  for $(i = 1 \ldots n)$
    if $\nexists (\tau, v) \in pr$ where $\tau = \tau_i$ then
      $pr = pr \cup (\tau_i, v_i)$
    else
      if $f_i \neq \epsilon$ then
        $pr = (pr - (\tau, v)) \cup \{(\tau, f_i(v, v_i))\}$
      else if $termResolve(\tau) \neq \epsilon$ **then**
        $f = termResolve(\tau)$
        $pr = (pr - (\tau, v)) \cup \{(\tau, f(v, v_i))\}$
      else
        $pr = (pr - (\tau, v)) \cup \{(\tau, v_i)\}$
      end if
    end if
  end for
end function

6.3.4 Profile composition example

Listing 6.6(a) illustrates a composite profile example that uses two external references, in order to allow more efficient transmission of the adaptation requirements. Specifically, it includes two user preference statements and two external references to a default and a custom mobile profile, illustrated in Listings 6.3 and 6.4, respectively. A preprocessing initiation entity, such as a mobile user over a low bandwidth mobile network, must only transmit the document in Listing 6.6(a) to the preprocessing implementation. The preprocessing implementation, which may run on a mobile proxy within a faster network, can subsequently import the external profiles.

The profile composition process must resolve all statement conflicts and produce the initial mobile adaptation profile, which has been illustrated in 6.2 (page 97). Firstly, the profile composition process combines all sub-profiles into a composite profile, which

(a)

```
1  <profile name="mobileUser">
2   <include
3     ref="mobileDefaultURI"/>
4   <include
5     ref="mobileCustomURI"/>
6   <statm ...
7      name="maxImageX">96</statm>
8   <statm ...
9      name="maxImageY">100</statm>
10 </profile>
```

(b)

```
1  <profile name="mobileUser">
2   <statm ... name="deviceType">
3     mobile
4   </statm>
5   <statm ... name="supported">
6     <item>WML URI</item>
7   </statm>
8   <statm ...
9     name="maxImageX">150</statm>
10  <statm ...
11    name="maxImageY">150</statm>
12  <statm ... name="supported">
13    <item>WBMP URI</item>
14  </statm>
15  <statm ns="xmlPipeUri"
16    name="maxImageX">96</statm>
17  <statm ns="xmlPipeUri"
18    name="maxImageY">100</statm>
19 </profile>
```

(c)

```
1  <term ns="..."
2   name="supported"
3   type="SetOfStrings">
4   <default/>
5   <resolution>
6     <union>
7       <prevTermVal/>
8       <termVal/>
9     </union>
10  </resolve>
11 </term>
```

(d)

```
1  <statm ... name="supported">
2   <item>WML URI</item>
3   <resolve><expr>
4      <termVal/>
5    </expr></resolve>
6  </statm>
```

Listing 6.6: Profile composition example

is illustrated in 6.6(b). Subsequently, *ccompose* resolves all statement conflicts. Specifically, it initiates the profile composition by copying the statements in lines 1–11, of Listing 6.6(b), because they do not introduce any conflicts. The `supported` statement at line 12 conflicts with the adaptation statement at line 5. Listing 6.6(c) illustrates a simplified representation of the `supported` term semantics. The semantics specify a term's type, default value (empty set) and default conflict resolution expression. The latter returns the union of the two conflicting values: $f(fvalue_1, fvalue_2) = v_1 \cup v_2$. *ccompose* uses the term conflict resolution expression to resolve the introduced conflict. Specifically, it updates the resulting profile's `supported` statement to specify both WML and WBMP support. The image dimension statements in lines 15–18 conflict with the ones in lines 8–11. We assume that the corresponding terms do not define an explicit conflict resolution expression. Therefore, *ccompose* applies the default XMLPipe conflict resolution, and it favours the latest occurring values. The resulting adaptation profile is equivalent to the profile in Listing 6.2.

Statement conflict resolution expressions can override the default behaviour. For instance, if the statement of Listing 6.6(d) was appended to the initial composite profile, the resulting profile would differ from Listing 6.2. The introduced statement includes the conflict resolution expression $f'(fvalue_1, fvalue_2) = v_2$, which is the same as the default XMLPipe conflict resolution that always favours the most recent value. The statement expression would override the `supported` term resolution expression, and the resulting profile would only declare WML support, as opposed to both WML and

WBMP. Such statement specific expressions ensure that the preprocessing initiation entity can override the default conflict resolution mechanism when necessary.

6.4 Binding adaptation specification

An adaptation model must define an adequate method for choosing the optimal alternative over a set of independently developed transformations. Specifically, the optimal interpretation of a document can differ for separate sets of adaptation requirements. Consequently, the transformation semantics binding can depend on the adaptation requirements, as it has been illustrated in Section 4.4. For each language, its alternative transformation specifications can be developed by several independent semantics authors. Therefore, the adaptation model must define an adequate set of adaptation requirement queries, which can be associated with each transformation specification, and a corresponding evaluation method that allows the selection of the optimal transformation specification, for an adaptation profile.

The XMLPipe transformation selection mechanism is based on two adaptation measures: the *adequacy measure* and the *applicability measure*. The adequacy measure evaluation uses a set of adaptation expressions that are sufficiently constrained to provide a comparable adequacy measure, between independently developed transformation specifications. In contrast, the applicability measure evaluation uses a single arbitrary complex expression, which enables semantics authors to precisely specify whether a transformation applies to an adaptation profile. Each transformation is associated with an applicability expression and a set of adequacy expressions. Their evaluation provides a comparable measure, which allows the selection of the optimal transformation specification, between a set of independently defined alternatives. The subsequent sections 6.4.1, 6.4.2 and 6.4.3 will describe the details of the adequacy measure, the applicability measure and the combined adaptation measure, respectively.

6.4.1 The adequacy measure

The adequacy measure evaluation uses a set of *adequacy expressions*, which are purposely constrained, in order to correspond to a single adaptation requirement. Specifically, an adequacy expression is an adaptation expression that can only use the value of a single adaptation term. Therefore, the evaluation of an adequacy expression can only represent the fulfilment of a single adaptation requirement, because each adaptation term corresponds to a distinct adaptation requirement. In order to ensure that adequacy expressions provide comparable results, their return values are constrained to be either numeric, within the $[0, 1]$ range, or boolean. A result of 1 or `true` represents full satisfaction of the corresponding adaptation requirement. In contrast a result of 0 or `false` represents no requirement satisfaction. Intermediate numeric values represent partial satisfaction. The set EAd will represent the set of all adequacy expressions. Each member of EAd is a pair of a term and a unary expression that evaluates to either a *Numeric* or a *Boolean* value.

Adequacy expressions (EAd): The set EAd of all adequacy expressions contains all pairs of terms and unary expressions that evaluate to a *Numeric* or a *Boolean* value.

$$EAd = \bigcup_{\forall \tau} \left(\left(\{\tau\} \times \mathcal{F}^{Numeric}_{(termType(\tau))^1} \right) \cup \left(\{\tau\} \times \mathcal{F}^{Boolean}_{(termType(\tau))^1} \right) \right)$$

If each transformation specification is associated with a set of adequacy expressions, a weighted sum of their evaluation could provide an absolute measure of the transformation's adequacy, for an adaptation profile. Specifically, the transformation's adequacy can relate to multiple adaptation requirements. The evaluation of each adequacy expression provides a comparable measure of the transformation's adequacy, according to a distinct adaptation requirement. If the relative importance of the individual adaptation terms was well defined, such a weighted sum would provide a measure of a transformation's adequacy for an adaptation profile.

However, the relative importance of the adaptation terms cannot be well defined, within a generic and extensible processing model. Firstly, predefined importance relationships are not adequate, because adaptation expressions can use an open set of adaptation terms. Moreover, the relative importance of terms can vary according to the individual adaptation scenarios. For instance, consider a transformation specification T, which is only adequate for WML mobile browsers that run on 100 or more pixels wide displays. An adequacy expression that requires WML support can be considered as equally important to an expression that requires a sufficiently wide display, because the presentation of T output requires both. In contrast, the WML support expression is more important for an alternative transformation T' that *optimises* its output for displays that are wider than 100 pixels, but it is also adequate for smaller displays. Therefore, a weighted sum of the two adequacy expressions is not sufficient for choosing between T and T', since the relative importance of the two adaptation requirements is not well defined. Consequently, the measure evaluation cannot use a weighted sum of the adequacy expression evaluations.

The XMLPipe adequacy measure evaluation avoids the ambiguous weighted sums, and it favours extensibility and generality by considering all adequacy expressions as equally important. Specifically, such an approach cannot cover all adequacy expression cases, but, since their relative importance cannot be well defined, a generic approach can only assume that they are equally important. Therefore, if a transformation is associated with a set of adequacy expressions, its adequacy measure for an adaptation profile can be obtained by the sum of their evaluation. BA will represent all sets of adequacy expressions. The *adequacy* function will map a pair of a profile and a set of adequacy expressions to the corresponding adequacy measure. In addition to summing the evaluation results of all adequacy expressions, *adequacy* is also responsible for normalising their results to the predefined adequacy expression range. Moreover, when an adaptation expression refers the value of a term that is not specified in an adaptation profile, *adequacy* uses its corresponding default value.

6.4.2 The applicability measure

The *applicability measure* and the *applicability expressions* are necessary, because adequacy expressions are too constrained to sufficiently express the applicability of a transformation specification, for an adaptation profile. Specifically, the adequacy expressions have been restricted to only access the value of a single term, in order to provide comparable adequacy measures, which are essential for independently developed transformation specifications. However, more complex adaptation expressions are necessary for specifying the applicability of transformation specification, because it might depend on an arbitrary combination of multiple adaptation requirements.

For instance, consider two transformation specifications T and T'. T produces

Set of adequacy expression sets (BA): BA is the set of all binding expression sets: $BA = \wp(EAd)$

Adequacy measure function ($adequacy$): The adequacy measure function $adequacy : Profiles \times BA \rightarrow Numeric$ maps a profile and a set of adequacy expressions to the corresponding adequacy measure.

$$adequacy(pr, ba) = \sum_{\forall ead \in ba} mt(pr, ead)$$

where the function $mt : Profiles \times EAd \rightarrow Numeric$ is defined as:

$$mt(pr, (\tau, f)) = \begin{cases} 1 & , m \in (1, \infty) \cup \{true\} \\ m & , 0 \leq m \leq 1 \\ 0 & , m \in (-\infty, 0) \cup \{false\} \end{cases} \text{, where}$$

$$m = \begin{cases} f(v) & , \exists (\tau, v) \in pr \\ f(termDefault(\tau)) & , \nexists (\tau, v) \in pr \end{cases}$$

WML documents, and T' produces a combination of WML mark-up and WBMP images. ba and ba' will represent their corresponding sets of adequacy expressions.

$$ba = \{(\texttt{supported}, \texttt{WMLURI} \in v)\}$$
$$ba' = \{(\texttt{supported}, \texttt{WMLURI} \in v), (\texttt{supported}, \texttt{WBMPURI} \in v)\}$$

Their common adequacy expression evaluates to *true* for adaptation profiles that specify WML support. The second adequacy expression of ba' evaluates to true only for adaptation profiles that specifie WBMP support. Additionally, consider the adaptation profiles pr_1, pr_2 and pr_3 that specify WML support, WML/WBMP support and neither WML nor WBMP support, respectively.

Within the context of profiles pr_2 and pr_3, ba and ba' are sufficient for evaluating the applicability of T and T'. Specifically, according to profile pr_2, both T' and T are applicable, because it specifies support for both WML and WBMP. The corresponding adequacy measures return positive values that reflect this observation: $adequacy(pr_2, ba') = 2$, $adequacy(pr_2, ba) = 1$. In a similar manner, neither T nor T' apply to pr_3, because it does not support the required WML rendering. The adequacy measures also reflect this observation: $adequacy(pr_3, ba') = adequacy(pr_3, ba) = 0$.

However, an adequacy measure does not always provide an applicability measure. Specifically, T' is not adequate for pr_1, because pr_1 does not state support for WBMP. However, the adequacy measures for both T and T' are positive: $adequacy(pr_1, ba') = adequacy(pr_1, ba) = 1$. No combination of adequacy expressions can express the applicability of T', because it requires WML *AND* WBMP support. However, the *adequacy* function sums the results of the individual expressions, and it corresponds to a binary *OR* of the individual expressions. Therefore, separate applicability expressions are necessary.

The XMLPipe applicability expressions can be arbitrarily complex and refer to multiple terms, in order to precisely express the applicability of a transformation. Applicability expressions do not have to be as constrained as the adequacy expressions, because their evaluation does not have to produce a comparable measure. Nevertheless,

the range of their evaluation result is constrained, because it must provide an unambiguous applicability measure. *EAp* will represent the set of all XMLPipe *applicability expressions*. Each *EAp* member is an adaptation expression over a tuple of term values, which can result to either a *Numeric* or a *Boolean* value. For instance, the applicability expression of T' must require WML and WBMP support:

$$eap = \{(\texttt{supported}, \texttt{supported}),\ \texttt{WMLURI} \in v_1\ AND\ \texttt{WBMPURI} \in v_2\}$$

The *applicability* function maps a pair of an applicability expression and an adaptation profile to their corresponding applicability measure.

Applicability expressions (*EAp*): The set *EAp* contains all applicability expressions. Each *EAp* member is a pair of an n-tuple of terms and of an adaptation expression, which has a corresponding n-tuple of arguments. The expression must evaluate to either a *Numeric* or a *Boolean* value.

$$EAp = \bigcup_{\forall n \in \mathbb{N}} \bigcup_{\forall \tau_1, \ldots \tau_n \in Terms} \bigcup^{\forall Type \in \{Numeric, Boolean\}} \left(\{(\tau_1, \ldots, \tau_n)\} \times \mathcal{F}^{Type}_{\{termType(\tau_1), \ldots, termType(\tau_n)\}} \right)$$

Applicability measure function expression (*applicability*): The applicability measure function $applicability : Profiles \times EAp \rightarrow Numeric$ maps a pair of a profile and an applicability expression to the corresponding applicability measure:

$$applicability(pr, ((\tau_1, \ldots \tau_n), f)) = \begin{cases} 1 & , \gamma \in [1, \infty) \cup \{true\} \\ \gamma & , 0 \leq \gamma \leq 1 \\ 0 & , \gamma \in (-\infty, -1] \cup \{false\} \end{cases}$$

where $\gamma = f(v_1, \ldots v_n)$ and $\forall i \in [1, n]$

$$v_i = \begin{cases} v & , \exists (\tau_i, v) \in pr \\ v' & , \nexists (\tau_i, v) \in pr, termDefault(\tau_i) == v' \end{cases}$$

6.4.3 The adaptation measure

The XMLPipe *adaptation measure* combines the aforementioned applicability and adequacy expressions to allow the selection of the optimal transformation, over a set of alternatives, for an adaptation profile. Specifically, $\mathcal{B}$ is the set of all *binding adaptation specifications*. Each member of $\mathcal{B}$ consists of an optional applicability expression and a set of adequacy expressions. If each transformation is associated with a binding adaptation specification, the adaptation model can choose the optimal transformation specification, over a set of independently defined alternatives.

The adaptation measure evaluator is the adaptation model component that enables the selection of the optimal transformation specifications. It maps a pair of a binding adaptation specification and an adaptation profile to the corresponding adaptation measure. The adaptation measure evaluator implements the *measure* function, which combines the adequacy and applicability measures, returned by the *adequacy* and *applicability* functions, respectively. Specifically, consider a binding adaptation specification $B = (eap, ba)$. If B contains an applicability expression ($eap \neq \epsilon$), *measure* returns the product of the applicability and adequacy measures: $measure(pr, B) =$

$applicability(pr, eap) \times adequacy(pr, BAdset)$. Such a product is an adequate measure, because it equals the comparable adequacy measure, if a transformation is applicable, and it equals zero, if a transformation is not applicable. Moreover, it results to a reduced adequacy measure, when a transformation is partially applicable. If there is no applicability expression, XMLPipe uses the most common applicability expression, which is a logical *AND* of all adequacy expressions. In such a case, *measure* is equivalent to *adequacy*, if all adequacy expressions evaluate to non-zero values, and it equals zero, otherwise.

Binding adaptation specification ($\mathcal{B}$): The set of all binding adaptation specifications $\mathcal{B}$ consists of pairs of an optional applicability expression and a set of adequacy expressions.

$$\mathcal{B} = (\{\epsilon\} \cup EAp) \times BA$$

Adaptation measure function (*measure*): The adaptation measure function $measure : Profiles \times \mathcal{B} \rightarrow Numeric$ provides the absolute adaptation measure that corresponds to a binding specification, according to an adaptation profile.

$$measure(pr, (eap, ba)) = \begin{cases} 0 & \text{, if } eap = \epsilon,\ \exists ead \in ba \text{ so that} \\ & mt(pr, ead) = 0 \\ ad & \text{, if } eap = \epsilon, \text{ and} \\ & \forall ead \in ba, mt(pr, ead) > 0 \\ ap \cdot ad & \text{, otherwise} \end{cases}$$

where $ad = adequacy(pr, ead)$, and $ap = applicability(pr, eap)$, and mt is the function defined in the adequacy measure function definition, in page 105.

Listing 6.7 contains three adaptation specifications that enable the illustration of the adaptation measure usage and the adaptation measure evaluator processing. Consider three independently developed transformations T_1, T_2 and T_3. T_1 and T_2 output XHTML and WML markup, respectively. T_3 outputs WML markup that may reference WBMP images, which are optimised for 100 pixel wide displays. B_1, B_2 and B_3 will represent their corresponding binding adaptation specifications, and they are illustrated in Listings (a), (b) and (c), respectively. Specifically, B_1 contains a single adequacy expression, which declares the adequacy of T_1 for XHTML browsers. B_1 does not require an applicability expression, because it can be inferred from the adequacy expression: T_1 is only applicable to XHTML supporting profiles. In a similar manner, B_2 also includes a single adequacy expression, which declares the adequacy and applicability of T_2 for WML browsers.

In contrast, T_3 requires an an explicit applicability expression (lines 2-9) and a set of three adequacy expressions (lines 10-25). The first two adequacy expressions declare its adequacy for browsers that support WML and WBMP. The third adequacy expression declares the adequacy of T_3 for 100 pixel wide displays. Specifically, it evaluates to 1 for a 100 pixels display, and it tends to 0 for narrower or wider displays: $1 - \left|\frac{\texttt{maxImageX} - 100}{100}\right|$. A display width of 100 pixels is beneficial, but not necessary. Consequently, B_3 contains an explicit applicability expression that only requires WML and WBMP support.

The proposed adaptation measure enables the straightforward selection of the optimal transformation, according to an adaptation profile, because all of T_1, T_2 and T_3

```
<adaptation>
 <adequacy>
  <expr ns="..."
        name="supported">
   <contains>
    <termVal/>
    <val>XHTML URI</val>
   </contains>
  </expr>
 </adequacy>
</adaptation>
```

(a)

```
<adaptation>
 <adequacy>
  <expr ns="..."
        name="supported">
   <contains>
    <termVal/>
    <val>WML URI</val>
   </contains>
  </expr>
 </adequacy>
</adaptation>
```

(b)

```
<adaptation>
 <applicability><expr>
  <and>
   <contains><termVal name="supported"/>
    <val>WML URI</val></contains>
   <contains><termVal name="supported"/>
    <val>WBMP URI</val><contains>
  </and>
 </expr></applicability>
 <adequacy>
   <expr name="supported"><contains>
    <termVal/>
    <val>WML URI</val></contains></expr>
   <expr name="supported"><contains>
    <termVal/>
    <val>WBMP URI</val></contains></expr>
   <expr name="maxImageX">
    <sub>
     <val>1</val>
     <abs><div>
       <sub><termVal/><val>100</val></sub>
       <val>100</val>
     </div></abs>
    </sub></expr>
 </adequacy>
</adaptation>
```

(c)

Listing 6.7: Adaptation binding information example

have an associated binding adaptation specification. Specifically, the optimal transformation is the one that corresponds to the maximum non-zero *measure* result. For instance, consider the adaption profiles pr_1 and pr_2, which have been illustrated in Listings 6.1 and 6.2 (page 97), respectively. Only T_1 is adequate for pr_1, because it is the only XHTML producing transformation. The adaptation measures reflect this observation: $measure(pr_1, B_1) == 1$, $measure(pr_1, B_2) == measure(pr_1, B_3) == 0$. In contrast, T_1 is not adequate for pr_2, which declares support for WML mark-up, WBMP images and a 96 pixels wide display. T_2 and T_3 are both applicable. T_3 is the optimal alternative, because it utilises the WBMP presentation capability and its output is optimised for 100 pixel wide displays. *measure* evaluation is consistent with these observations: $measure(pr_2, B_1) == 0$, $measure(pr_2, B_2) == 1$ and $measure(pr_2, B_3) == 2.96$. Therefore, when transformation specifications are associated with binding adaptation specifications, the adaptation measure evaluator allows the identification of the optimal alternative, for each adaptation profile.

6.5 The complete adaptation model

The complete XMLPipe adaptation model consists of the *profile composer*, the *adaptation measure evaluator* and the defined term semantics, as illustrated in Figure 6.2.

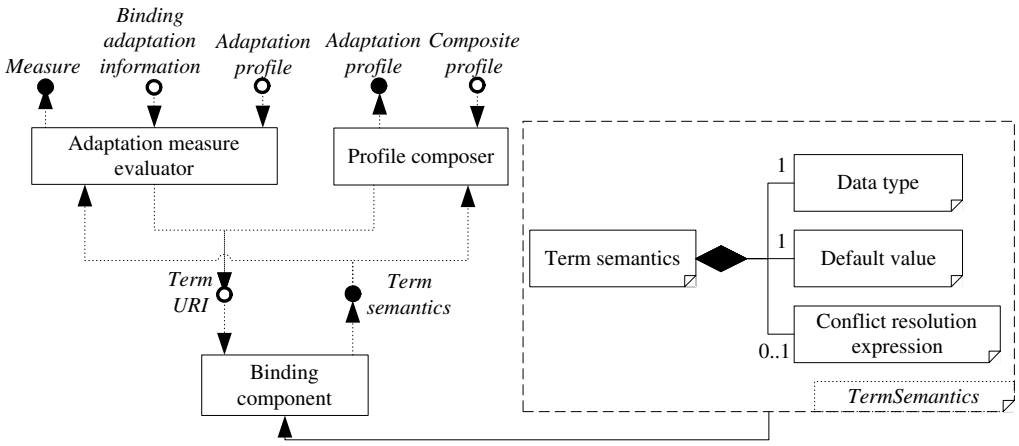

Figure 6.2: Adaptation requirements processing

The *profile composer* component implements the *ccompose* function, which maps the externally provided composite profile to a conflict-free adaptation profile. *ccompose* uses either the default XMLPipe conflict resolution mechanism or explicit conflict resolution expressions, which can be specified in the term semantics or the composite statements. Each XMLPipe transformation must be associated with a *binding adaptation specification*, which expresses its dependency to the adaptation requirements. The *adaptation measure evaluator* component implements the *measure* function, which maps pairs of adaptation profiles and binding adaption specifications to comparable adequacy measures. These adequacy measures enable the straightforward identification of the optimal transformation, for an adaptation profile, because it is the one that corresponds to the maximum positive adequacy measure. Both the adaptation measure evaluator and the profile composer interact with the binding component, in order to obtain all necessary term semantics.

6.6 Discussion

The preprocessing of presentation documents requires a well defined representation of adaptation requirements and a well defined way to process them, in order to choose the optimal processing for each document subtree. The proposed XMLPipe binding model covers all necessary concepts. Specifically, the introduced composite profiles cover the external representation of adaptation requirements, because a preprocessing initiation entity can use a composite profile to provide all necessary adaptation information. The profile composer provides the necessary functionality to convert an input composite profile into a conflict-free adaptation profile, which can be accessed by all adaptation sensitive processing components. Finally, the well defined binding adaptation specification and the adaptation measure evaluator allow any component to choose the optimal transformation, over a set of alternatives, for an adaptation profile.

The preprocessing framework covers both the representation of the adaptation requirements and their processing. Specifically, it requires that an adaptation model must

define a composite and extensible representation of adaptation requirements. Additionally, it requires a well defined methodology for combining independently developed sets of adaptation requirements and for choosing the optimal transformation over a set of alternatives.

An extensible adaptation requirement representation that allows an open set of adaptation terms is essential for covering the multitude of current and future adaptation requirements. The proposed adaptation terms, adaptation statements and adaptation profiles adopt the extensible CC/PP concept of URI qualified names, and they ensure the extensibility of the proposed representation. Moreover, the proposed adaptation term semantics provide all the necessary information for processing adaptation requirements that use an open set of adaptation terms. Furthermore, the default term values enable the semantics authors to liberally use newly introduced adaptation terms. If a profile does not specify a value for a newly introduced term, the measure evaluator can use the corresponding default value to evaluate any expressions that reference the new term. Consequently, the XMLPipe adaptation model allows both the representation and the processing of an open set of adaptation requirements.

The introduced concept of composite profiles and the profile composer enable the use of composite adaptation requirement specifications. Specifically, the composite profile external references enable the arbitrary combination of distributed adaptation requirement sources. The profile composer is based on an ordering guideline. The ordering guideline is adequate for the Web, because it allows the specification of conflict resolution expressions, without requiring explicit relationships between the adaptation requirement sources. Additionally, the three prioritised levels of conflict resolution expressions enable the fine grained specification of how to resolve any introduced conflicts, during profile composition. The default XMLPipe conflict resolution always favours the values that are more relevant to the document processing. The entities that define the adaptation terms or the adaptation statements can override the default behaviour by providing term and statement specific resolution expressions, respectively.

The CSS recommendation uses a similar ordering guideline, in order to resolve conflicts between style attribute declarations. Specifically, it orders all declared style attributes according to three factors: the entity that specifies each attribute, the specificity of each declaration and the order of the declaration. The entity dependent ordering is based on a set of three predefined entities: the user agent, the document author and the document user. The specificity is a measure of how specific is a declaration for a document element. A CSS implementation resolves conflicts by always using the latest occurring declaration, according to the above ordering, because it is the most specific declaration that is defined by the most relevant entity. Predefined sets of entities can allow fine grained resolution, but the XMLPipe adaptation model does not rely on such sets, because they are not adequate for the resolution of conflicting adaptation requirements, as described in Section 6.3.1. In contrast, it requires that the adaptation statements are ordered, according to their document processing relevance. Such an ordering allows the resolution of conflicting statements that can be specified by a multitude of entities. Additionally, XMLPipe allows the specification of conflict resolution expressions, which enable more fine grained resolution than simply choosing one of the conflicting values. Consequently, the CSS conflict resolution model can be adequate for the constrained domain of style attributes specification, but the XMLPipe conflict resolution proposal is more applicable for the composition of extensible adaptation profiles.

The proposed adaptation requirements representation is more appropriate for a preprocessing model than CC/PP, because of the introduced term semantics and profile composer. CC/PP is the most prominent existing approach for representing adaptation requirements. The XMLPipe adaptation model adopts the CC/PP concept of URI qualified adaptation terms, which is essential for an extensible representation of an open set of adaptation requirements. Furthermore, the XMLPipe adaptation model also covers the processing of an open set of adaptation requirements, as opposed to CC/PP, which only covers their representation. Specifically, the XMLPipe concepts of adaptation expressions and term semantics are the basis for the well defined processing, for an open set of requirements. Each term has a well defined default value, which ensures that semantics authors can specify adequacy expressions that use newly introduced adaptation terms, without requiring that all adaptation profiles specify their values. The proposed profile composer allows fine grained conflict resolution, which enables the integration of independently developed profiles. CC/PP does not enable such integration, as it does not provide a conflict resolution mechanism. Moreover, it cannot provide the foundation for supporting the proposed conflict resolution mechanism; CC/PP is RDF-based, and it is not designed to preserve the order of the adaptation statements, which is essential for the proposed ordering guideline.

The proposed adaptation measure evaluator enables the straightforward selection of an optimal transformation, over a set of independently developed alternatives. Specifically, each transformation must be associated with an applicability expression and a set of adequacy expressions. The evaluation of each adequacy expression provides a measure of the extent to which a transformation fulfils an adaptation requirement. Under the assumption that all adaptation requirements are equally important, the set of adequacy expressions can be used to calculate a comparable adequacy measure, for each alternative transformation. The adaptation measure evaluator composes the adaptation measure out of the comparable adequacy measure and of the applicability measure, which ensures that the adaptation measure is positive only for applicable transformations. The identification of the optimal transformation specification, for an adaptation profile, is straightforward, because the optimal transformation is the one that corresponds to the highest positive adaptation measure.

The XMLPipe adaptation model does not adapt documents in itself, but it provides the necessary foundation for more powerful adaptation than existing generic adaptation approaches. The CC/PP based stylesheet selection, proposed in [OH02], is the most prominent existing approach for generic adaptation. Nevertheless, it uses simplistic CC/PP queries and it does not allow the independent development of the individual stylesheets. In contrast, the proposed combination of applicability and adequacy expressions enables significantly more powerful adaptation. The applicability expressions provide a precise specification of when a transformation applies to an adaptation profile, and the adequacy expressions allow the evaluation of the relative adequacy of multiple independently developed specifications.

6.7 Summary

A well defined representation and processing of the adaptation requirements is necessary, because both the binding of transformation specifications and the individual transformation components depend on the adaptation requirements. The proposed

XMLPipe adaptation model covered their representation, by introducing the composite profiles and the adaptation profiles. Additionally, it defined the necessary processing for composing independently defined profiles and for choosing the optimal transformation specifications, over a set of independently developed alternatives.

Specifically, the internal XMLPipe representation of adaptation requirements is an *adaptation profile*, which consists of a set of *adaptation statements*. Each adaptation statement is a pair of an *adaptation term* and of its corresponding value. Adaptation terms are uniquely identified by URIs, which allow the unambiguous representation of an open set of adaptation requirements. In contrast, the proposed external representation consists of *composite profiles*, which are *sequences* of statements. The *profile composer* component maps the externally provided composite profiles into adaptation profiles, and it is responsible for resolving any conflicting adaptation requirements.

The *adaptation measure evaluator* is responsible for supporting the selection of the optimal transformation specification, according to an adaptation profile. The XMLPipe adaptation model requires that each transformation specification is associated with an adaptation binding specification, which consists of an applicability expression and a set of adequacy expressions. The *adaptation measure evaluator* maps a pair of an adaptation binding specification and an adaptation profile to an absolute comparable adaptation measure. Within the context of an adaptation profile, the transformation specification that corresponds to the higher adaptation measure is considered as the optimal transformation for a document subtree.

The proposed processing and representation of adaptation requirements fulfill the corresponding preprocessing framework requirements, and they compare favourably to the most prominent existing approaches. Therefore, they provide an adequate foundation for the adaptive transformation of presentation documents. The next chapter will describe the XMLPipe transformation model, which interoperates with the introduced adaptation model, in order to choose the optimal transformation for each document subtree.

Chapter 7

XMLPipe transformation model

This chapter describes the XMLPipe transformation model, which is the core XMLPipe sub-model, and it defines the necessary processing for adapting a presentation document, according to a set of adaptation requirements. The XMLPipe transformation model is based on the proposed integration and adaptation models. Specifically, it uses the XMLPipe integration model, described in Chapter 5, to enable the combination of independently developed transformation specifications for processing mixed namespace documents. Moreover, it uses the adaptation model, described in the previous chapter, which enables the selection of the optimal transformation specifications, according to an adaptation profile.

The XMLPipe transformation model consists of the integration model transformation driver and the built-in transformation pipelines. The former drives the document transformation process, and it is based on a subtree separation process and a recursive postorder document traversal. The transformation driver interoperates with the adaptation measure evaluator to choose the optimal transformation semantics for each document subtree. The XMLPipe transformation semantics are specifications of transformation pipelines, which are essential for enhancing the existing transformation functionality. Each XMLPipe transformation pipeline consists of multiple atomic transformations that encapsulate the existing transformation functionality under a common interface, in order to enable the seamless integration of a multitude of existing transformation technologies.

Section 7.1 overviews the existing transformation approaches, which were initially described in Section 2.3, and it identifies the core issues that a transformation model must address. Sections 7.2 and 7.3 establish the necessary foundation for describing the XMLPipe transformation model. The former establishes a driving example of a presentation document that assists the subsequent transformation model description. The latter establishes the fundamental notation and the necessary assumptions, for illustrating that mixed namespace document transformation is feasible, in Section 7.4. Section 7.5 proceeds to the description of the binding and selection of the transformation semantics. Section 7.6 loosens the most restrictive assumptions, in order to extend the applicability of the proposed transformation. Finally, Section 7.7 introduces the XMLPipe pipelines and atomic transformations, and Section 7.8 combines all the introduced components into the XMLPipe transformation model.

7.1 Transformation model considerations

The XMLPipe transformation model cannot be based on existing transformation technologies, because they do not allow generic transformation of mixed namespace documents. Specifically, modular transformation specifications, such as XSL-T, allow the composition of separate single namespace specifications. However, such compositions use predefined language relationships and have the same drawbacks as profile-based approaches. Sequential composition approaches, such as the XEBRA browser[THHH01] (described in Section 2.5), do not necessarily require predefined language combinations. However, sequential composition is not sufficient for the generic processing of mixed namespace documents, as described in Section 4.4.

A preprocessing model benefits from built-in support of transformation pipelines. As described in Section 4.4, transformation pipelines allow simpler modular transformation specifications, and they enable seamless integration and extension of the existing transformation functionality. Consequently, the XMLPipe transformation semantics can define the necessary subtree transformations using transformation pipeline specifications, as opposed to plain transformation specifications.

Existing transformation pipeline approaches fit into a single framework: they compose complex transformations out of transformers, sources, mergers and sinks, as described in Section 2.3.2. The XMLPipe built-in transformation pipeline mechanism can adopt the existing concepts by extending them to cover document subtree processing and adaptive transformation composition. Subtree processing is necessary, because the proposed integration model partitions the processing of a document into the processing of its individual subtrees. Adaptive transformation composition is beneficial, because it allows the composition of adaptive transformations out of existing transformation specifications.

A preprocessing model must define the necessary interoperation, between its validation and transformation models, in order to validate the input of transformation processes prior to their execution. Such interoperation is necessary, because the result of a transformation is only well defined if its input is valid. Integrated validation approaches, such as XDuce, are an efficient method to ensure the validity of document subtrees, as described in Section 2.2. However, they are inadequate for generic document processing, because they require constrained processing environments and constrained development of transformation specifications. In contrast, transformation pipeline based validation, such as in the XML Pipelines (described in Section 2.3.2), is more adequate, because it enables the liberal introduction of validation processing steps. XMLPipe can adopt the pipeline based validation by introducing validation processing steps, within its built-in transformation pipelines.

The transformation model is responsible for adapting its input documents. Existing transformation approaches provide either restricted generic adaptation or powerful domain-specific adaptation. Domain-specific adaptation approaches, such as device independent authoring proposals, offer powerful adaptation. However, they only support a narrow set of XML languages, and they are not adequate for generic document processing. CC/PP based stylesheet selection is the most prominent generic approach, but, as described in the previous chapter, the XMLPipe adaptation measure evaluator allows more fined grained transformation selection. Consequently, the XMLPipe transformation model can utilise the proposed adaptation model to transform presentation documents.

Language	*COC*	*FOC*	*SMC*	*COC* placeholders
L_{doc}	`doc:document`, `doc:em`, `doc:img`, `doc:p`			`doc:p`, `doc:em`
L_{alt}			`alt:alt`	`alt:case`
L_{xl}		`xl:href`, `xl:type`		
L_{imp}			`imp:import`	

Table 7.1: Driving example integration model semantics: the handled constructs and the places where content is expected

7.2 A driving example

This section introduces a presentation document driving example, illustrated in Listing 7.1, that will assist the subsequent transformation model discussion. Specifically, `document.xml` is the core example document, and it contains external references to `authors.xml` and `imp.xml`. `document.xml` combines the constructs of the four languages L_{doc}, L_{alt}, L_{imp} and L_{xl}, which are associated with the `doc`, `alt`, `imp` and `xl` namespace prefixes, respectively. Table 7.1 summarises their corresponding integration model semantics, which consist of each language's handled constructs and the places where context is expected.

The constructs of L_{doc} describe a document's layout. Specifically, the `doc:document` element (line 1) introduces a document with multiple nested `doc:section` constructs (such as in lines 11 and 15), which represent its sections and subsections. Each L_{doc} document must have a single title and a set of authors, which are represented by the `doc:title` (such as in lines 4 and 12) and the `doc:authors` (`authors.xml`, line 2) elements, respectively. Each section can contain zero or more paragraphs (`doc:p` – line 13) that can contain text, emphasised text (`doc:em` element – line 13) and inline images (`doc:img` element – line 14). L_{doc} contains four content oriented handled constructs (*COC*):

$$COC(L_{doc}) = \{\texttt{doc:document}, \texttt{doc:em}, \texttt{doc:img}, \texttt{doc:p}\}$$

`doc:document`, `doc:em`, `doc:img` and `doc:p` are *COC* handled constructs, because they introduce the well defined pieces of presentable information that correspond to a document, emphasised content, an image and a paragraph, respectively. In contrast, the remaining L_{doc} constructs introduce context dependent information that cannot be processed independently. For instance, the `doc:title` element introduces the title of its parent element. Arbitrary *COC* subtrees can only occur within the `doc:p` and `doc:em` constructs, because they are placeholders of arbitrary presentable information: most presentable information can be part of a paragraph and can be emphasised.

The `alt:alt` element (line 2) of L_{alt} introduces adaptive content, as a sequence of `alt:case` elements. Each `alt:case` element has an optional `test` attribute, which contains a boolean expression over an adaptation term. The semantics of the `alt:alt` element is to substitute itself with the content of the first adequate `alt:case` element, for an adaptation profile, which must contain either an expression that evaluates to *true* or no expression. The `alt:alt` element is a structure modification construct (*SMC*), because its primary semantics is to modify the document tree. The `alt:case` element is a *COC* placeholder, because it is the parent of the arbitrary substitution alternatives.

```
<doc:document>
 <alt:alt>
  <alt:case test="uri#deviceType=mobile">
   <doc:title>Mobile example</doc:title>
  </alt:case>
  <alt:case>
   <doc:title>Desktop example</doc:title>
  </alt:case>
 </alt:alt>
 <imp:import href="authors.xml" select="**/[@id='MP_DHS']"/>
 <doc:section>
  <doc:title>The doc language</doc:title>
  <doc:p>The root language allows <doc:em>emphasized</doc:em> text,
    images <doc:img href="xmlPipe.gif"/> and nested sections.</doc:p>
  <doc:section>
   <doc:title>Nested section</doc:title>
  </doc:section>
 </doc:section>
 <doc:section>
  <doc:title>Mixed namespace support</doc:title>
  <doc:p>A foreign namespace SMC construct to import textual content:
    <imp:import href="imp.xml" select="**/text/text()"/>, an FOC XLink
    attribute for <doc:em xl:type="simple" xl:href="...">links</doc:em>
    and an SMC subtree that allows adaptation sensitive content:
  </doc:p>
  <alt:alt>
   <alt:case test="http://.../#deviceType=mobile">
    <doc:p>This is a mobile</doc:p>
   </alt:case>
   <alt:case>
    <doc:p>This is NOT a mobile</doc:p>
   </alt:case>
  </alt:alt>
 </doc:section>
</doc:document>
```

document.xml

```
<c:collection>
 <doc:authors id="MP_DHS">
  <doc:author first="M" last="Ped"
      mail="mp49@kent.ac.uk"/>
   <doc:author first="D" last="Shr"
      mail="dhs@kent.ac.uk"/>
 </doc:authors>
</c:collection>
```

authors.xml

```
<root>
  <text>Text node 1</text>
  <text>Text node 2</text>
</root>
```

imp.xml

Listing 7.1: The driving presentation document example

Language L_{imp} introduces the `imp:import` element (line 10), which is also an *SMC*. Its semantics is to substitute itself with the external content that is referenced by its attributes. The `href` attribute specifies the URL of a document, and the optional `select` attribute specifies an XPath expression, which allows the selection of a document portion.

Finally, L_{xl} represents the simple links of the XLink recommendation[DMO01]. L_{xl} uses `xl:href` attribute (line 23), which specifies the link target, and the `xl:type` attribute, which contains the fixed value "simple", which denotes a simple XLink link. Both attributes are functionality oriented constructs (*FOC*), because they amend the presentation of their parent in a well defined way: they convert it into a link.

The XMLPipe transformation model must successfully process `document.xml`, because `document.xml` combines the constructs of the four presentation languages in a meaningful way, according to their semantics. Specifically, the processing of the *SMC* constructs, in lines 2 and 10, provides the required title and author information. According to the L_{alt} semantics, the processing of the `alt:alt` element, in line 2, will result in a `doc:title` element. According to the semantics of L_{imp}, the processing of the `imp:import` element, in line 10, will result in the `doc:authors` rooted subtree, illustrated in `authors.xml`. The processing of the remaining *SMC* occurrences will also result in a valid document: `imp:import` in line 22 introduces textual content within a paragraph, and `alt:alt` in line 26 introduces a paragraph within a section. Furthermore, the L_{xlink} attributes, in line 23, convert their parent `doc:em` element into a link. The presentation interpretation of `document.xml` is well defined: it is a document that contains a title, a set of authors and a sequence of nested sections, which may contain formated text, links and images. Therefore, the XMLPipe transformation model must be able to adapt it to a variety of devices, if the corresponding processing semantics are provided.

A document that has a well defined interpretation is not necessarily valid. For instance, `document.xml` combines the constructs of the example languages in a meaningful way. However, before the processing of the *SMC* constructs, it is not a valid XMLPipe document. Specifically, `document.xml` does not contain the mandatory author information and `doc:document` does not contain the required `doc:title` child, which specifies the document's title. Additionally, the `doc:title` elements, in lines 4 and 7, occur under foreign namespace elements, but they are not handled constructs. Consequently, `document.xml` is not valid according to the XMLPipe integration model.

We will introduce the concept of *semantic correctness*, in order to describe documents that have a well defined interpretation, independently of their validity. For instance, `document.xml` is semantically correct, but it is not valid. The XMLPipe transformation model will focus on the processing of semantically correct documents, because a document that has a well defined interpretation, must also have a well defined processing.

The existence of semantically correct but invalid documents does not indicate a deficiency of the XMLPipe integration model, which defines the validity of mixed namespace documents. The processing of *SMC* constructs can introduce arbitrary tree modifications. An integration model's definition of valid documents cannot account for such modifications, because they can depend on processing semantics that are outside the scope of an integration model. For instance, the document modifications introduced by both the `imp:import` and `alt:alt` constructs depend on their corresponding transformation semantics. An integration model cannot use the transformation semantics of a construct, because they are specific to a possibly independent transformation

model. Nevertheless, the explicit identification of such constructs, by the introduction of the *SMC* category, enables their adequate processing by a preprocessing model. Specifically, as the next chapter will illustrate, the XMLPipe validation model tests the semantic correctness of a document by separately processing its *SMC* rooted subtrees. Consequently, the proposed integration model is not deficient, because it provides the necessary foundation for processing documents that contain *SMC* constructs.

7.3 Transformation fundamentals

Before proceeding to the transformation model proposal, this section introduces the necessary notation and assumptions. Specifically, Section 7.3.1 introduces a precise transformation notation.[1] Section 7.3.2 establishes the necessary assumptions for the subsequent transformation processing proposal.

7.3.1 Mixed namespace transformation notation

The previously introduced transformation notation did not identify the individual languages of a document. Specifically, Chapter 2 introduced the set $\mathcal{T}_{L_1}^{L_2}$, which contains all transformations that map the constructs of a language L_1 to the constructs of a language L_2. If L_1 and L_2 contain constructs from multiple namespaces, the transformation's input and output are mixed namespace documents. However, there was no explicit specification of the individual languages, because there was no well defined relationship between the interpretation of a mixed namespace document and the interpretation of its individual languages.

The introduced concept of integration models enables a fined grained expression of mixed namespace transformations, which reflects the independent definition of the XML languages. Specifically, a mixed namespace transformation T can process documents that combine a set of languages $\mathcal{L}_1$, according to an integration model Im_1. A transformation does not necessarily transform the constructs of all input languages, but it may only transform the constructs of a language subset $\mathcal{L}_2 \subseteq \mathcal{L}_1$. For instance, a transformation that corresponds to a language might be able to transform its constructs, within a mixed namespace document that also contains constructs from additional languages. The output of T can also combine a set of languages $\mathcal{L}_3$, according to an integration model Im_2. The set of *mixed namespace transformations* $\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2}$ will represent all such transformations: $T \in \mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2}$.

Set of mixed namespace transformations ($\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2}$): Consider that $\mathcal{L}_2 \subset \mathcal{L}_1 \in \wp(\mathcal{L})$, $\mathcal{L}_3 \in \wp(\mathcal{L})$, and that Im_1 and Im_2 are two integration models. The set of mixed namespace transformations $\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2} \subset \mathcal{T}$ includes all transformations that process the constructs of languages in $\mathcal{L}_2$ for mapping an input document, which combines the constructs of the languages in $\mathcal{L}_1$ using the integration model Im_1, to an output document, which combines the constructs of the languages in $\mathcal{L}_3$, using the integration model Im_2.

The above notation enables the expression of the required transformation model functionality. Specifically, for each valid mixed namespace document d, the transformation model must combine the transformation semantics of all languages in $\mathcal{L}_{doc}$ to

[1] Such notation could not be introduced in the literature review chapter, because it uses the concept of integration models, which has been introduced by this thesis.

define a transformation T. T must transform d to its most adequate interpretation d', according to an adaptation profile pr. Therefore, $T \in \mathcal{T}^{\mathcal{L}_p:Im_P}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}$, where $\mathcal{L}_p$ and Im_P correspond to the set of natively supported languages and the natively supported integration model, as specified by the adaptation profile pr. Im_X represents the XMLPipe integration model.

7.3.2 Assumptions

The feasibility of generic mixed namespace transformations can only be established under the set of assumptions summarised in Table 7.2. They ensure that a transformation model can transform valid XMLPipe documents, according to an adaptation profile. Each assumption is associated with a unique identifier. The subsequent discussion will use these identifiers to refer to the individual assumptions.

Assumption 1: each non natively supported language must be associated with at least an adequate transformation specification, for each adaptation profile. Such an assumption is necessary to ensure that there is sufficient information to process presentation documents. If there is no adequate transformation specification for a non natively support language, according to an adaptation profile, a preprocessing model cannot create an adequate document interpretation for that profile.

Assumption 2: a transformation specification, which is associated with a language L, must allow the processing of all handled construct rooted subtrees that are valid documents of L. Specifically, the processing of handled construct rooted subtrees is well defined, independently of their context, according to Corollary 1. Therefore, the transformation semantics of L must enable their processing as individual entities. For instance, the `doc:p` rooted subtree, illustrated in lines 13–14 of the driving example, is a valid XMLPipe document, because it is a valid document of L_{doc} and `doc:p` is a *COC* handled construct. The transformation specification that corresponds to language L_{doc} must allow the independent transformation of `doc:p` rooted subtrees, and it must not always require a `doc:document` ancestor. For example, an XSL-T specification can define separate root level transformation templates for each handled construct.

Assumption 3: transformation specifications must copy all unknown content to its corresponding place in the transformation output. Such an assumption ensures that the individual transformations preserve the document's interpretation. Specifically, each document subtree can contain multiple handled construct rooted subtrees, that use constructs of several languages. A transformation, which is associated with a language, must copy the subtrees of other languages, in order to preserve the information they convey. For instance consider the `alt:alt` rooted subtree, in lines 26–33 of the driving example, which contains two `doc:p` constructs. Independent of the invocation order of the individual transformers, the transformation that corresponds to `alt:alt` must copy the contents of `alt:case` to their corresponding place, within its output. Otherwise, the resulting document will lose the information that was enclosed in the `doc:p` elements.

Assumption 4: the design of transformation specifications that correspond to *FOC* or attribute handled constructs must allow their processing within the context of a foreign namespace element. Specifically, the processing of all handled construct rooted subtrees can be defined independently of their context. However, the processing of *FOC* and attribute handled constructs might require access to the their context; consequently, the XMLPipe transformation model must allow the corresponding transformations to also access the context of the handled constructs. For instance, consider a

ID	Assumption	Description
1	Transformation existence	There is an adequate transformation specification for each language not in $\mathcal{L}_p$.
2	Valid subtree transformation	Transformation specifications can process all valid handled construct rooted trees of their corresponding language.
3	Copy unknown content	Each transformation must copy any foreign namespace subtrees to their corresponding place, within the transformation output.
4	*FOC* and attribute context	The transformation specifications that correspond to either *FOC* or attribute handled constructs must be able to process them within the context of a foreign namespace element.
5	No circular dependencies	No circular dependencies between transformation specifications of separate languages.
6	No $\mathcal{L}_p$ dependencies	Transformations must not produce $\mathcal{L}_p$ constructs, the processing of which can introduce non $\mathcal{L}_p$ constructs.
7	Integration models equivalence	$d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ iff $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_P}$.
8	Valid output	If T corresponds to a language L, then $T \in \mathcal{T}_{\mathcal{L}:Im_X(\{L\})}^{\mathcal{L}:Im_X}$.
9	*COC* preservation	A *COC* rooted subtree must be transformed to another *COC* rooted subtree.
10	*SMC* restriction	The transformation of an *SMC* rooted subtree cannot produce a *COC* subtree, unless it occurs at a place where content is expected.
11	*FOC* valid processing	The processing of *FOC* and attribute handled constructs must preserve the semantic correctness of the document. Specifically, they can add ancestors to their parent element, only if • the parent element is a handled construct • it can occur at the corresponding place, within the added ancestors • the root of the new content is a handled construct, which can be a *COC* only if the parent element is a *COC*. Additionally, they can add content to their parent, only if • the added content is valid • it occurs at a place where content is expected, if it is rooted at a *COC* construct.

Table 7.2: Transformation assumptions

transformation T that adapts the L_{xl} constructs, for an XHTML browser presentation. T must have access to the context of the L_{xl} attribute, in order to be able to enclose it within an anchor element, which is the XHTML equivalent of the L_{xlink} simple links. For example, if T processes the `xl:href` attribute, in line 23 of the driving example, it must be able to enclose the `doc:em` element within an anchor XHTML element. Therefore, the transformation model must allow T to process the L_{xl} attributes, within the context of their parent element.

Assumption 5: the transformation of language constructs is not necessarily a one step process, and it can require the recursive application of multiple transformations. In the majority of cases, a transformation maps a construct to an alternative representation that is closer to its natively supported interpretation. However, there are cases where circular relationships may be introduced. A typical example is the transformation of an `imp:import` construct, which can introduce content that includes further `imp:import` constructs. Such relationships must be avoided, in order to ensure the termination of a document's transformation. For instance, consider the XML constructs σ_1 and σ_2 and the transformations T_1 and T_2, which are associated with their corresponding languages. If T_1 transforms σ_1 to σ_2 and T_2 transforms σ_2 to σ_1, the processing of a document that contains either σ_1 or σ_2 will never terminate.

Assumption 6: document preprocessing can only be sufficient, if no transformation outputs natively supported constructs that can require further preprocessing. Specifically, such constructs can introduce non natively supported constructs during the presentation of a document, after the preprocessing has finished. Moreover, they might delay the introduction of constructs that are necessary for document preprocessing. For instance, consider that σ is an $\mathcal{L}_p$ construct that imports arbitrary external content. Additionally, consider that transformation T corresponds to the import language L_{imp} and that it maps the `imp:import` construct to the natively supported σ construct. If the `imp:import` element in line 10 of the driving example is processed by T, the author information will only be imported at presentation time. Consequently, it will not be available for the subsequent preprocessing of the `doc:document` construct, which will fail. Therefore, transformation specifications must not introduce $\mathcal{L}_p$ constructs that may require further preprocessing.

Assumption 7: the XMLPipe transformation model must provide the means to adapt documents, which are valid according to the XMLPipe integration model, to their interpretations, which must be valid according to a natively supported integration model. However, the feasibility of generic mixed namespace transformation can only be proven if the XMLPipe and the natively integration models are closely related.

Firstly, a valid output document must be a valid XMLPipe document. Specifically, a transformation result must be valid, according to the natively supported integration model. Since the transformation of a construct is not necessarily a one step process, a transformation result can contain both natively and non-natively supported constructs. Therefore, if there are non-natively supported constructs, the transformation result must be also valid according to the XMLPipe integration model, because they require further preprocessing. Consequently, the presentation integration model Im_P must be less or equally generic than the XMLPipe integration model Im_X.

Secondly, the inverse integration model relationship must also hold, because a generic transformation component cannot be aware of all presentation integration models. Specifically, there must be no interoperation between the individual transformation specifications, since they can be independently developed. Therefore, the XMLPipe

transformation model, which is specific to the XMLPipe integration model, is responsible for combining the output of the separate transformation results into the resulting document. The transformation driver cannot be a priori aware of all presentation integration models. Therefore, it can only combine the transformation results according to its corresponding integration model, which is the XMLPipe integration model.

The last two observations necessitate the equivalence of the XMLPipe integration model with the presentation integration models, because they must both be equally or less generic than the other. Therefore, a document d belongs to $\mathcal{V}_{\mathcal{L}_d}^{Im_X}$ if and only if $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_P}$. Such an assumption is sufficiently restrictive to offset all XMLPipe practical applications. Specifically, if the integration models must be equivalent, XMLPipe cannot preprocess documents for any existing browser, because no existing browser supports the XMLPipe integration model. Nevertheless, the integration model equivalence is only necessary for proving the feasibility of generic document transformation, but it is not necessary in practice, as Section 7.6 will illustrate.

Assumption 8: the transformation specifications must preserve the document's presentation structure and semantic validity. Specifically, each transformation specification must produce a valid output subtree, when its input is valid. Consequently, since the presentation and XMLPipe integration models must be equivalent, an XMLPipe transformation T, which corresponds to a language L, must belong to $\mathcal{T}_{\mathcal{L}:Im_X(\{L\})}^{\mathcal{L}:Im_X}$. The set of all languages $\mathcal{L}$ is used for both the input and output of T, because the parent element of attribute and *FOC* handled constructs can belong to any language.

Assumptions 9 and 10: *COC* constructs must be both preserved and not arbitrarily introduced, in order to preserve the document's interpretation and validity. Specifically, since only *COC* rooted subtrees introduce well defined pieces of presentable information, the transformation of a *COC* rooted subtree must always result to another *COC* rooted subtree. Additionally, the transformation of an *SMC* construct must not produce a *COC* subtree, unless it occurs at a place where content is expected. For instance consider the *SMC* `imp:import` element, illustrated at line 10 of the driving example. If it is transformed to a *COC* rooted subtree, the resulting document would be invalid, because no content is expected under the `doc:document` element.

Assumption 11: finally, the processing of *FOC* or attribute rooted subtrees can modify their context, but it must do so in a valid way. Specifically, their processing can amend the presentation of their parent by either enclosing it within a subtree or by adding attribute or element descendants. In the former case, the transformation result will only be valid, if the parent is a handled construct, which can occur within the generated subtree. In the latter case, the added descendants must be valid and only occur at places where content is expected if they are *COC*. In both cases, all added content must be rooted at a handled construct. For instance, consider the aforementioned transformation T, which processes the L_{xl} constructs by enclosing their parent element within an XHTML anchor element. T can enclose the `doc:em` element within a `xhtml:a` element, because `doc:em` is a handled construct. The `xhtml:a` element can be the root of the added context, because it is also a handled construct. If `xhtml:a` was not a handled construct, the output of T would be invalid, because `xhtml:a` would be included under the foreign namespace `doc:p` element.

7.4 Transforming valid documents

This section illustrates the feasibility of a generic mixed namespace transformation and defines the corresponding transformation algorithm. Firstly, Section 7.4.1 establishes the relationship between valid documents and their processing. Section 7.4.2 uses this relationship and the assumptions of Table 7.2 to illustrate that there is a well defined transformation T that maps each valid XMLPipe document to its optimal interpretation. Finally, Section 7.4.3 describes an algorithmic definition of T.

7.4.1 Valid documents processing

This section will assist the subsequent transformation investigation by mapping the definition of valid documents (introduced in Section 5.3.1) to its document processing equivalent and introducing an alternative definition of valid documents, which is based on a sequence of compositions.

The set of XMLPipe valid documents can be expressed in a non-recursive manner. Corollary 2 is an alternative valid document definition that uses a sequence of single namespace tree compositions, as opposed to the recursive composition of mixed namespace trees.

Corollary 2 *$d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ if and only if there is a sequence of $n \geq 1$ single namespace valid documents $d_1, \ldots, d_n$, so that $d \in (d_n \cdots \underset{Im_X}{\overset{+}{\leftarrow}} (d_3 \underset{Im_X}{\overset{+}{\leftarrow}} (d_2 \underset{Im_X}{\overset{+}{\leftarrow}} d_1)) \cdots)$, where $\forall i \in [1, n]$, d_i is a valid single namespace document of L_i ($d_i \in L_i$), it is rooted at a handled construct $\sigma_i \in langConstructs^{HC}(L_i)$ and σ_n can only be an element COC or SMC handled construct.*

Proof: According to the valid documents definition, d in $\mathcal{V}_{\mathcal{L}_d}^{Im_X}$ if and only if there is a sequence of $n \in [1, 2]$ valid documents $d_1, \ldots, d_2$, so that $d \in d_n \underset{Im_X}{\overset{+}{\leftarrow}} \cdots \underset{Im_X}{\overset{+}{\leftarrow}} d_1$,[2] where d_n is rooted at an element *COC* or *SMC* handled construct, $d_n \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ for $n > 1$, and $\forall i \in [1, \min(n-1, 1)]$ d_i is a valid single namespace document of L_i. The recursive application of the valid documents definition resolves recursive references to valid mixed namespace documents. Specifically, d in $\mathcal{V}_{\mathcal{L}_d}^{Im_X}$ if and only if there is a sequence of $n \in [1, \infty)$ valid documents $d_1, \ldots, d_n$, so that $d \in d_n \underset{Im_X}{\overset{+}{\leftarrow}} \cdots \underset{Im_X}{\overset{+}{\leftarrow}} d_1$, where d_n is rooted at an element *COC* or *SMC* handled construct, $d_n \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ for $n > 1$, and $\forall i \in [1, \min(n-1, 1)]$ d_i is a valid single namespace document of L_i. If n equals the number of single namespace subtrees in d, then d_n is also a single namespace document. Therefore, each valid document can be composed out of n single namespace subtrees. Inversely, the composition of n single namespace subtrees results to a valid document. Therefore, Corollary 2 is correct.

Proposition 1 maps the original definition of valid documents to a definition of documents with well defined processing. It assists the subsequent transformation discussion, because proving the feasibility of document transformation is subproblem of proving the feasibility of their well defined processing.

[2] The valid tree composition notation $\underset{}{\overset{+}{\leftarrow}}$ has been introduced in the integration model chapter, in page 85.

Proposition 1 *All valid XMLPipe documents have well defined processing if*

- *d has well defined processing* $\forall d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$, *where* $\mathcal{L}_d = \{L\}$ *and d is rooted at* $\sigma \in langConstructs_e^{HC}(L) \cap (COC(L) \cup SMC(L))$.
- *All documents in* $d_1 \xleftarrow[Im_X]{+} d_2$ *have a well defined processing, if* $d_1 \in \mathcal{V}_{\mathcal{L}_{d_1}}^{Im_X}$, $\mathcal{L}_{d_2} = \{L_2\}$, d_2 *is rooted at* $\sigma_2 \in langConstructs^{HC}(L_2)$ *and* d_1 *has well defined processing.*

Proof: In order to prove Proposition 1, it is sufficient to prove that, if both of its conditions are true, the processing of a valid document is well defined. Consider that $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ is a valid document and that both proposition conditions are true.

If d is a single namespace document, then $\mathcal{L}_d = \{L\}$ and d is rooted at $\sigma \in langConstructs_e^{HC}(L) \cap (COC(L) \cup SMC(L))$, according to the valid documents definition. Therefore, its processing is well defined, according to the first proposition condition.

If d is not a single namespace document, then $d \in d_1 \xleftarrow[Im_X]{+} d_2$, where $d_1 \in \mathcal{V}_{\mathcal{L}_{d_1}}^{Im_X}$ and $d_1 \in L_2$ where $\mathcal{L}_{d_2} = \{L_2\}$. According to the second proposition condition, d has well defined processing. Therefore, the above proposition is correct.

7.4.2 Transformation of valid documents

In order to prove the feasibility of mixed namespace document transformation, it is sufficient to prove that for all valid documents $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$ and adaptation profiles $pr \in Profiles$, there is a well defined transformation $T \in \mathcal{T}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}^{\mathcal{L}_p:Im_X}$, where $d \xrightarrow{T} d'$ and d' is the most adequate interpretation of d, according to the profile pr. The output of T uses the XMLPipe integration model, as opposed to the presentation integration model, because they are equivalent, according to Assumption 7. The remainder of this section will use Proposition 1 to prove the existence of a finite T, under the transformation assumptions summarised in Table 7.2.

Proposition 2 *Under the assumptions of Table 7.2, all valid XMLPipe documents d can be transformed by a finite iterative transformation* $T \in \mathcal{T}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}^{\mathcal{L}_p:Im_X}$ *to their most adequate representation d′, according to an adaptation profile pr.*

Proof: Proposition 2 can be proven by separately considering whether d uses natively or non-natively supported constructs and whether d is single or mixed namespace document.

If d is a valid document that solely consists of natively supported constructs ($d \in \mathcal{V}_{\mathcal{L}_p}^{Im_X}$), then it is a valid document of the presentation integration model and it does not require further XMLPipe processing, according to assumptions 6 and 7. Therefore, T is the identity transformation T^ϵ, where $d \xrightarrow{T^\epsilon} d$.

The transformation of each single namespace valid document is well defined. Specifically, consider the case where d is not natively presentable ($d \notin \mathcal{V}_{\mathcal{L}_p}^{Im_X}$) and d is a single namespace document ($\mathcal{L}_d = \{L\}$), which is rooted at an element COC or SMC handled construct $\sigma \in (COC(L) \cup SMC(L)) \cap langConstructs_e^{HC}(L)$. According to assumptions 1, 5 and 8, there is a non identity transformation T' that is adequate for pr and can transform d to a valid XMLPipe representation d_1. T' is not an identity transformation, because if $d_1 = d$ then T' would introduce a circular dependency with itself, which is not allowed by Assumption 5. Consequently, a valid single namespace document either is in its most adequate representation or there is a non-identity transformation that maps it to another valid XMLPipe document.

For the mixed namespace document case, consider a valid document $d_1 \in \mathcal{V}_{\mathcal{L}_{d_1}}^{Im_X}$ and a valid single namespace document d_2, where $\mathcal{L}_{d_2} = \{L_2\}$ and d_2 is rooted at a handled construct $\sigma_2 \in langConstructs^{HC}(L_2)$. Additionally, consider that d_1 is either natively presentable or there is a corresponding non-identity transformation T_1 that maps it to its valid interpretation d_1'.

If d_2 can be natively presented, the transformation of all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ is well defined. Specifically, consider that d_2 can be natively presented. If d_1 can also be natively presented ($\mathcal{L}_{d_1} \subseteq \mathcal{L}_p$), all documents $d_3 \in d_1 \xleftarrow[Im_X]{+} d_2$ will be valid documents that only contain natively presentable constructs.

Otherwise, if d_1 is not in its presentable form, there is a transformation T_1 that can process all documents in $d_3 \in d_1 \xleftarrow[Im_X]{+} d_2$. Specifically, according to Assumption 3, T_1 must copy all foreign namespace content to its corresponding place, within its output. Therefore, if $d_3 \xrightarrow{T'} d_3'$, d_3' is a valid XMLPipe document, which contains d_2 at a place that corresponds to its initial placement in d_3. Consequently, if d_2 can be natively presented and both d_1 and d_2 are valid documents, all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ either are natively presentable or have a well defined a non-identity transformation that maps them to other valid XMLPipe documents.

If d_2 cannot be natively presented and its processing does not relate to its context, the transformation of all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ is also well defined. Consider that d_2 contains non natively presented constructs and its root handled construct is an element COC or SMC ($\sigma_2 \in (COC(L_2) \cup SMC(L_2)) \cap langConstructs_e^{HC}(L_2)$). According to the above single namespace document discussion, there must be a non-identity transformation T_2 that maps d_2 to a valid document d_2': $d_2 \xrightarrow{T_2} d_2'$. If d_2' replaces d_2 within any $d_3 \in d_1 \xleftarrow[Im_X]{+} d_2$, the resulting document d_3' will be valid, according to assumptions 9 and 10. Specifically, if σ_2 is a COC, d_2 can only occur at at a place where content is expected, because d_3 is valid. Therefore, d_2' can also occur at the same place within d_1, because it is rooted at a handled construct. If d_2 is rooted at an SMC construct, d_2' will only be rooted at a COC construct, if d_2 occurs at a place where content is expected. Therefore, d_2' can always occur at the same place as d_2. Consequently, if $\sigma_2 \in COC(L_2) \cup SMC(L_2)$, there is a non-identity transformation that can transform all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ to valid XMLPipe documents.

Furthermore, if d_2 cannot be natively presented and its processing relates to its context, the transformation of all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ is well defined. Specifically,

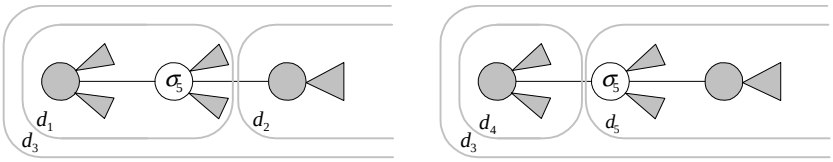

Figure 7.1: Tree separation illustration

consider all documents $d_3 \in d_1 \xleftarrow[Im_X]{+} d_2$, where d_2 is placed within an element σ_5 of d_1. Additionally, consider the documents d_4 and d_5, where d_4 is d_1 without the σ_5 rooted subtree and d_5 consists of σ_5, its descendants and d_2. Figure 7.1 illustrates the relationship between d_3, d_4 and d_5. There must be a transformation T_2 that can process d_2, within the context of its parent element, according to assumptions 1, 2 and 4. Consequently, T_2 can process d_5, so that $d_5 \xrightarrow{T_2} d'_5$. According to assumption 11, replacing d_5 with d'_5 results in a valid document. Therefore, if $\sigma_2 \in FOC(L_2)$ or $\sigma_2 \in langConstructs_a^{HC}(L_2)$, there is a well defined non-identity transformation that maps d_3 to another valid document

The composition of all the above cases proves Proposition 2. Specifically, all single namespace valid documents are either directly presentable or can be transformed by a non-identity transformation. Moreover, if both d_1 and d_2 are valid, d_2 is a single namespace document and d_1 is either directly presentable or can be transformed by a non-identity transformation, then all documents in $d_1 \xleftarrow[Im_X]{+} d_2$ are either directly presentable or can be transformed by a non-identity transformation. Consequently, every valid XMLPipe document either is directly presentable or there is a non-identity transformation that maps it to another valid document, according to Proposition 1. According to Assumption 5, there are no circular dependencies, between the individual transformation specifications. Therefore, a finite iterative transformation T can transform a valid XMLPipe document to its most adequate representation, according to a profile pr.

7.4.3 The transformation algorithm

Proposition 2 establishes the existence of an adequate transformation T for a mixed namespace document d, but it does not define how to transform d. Nevertheless, the above proof of Proposition 2 is a declarative construction of T. This section will use the above proof as its foundation, in order to establish the various document transformation aspects: the separation of d into its individual subtrees, the independent application of the language transformations to each subtree and the combination of the transformation outputs into the output document d'. Subsequently, it will describe the resulting XMLPipe transformation algorithm.

A subtree separation process that individually processes each handled construct subtree is adequate for the XMLPipe transformation process. Specifically, the language specific transformations, for a language L, can process document subtrees that are rooted at either a handled construct σ of L, if σ is a *COC* or an *SMC* element construct, or an ancestor of a handled construct, otherwise. The proof of Proposition 2 performed only the minimum necessary separations, at the inter-language boundaries

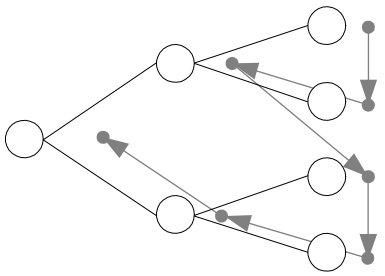

Figure 7.2: Post order tree traversal

of a document. However, a subtree separation process that individually processes each handled construct subtree is beneficial. Such a process can impede the use of intra-language relationships between handled constructs, because it separates all handled construct subtrees (even if they belong to the same namespace as their context). Nevertheless, the lack of such relationships ensures the uniform and language independent processing of all handled constructs, independently of their corresponding language. Consequently, the XMLPipe subtree separation process creates a separate subtree for each document handled construct, in a similar manner to the process introduced in Section 5.3.3 (page 88).

The individual subtrees of a document d must be processed in a postorder manner, illustrated in Figure 7.2, because the proof of Proposition 2 transformed each document subtree before its context. Such a transformation order is necessary for preserving the context of handled constructs and for ensuring that all necessary content is available, prior to a transformation. Specifically, the transformation of *FOC* rooted subtrees can relate to their context, within the source document. If a *FOC* rooted subtree is transformed after its ancestors, its relationship with its context may be harmed by the transformation of its ancestors. Additionally, the transformation of *SMC* rooted subtrees must precede the transformation of their context, because they can introduce document modifications that are necessary for transforming their context. For instance, consider the `alt:alt` and `imp:imp` rooted subtrees, in lines 2 and 10 of the driving example. Both subtrees must be processed before `doc:document`, because they introduce the document title and author information, which is necessary for transforming the `doc:document` rooted subtree. Consequently, XMLPipe processes the individual subtrees in a postorder manner.

An iterative transformation that processes a document's subtrees, in a postorder manner, can produce its optimal interpretation in a finite number of iterations. Specifically, consider an iterative postorder separation process that replaces all subtrees with the output of their corresponding transformations, until all document constructs belong to languages in $\mathcal{L}_p$. As described in the proof of Proposition 2, replacing a subtree with the output of its corresponding transformation results in a valid document, which is an interpretation of the original presentation information. Consequently, each iteration will result in a document interpretation that is closer to the document's optimal presentation, according to a profile pr. Since there are no circular transformation dependencies, such an iterative process will result to a natively presentable document interpretation, within a finite number of steps.

XMLPipe mixed namespace transformation:
The function *transform* : $D \times Profiles \rightarrow D$ represents the transformation $T \in \mathcal{T}^{\mathcal{L}_p:Im_X}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}$, which maps a valid XMLPipe document d to its most appropriate representation d', according to a profile pr.

function *transform*(D *doc*, *Profiles pr*) $\rightarrow D$
  let $d' = d$
  let n be the first node of d', according to a postorder tree traversal
  while (true)
    let σ be the XML construct that corresponds to n
    let L be the language that corresponds to σ
    let n' be the the next postorder tree traversal node after n
    if $L \notin \mathcal{L}_p$ AND $\sigma \in langConstructs^{HC}(L)$
      if T' is the optimal transformation for L, according to profile pr
        if $\sigma \in (COC(L) \cup SMC(L)) \cap langConstructs^{HC}_e(L)$
          Separate d_1 from d', where d_1 is rooted at n.
        else
          Separate d_1 from d', where d_1 is rooted at the parent of n.
        end if
        apply T' to d_1: $d_1 \xrightarrow{T'} d'_1$
        if d'_1 is not an empty tree
          replace d_1 with d'_1, within d'
          let n be the first node of d'_1 according to a postorder traversal
        else
          let $n = n'$
        end if
      else *//there is no appropriate transformation*
        the transformation **fails**; **exit**
      end if
    else *//the n rooted subtree does not require separate processing*
      if n is the root of d'
        if all d' nodes belong to languages in $\mathcal{L}_p$
          the transformation is **successful**; **exit**
        else
          the transformation **fails**; **exit**
        end if
      else
        let $n = n'$
      end if
    end if
  end while
end function

Figure 7.3: The XMLPipe transformation algorithm

Figure 7.3 illustrates the XMLPipe transformation algorithm (*transform* function), which is such an iterative process. Specifically, the main algorithm loop performs a postorder search for non-natively supported handled constructs. For each such construct, it uses the XMLPipe binding model to locate the optimal corresponding transformation. *transform* fails if there is no adequate transformation, because it cannot map the subtree to a natively supported representation. Otherwise, *transform* separates the appropriate subtree, applies the corresponding transformation and uses its output to replace the initial subtree. If the generated subtree is not empty, the postorder search proceeds into the newly introduced nodes, because they may require further preprocessing.

transform terminates after traversing all document nodes. If all the resulting constructs are natively supported, *transform* considers d' as the optimal interpretation of d, and it terminates successfully. If there are remaining non-natively supported constructs, *transform* fails, because either d is invalid or the used transformation specifications do not comply with the assumptions of Table 7.2. In both cases, the algorithm terminates in a finite number of steps, because there are no circular dependencies between the transformation specifications.

7.5 Transformation semantics

A generic document transformation process must use an optimal transformation specification for each document subtree, but *transform* does not define the details of such a process. This section introduces the necessary transformation semantics, for performing such a selection. Moreover, it investigates their binding and the necessary interoperation between the transformation model and the binding and adaptation models, which allows the selection of the optimal subtree transformation.

The specification of subtree transformations by transformation pipelines is beneficial, because pipelines integrate and enhance functionality of the existing transformation technologies. Consequently, XMLPipe semantics describe all transformations using transformation pipeline specifications. *PipeSpec* will represent the set of all XMLPipe transformation semantics specifications, and it will be described in the subsequent transformation pipelines section (Section 7.7).

Set of all pipeline specifications (*PipeSpec*): The set *PipeSpec* contains all XMLPipe pipeline specifications

The pipeline specifications can be associated with language URIs, but a more fine grained association that is based on handled constructs is beneficial. Specifically, for each language L, its corresponding transformations must define the processing of all its valid documents, according to Assumption 2. Therefore, it is sufficient to associate the namespace URI of L with its corresponding pipeline specifications. However, their association with the individual language handled constructs assists the modular specification of transformation semantics, because the individual constructs of a language can require significantly different processing. For instance, consider a transformation that is adequate for XHTML browsers and is associated with language L_{doc} (introduced in the driving example). A declarative transformation technology, such as XSL-T, can straightforwardly map the L_{doc} constructs to their corresponding XHTML representation. However, if the semantics author also wishes to adapt the `doc:img` referenced images to an adequate representation for the target browser, an imperative

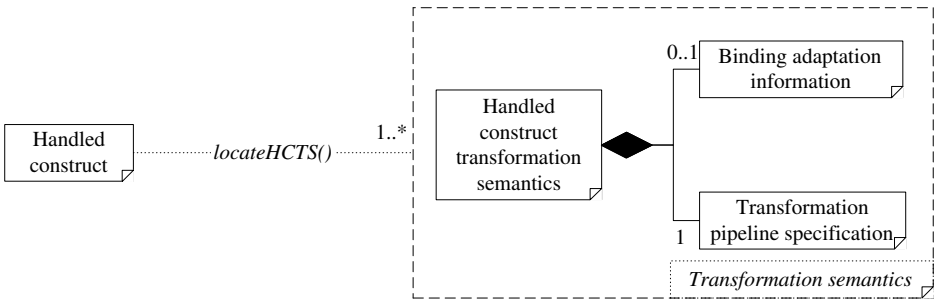

Figure 7.4: Handled construct transformation semantics

transformation specification is necessary, such as a Java based DOM manipulator. If transformation specifications are associated with the individual handled constructs, the transformation semantics of L_{doc} can be composed out of an imperative and a declarative specification, for the `doc:img` rooted subtrees and the remaining L_{doc} constructs, respectively. Therefore, the XMLPipe transformation specifications are associated with each individual handled construct.

The XMLPipe adaptation model established the *binding adaptation specification*, which provides the necessary information for choosing the optimal transformation specification, for an adaptation profile. The *adaptation measure evaluator* enables the selection of the optimal transformation alternative, because it maps each binding adaptation specification to an absolute comparable measure. Therefore, the XML transformation semantics contain an optional binding adaptation specification, in addition to the transformation pipeline specification. The binding adaptation specification is optional, because not all transformation specifications depend on the adaptation requirements. For instance, the processing of most *SMC* constructs, such as the `imp:import` handled construct, is adaptation requirements independent.

Figure 7.4 outlines the organisation of the XMLPipe transformation semantics. Specifically, each handled construct is associated with one or more transformation semantics alternatives. Each alternative consists of a transformation pipeline specification and an optional binding adaptation specification. *HCTSemantics* will represent the set of all XMLPipe transformation semantics.

Transformation semantics (*HCTSemantics*): The set *HCTSemantics* contains all the transformation semantics, which consist of an optional binding adaptation specification and a pipeline specification:

$$HCTSemantics = ((\mathcal{B} \cup \{\epsilon\}) \times PipeSpec)$$

The adaptation and binding models enables the optimal transformation location and selection. Specifically, the XMLPipe binding model is responsible for retrieving all alternative transformation semantics that correspond to a handled construct. Function *locateHCTS*() will map a handled construct to its corresponding transformation specifications, and it is a part of the binding component interface. The process of selecting the optimal specification can use the adaptation model *measure* function, in

order to retrieve a comparable adaptation measure for each alternative. The optimal transformation alternative is the transformation that is associated to either the maximum adequacy measure or no binding adaptation specification, if there are no other adequate pipeline specifications. The latter case accommodates adaptation requirement independent specifications, which do not require an associated binding adaptation specification. Function *bestHCTS* uses *locateHCTS* and *measure* to return the optimal pipeline specification for a pair of a handled construct and an adaptation profile.

Transformation semantics location function (*locateHCTS*): The handled construct transformation semantics location function $locateHCTS : \Sigma \rightarrow \wp(HCTSemantics)$ maps a qualified term to its corresponding semantics: $\forall \sigma \in \Sigma, locateHCTS(\sigma) = \{hcts_1, hcts_2, \ldots, hcts_n\}$, where $\forall i$, $hcts_i$ is an alternative transformation semantics for σ.

Optimal pipeline selection function (*bestHCTS*): The optimal pipeline selection function $bestHCTS : \Sigma \times Profiles \rightarrow PipeSpec$ maps a pair of a handled construct and an adaptation profile to their corresponding optimal pipeline specification. If $locateHCTS(\sigma) = \{(B_1, ps_1), \ldots, (B_n, ps_n)\}$ then $bestHCTS(\sigma, pr) = ps_k$, where

$$measure(pr, B_k) = \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) \neq 0$$

OR

$$B_k = \epsilon, \text{ if } \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) = 0$$

7.6 Addressing the assumption constraints

The *transform* and semantics definitions allow the transformation of presentation documents only when the Table 7.2 assumptions apply. However, a subset of these assumptions are sufficiently restrictive to prohibit most XMLPipe practical applications. For instance, they require that the presentation integration model is equivalent to the XMLPipe integration model, but no existing browser supports such a model. Moreover, they do not allow the processing of semantically correct but invalid documents, such as the driving example, because they require that the input and output of all transformations is valid. Such assumptions were necessary for proving the feasibility of a mixed namespace transformation process, but the proposed algorithm has a significantly wider applicability. This section describes an investigation of the assumptions, within the context of the proposed algorithm, and proposes a looser set of assumptions and a corresponding set of algorithm modifications.

7.6.1 Subtree copying

The subtree transformations must copy all foreign namespace content to its corresponding place within their output, according to Assumption 3. Existing transformation technologies provide straightforward ways to copy XML subtrees that occur at well defined places within a document. However, the development of most transformation specifications that copy *all* foreign namespace subtrees, which can occur at *any* document position, is either impossible or prohibitively complex. Nevertheless, for the majority of XML languages, it is is sufficient to only copy foreign namespace subtrees at well defined places.

Copying foreign namespace subtrees only at places where content is expected is sufficient, in the majority of cases. Specifically, the assumed subtree copying is necessary for preserving the presentation information that is introduced by the individual subtrees. If a document subtree introduces presentable information, it must be either a *COC* rooted subtree or an *SMC* rooted subtree that its processing results in a *COC* rooted subtree. Both can validly occur only at places where content is expected, according to Assumption 10. *FOC* constructs amend the presentation of their context; consequently, most *FOC* rooted subtrees occur as children of *COC*, because *COC* explicitly introduce presentable content. Furthermore, most *COC* constructs, such as the driving example's `doc:em` and `doc:p` elements, expect arbitrary content as their children. Consequently, copying foreign namespace subtrees at places where content is expected is sufficient for preserving all presentable pieces of information and the majority of presentation customisations introduced by *FOC* rooted subtrees.

If a *FOC* rooted subtree occurs at a place where content is not expected, such a reduced subtree copying scheme will not result in presentable information loss, but it may result in reduced presentation functionality. Specifically, if the transformation of a *FOC* rooted subtree adds content to its parent element to amend its presentation and it occurs at a place where content is not expected, the processing of its ancestors will ignore the introduced content. The resulting presentation might not be optimal, but it will not miss any presentable information, because *FOC* constructs only amend existing presentable information. Alternatively, the transformation of a *FOC* rooted subtree can introduce new context to its parent. The new context must be rooted at a handled construct σ, which can be a *COC*, a *FOC* or an *SMC*. The first case can only happen, if σ occurs at a place where content is expected, according to Assumption 11. Consequently, the transformation result will be preserved by the processing of its context. If the introduced context is a *FOC* subtree, the new subtree only enhances the functionality its context; therefore, there will be no loss of presentable information, if it is not copied. Thirdly, if the introduced context is an *SMC* rooted subtree, its iterative processing will result in one of the first two cases.

Consequently, transformation specifications that copy all foreign namespace subtrees are beneficial. However, for most integration cases it is sufficient to only copy foreign namespace subtrees that occur at places where content is expected. When such copying is not sufficient the transformed document may not be optimal, but it does not miss any presentable information. Therefore, a looser version of Assumption 3 is beneficial, because it significantly simplifies the development of the transformation specifications. The revised Assumption 3 is that each transformation must copy, at a minimum, all foreign namespace subtrees that occur at places where content is expected to their corresponding place within its output.

7.6.2 Transformation of semantically correct invalid documents

There are two cases of semantically correct but invalid documents. The first case is when the processing of an *SMC* construct introduces content that is required by its ancestors. For instance, consider the `alt:alt` and `imp:import` constructs, in lines 2 and 10 of the driving example. Their transformation introduces the necessary document title and author information. However, the document is invalid prior to their processing, because the `doc:document` does not contain the required `doc:title` and `doc:author` elements. The second case of semantically correct but invalid documents is when either the ancestors or the descendants of an *SMC* are invalid as independent

subtrees, but become valid after its transformation. For instance, the `alt:alt` rooted subtree, it line 2 of the driving example, is an invalid document subtree, because the `doc:title` element is not a handled construct and it cannot validly occur under a foreign namespace element. The transformation of the `alt:alt` rooted subtree substitutes itself with one of the alternative `doc:title` elements, which can validly occur under the `doc:document`.

The design of the proposed transformation algorithm relied on the assumption that the input document is valid, according to the XMLPipe integration model. However, it can process semantically correct but invalid documents, without any modifications. Specifically, it successfully addresses the first case, because its postorder document traversal ensures that each subtree is processed after all its descendant *SMC* rooted subtrees. For instance, *transform* processes the `doc:document` rooted subtree after processing the `imp:import` construct, which imports the necessary author information. *transform* also addresses the second case of invalid semantically correct documents, because it requires no inter-language interoperation. Each language transformation must only copy foreign namespace subtrees and not attempt to interpret them. Therefore, the transformation of an *SMC* rooted subtree can copy the foreign namespace subtrees, without checking their validity. Furthermore, the postorder transformation order ensures that the necessary modifications of any invalid ancestor subtrees, will have been made prior to their processing. For instance, the `alt:alt` construct transformation can copy the `doc:title` elements to its output, independently of their validity. After its transformation, its previously invalid context (the `doc:document` rooted tree) becomes valid, because it contains the necessary document title information. Therefore, the defined *transform* function is adequate for semantically correct but invalid documents.

7.6.3 Circular transformation dependencies

Assumption 5 prohibits circular transformation relationships, but the processing of several languages can benefit from such relationships. Specifically, the termination of *transform* is guaranteed only when there are no circular dependencies, which can lead to infinite transformation loops. For instance, consider the L_{imp} language. The transformation of the `imp:import` construct can potentially introduce a circular dependency with itself, if an `imp:import` element imports a document that contains further import statements. If a document contains an `imp:import` element that imports the same document, the *transform* function will iterate indefinitely. L_{imp} semantics must be constrained to only allow single level content inclusion, in order to comply with Assumption 5. Such a restriction would guarantee the transformation termination, but it would significantly reduce its functionality.

Circular transformation relationships can be allowed, if there is a well defined method to identify and terminate any infinite transformation loops. A set of predefined interfaces/protocols could allow sufficient interoperation, between the individual transformations, to enable the identification of such loops. However predefined interfaces and/or protocols are not adequate for an open set of languages. An alternative method, which does not require inter-language interoperation, is to allow the interoperation of separate transformation invocations that correspond to the same language. Such interoperation is sufficient, because each infinite transformation loop must result in a periodic application of the same transformation, since there can only be a finite number of transformations. Therefore, the XMLPipe transformation model can allow

circular transformation relationships, if it enables the interoperation between transformations instances that correspond to the same language.

The XMLPipe transformation model utilises its internal adaptation profile representation to achieve the necessary interoperation, because the adaptation requirements information is available to all transformation components. Specifically, each language L can have a set of *language specific terms*, which are adaptation terms that belong to the same namespace as L. A transformation can introduce adaptation statements that use language specific terms, which correspond to the same language as the transformation. For each transformation T that processes a document subtree, the XMLPipe transformation model provides all relevant language specific statements: statements that correspond to the same language and have been introduced by the ancestors of its input subtree ancestors. This mechanism allows a transformer to pass information to all transformers that process its output and correspond to the same language. Each transformer can use the provided information to identify and terminate any infinite transformation loops. Additionally, this mechanism prohibits the interoperation between unrelated transformer instances, such as transformers that correspond to separate languages or to unrelated document subtrees.

For instance, a transformation T that corresponds to the L_{imp} language can use language specific terms to detect the recursive inclusion of the same document. Specifically, each transformation instance can add the URI of the imported document to a language specific statement. If an `imp:import` construct imports a document that has already been imported by its ancestors, the language specific statement will already include the corresponding document URI. The transformer can avoid the infinite inclusion loop by either terminating the transformation or ignoring the import construct.

Consequently, Assumption 5 is not necessary, if the transformation algorithm contains the necessary functionality for propagating the language specific term information. In contrast, it can be substituted by an alternative assumption: if a transformation can introduce a circular dependency, it must also detect and terminate any infinite transformation loops.

7.6.4 Processing natively supported constructs

Assumption 6 prohibits the generation of $\mathcal{L}_p$ constructs that require additional preprocessing, in order to ensure the sufficiency of a preprocessing approach. However, the preprocessing of natively supported constructs can be beneficial. For instance, the preprocessing of a natively supported construct for document inclusion can allow the preprocessing of any included content. Moreover, document preprocessing can adapt the binary representation of XHTML referenced images that are not supported by a target browser, in a similar manner to the `doc:img` processing, which was described in Section 7.5.

transform can enable the preprocessing of natively supported constructs, after the introduction of two minor modifications. The first "*if*" condition must no longer test whether a construct is non natively supported, but whether it has an associated transformation. Furthermore, it is beneficial that *transform* traverses the output of a transformation, only when it differs from the transformation's input. Specifically, the transformation of a natively supported construct can either modify it or leave it unchanged, since it can be natively presented. Such a transformation introduces a circular dependency with itself, and must use the necessary intra-language interoperation to avoid

infinite loops. However, since identity transformations are common in such transformations, the additional subtree traversal check can significantly simplify the development of the transformation specifications.

The introduced *transform* modifications allow the substitution of Assumption 6 with a looser alternative. Specifically, a transformation can introduce natively supported constructs that require further preprocessing, if there is an adequate transformation that can preprocess them.

7.6.5 Integration models equivalence

The final assumption that must be addressed is the equivalence of the XMLPipe and the presentation integration models (Assumption 7). This assumption was necessary, because the XMLPipe transformation model cannot be aware of all presentation integration models. However, such an assumption impedes all XMLPipe practical applications, because no existing browser supports the XMLPipe integration model.

However, adequate design of the transformation specifications can allow the document preprocessing for presentation integration models that are less generic than the XMLPipe integration model. Specifically, the postorder processing order ensures that each transformation only processes constructs of its corresponding language and of natively supported languages. Therefore, a transformation can integrate its output constructs with the previously processed subtrees in a valid way, according to the target presentation integration model. For instance, if a presentation integration model requires a connecting element between an element and its foreign namespace children, the individual transformations can add such connecting constructs, before copying the foreign namespace subtrees. Additionally, if the target browser does not support any integration model, a transformation can either output the same language as its subtrees or discard any foreign namespace subtrees. The latter can result in information loss, but it allows a partial presentation of an otherwise non presentable document.

If all transformation specifications enforce the target integration model, Assumption 7 can be loosened to only require that the presentation integration model is less generic than the XMLPipe integration model.

7.6.6 Alternative assumptions and transformation algorithm

Table 7.3 summarises the revised set of transformation assumptions, according to the aforementioned discussion. A subset of these assumptions are the transformation specification design principles. Specifically, specifications that are adequate for subtree transformations within a mixed namespace document must follow assumptions 2, 3 and 4. Furthermore, transformations that preserve the semantic validity of documents, must follow assumptions 9, 10 and 11.

Figure 7.5 illustrates the revised transformation algorithm, which includes the necessary modifications for the revised set of assumptions. Specifically, it no longer requires that all processed handled constructs are not natively supported. Additionally, it only traverses a transformation's output when it differs from its input, in order to assist the development of transformations that process natively supported constructs. Moreover, it uses the *bestHCTS* function to precisely define the transformation selection process.

Most algorithm modifications are responsible for the proposed intra-language interoperation, which allows the identification of infinite transformation loops. Specifically, consider a transformation T' that is associated with a language L. If T' transforms a

ID	Assumption
1	There is an adequate transformation specification for each language not in $\mathcal{L}_p$.
2	Transformation specifications can process all valid handled construct rooted trees of their corresponding language.
3	Each transformation *must copy at least the foreign namespace subtrees that occur at places where content is expected* to their corresponding place within the transformation output, *and it must enforce the presentation integration model.*
4	The transformation specifications that correspond to either *FOC* or attribute handled constructs must be able to process them within the context of a foreign namespace element.
5	Circular dependencies between transformation specifications *can occur, if they ensure that their recursive application does not result in an infinite transformation loop.*
6	The output of a transformation *can introduce an* $\mathcal{L}_p$ *construct* σ *that requires further preprocessing, if there are adequate transformation specifications for preprocessing* σ*.*
7	If $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_P}$ *then* $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$.
8	*If* T *corresponds to a non SMC construct of language* L*, its input and its output must be valid according to the XMLPipe integration model. In the case of SMC handled constructs the transformation input and output is not necessarily valid, but it must be semantically correct.*
9	A *COC* rooted subtree must be transformed to another *COC* rooted subtree.
10	The transformation of an *SMC* rooted subtree cannot produce a *COC* subtree, unless it occurs at a place where content is expected.
11	The processing of *FOC* and attribute handled constructs must preserve the semantic correctness of the document. Specifically, it must add ancestors to their parent element, only if • the parent element is a handled construct • it can occur at the corresponding place, within the added ancestors • the root of the new content is a handled construct, which can be a *COC* only if the parent element is a *COC*. Additionally, it can add content to their parent, only if • the added content is valid • it occurs at a place where content is expected, if it is rooted at a *COC* construct.

Table 7.3: Alternative transformation assumptions and design principles. The introduced modifications have been *emphasised*

Revised XMLPipe mixed namespace transformation:

function *transformRev*$(D\ d, Profiles\ pr) \rightarrow D$
  let $d' = d$
  let n be the first node of d', according to a postorder tree traversal
  while (true)
    let σ be the XML construct that corresponds to n
    let L be the language that corresponds to σ
    let n' be the next postorder tree traversal node after n
    if $\sigma \in langConstructs^{HC}(L)$
      if $bestHCTS(\sigma, pr) \neq \epsilon$
        if $\sigma \in (COC(L) \cup SMC(L)) \cap langConstructs_e^{HC}(L)$
          Separate d_1 from d', where d_1 is rooted at n.
        else
          Separate d_1 from d', where d_1 is rooted at the parent of n.
        end if
        let pr' contain all adaptation statements associated with L
        apply T' to d_1: $d_1 \xrightarrow{T'} d_1'$, using both pr and pr'
        let pr'' contain all L specific statements introduced by T'
        if $d_1' \neq d_1$ AND d_1' is not an empty tree
          associate pr'' with L and with the parent of n
          replace d_1 with d_1', within d'
          let n be the first node of d_1', according to a postorder traversal
        else
          if there is a language specific pr_1 associated to n' parent
            discard pr_1
          end if
          let $n = n'$
        end if
      else if $L \neq L'$ *//No appropriate transformation*
        the transformation **fails**; **exit**
      end if
    else *//the n rooted subtree does not require separate processing*
      if n the root of d'
        if all d' nodes belong to languages in $\mathcal{L}_p$
          the transformation is **successful**; **exit**
        else the transformation **fails**; **exit**
        end if
      else
        let $n = n'$
        if there is a language specific pr_1 associated with the parent of n'
          discard pr_1
  end if; **end if**; **end if**; **end while**
end function

Figure 7.5: Revised transformation algorithm

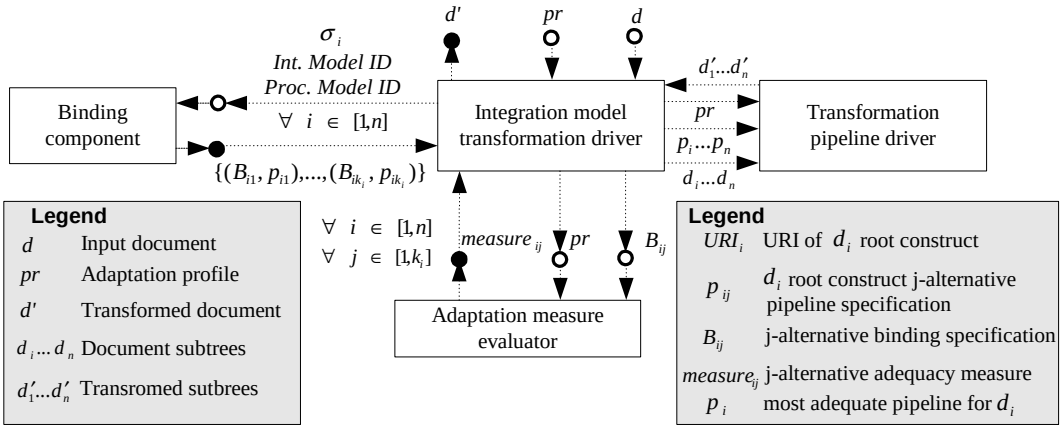

Figure 7.6: Integration model transformation driver

subtree that is rooted at a construct n and introduces adaptation statements that are specific to L, *transformRev* associates them with the parent of n. The adaptation statements are not associated with n, because the subsequent transformations can modify the n rooted subtree and may harm such associations. The proposed intra-language interoperation allows a transformation to only pass information to the transformations of its descendants. Therefore, language specific adaptation statements only apply to the descendants of n, and the postorder traversal removes all adaptation statement associations when moving up the tree.

The integration model transformation driver implements *transformRev*, in order to drive the transformation of a presentation document. Figure 7.6 illustrates the integration model transformation driver, its interface and its interoperation with the other processing components. Specifically, it converts an input document d to an output document d', according to an adaptation profile pr. The postorder subtree separation of d results in the $d_1 \dots d_n$ subtrees. For each subtree d_i, the integration model transformation driver provides the corresponding handled construct σ_i and the integration model identifier to the the binding component, in order to obtain the corresponding transformation specifications. The binding component returns a set of multiple pairs of transformation specifications and binding adaptation specifications. Subsequently, the adaptation measure evaluator is used, in order to choose the optimal transformation specification. Finally, the integration model transformation interacts with the transformation pipeline driver, which instantiates a pipeline transformation and applies it to d_i. After the substitution of d_i with the transformation output, the integration model transformation driver continues the postorder traversal, until it terminates.

7.7 Built-in transformation pipelines

The integration model transformation driver interoperates with the transformation pipeline driver, because the XMLPipe transformation semantics use pipeline specifications to define the processing of the individual handled constructs. The pipeline driver is responsible for instantiating the transformations that correspond to the pipeline

specifications and applying them to the document subtrees. This section describes the XMLPipe transformation pipelines and transformation pipeline driver.

7.7.1 Atomic transformations

Each transformation pipeline may consist of multiple *atomic transformations*, which allow the seamless integration of multiple transformation technologies. Specifically, the concept of *atomic transformations* enables the encapsulation of the individual transformations under a common interface. The atomic transformation interface can be implemented by both generic transformation technologies, such as XSL-T, or purpose built transformers, such as imperative DOM manipulators. Consequently, the XMLPipe transformation components can seamlessly use a variety of existing and future transformation technologies, if they access them through the common atomic transformation interface.

Atomic transformations must allow the transformation of document subtrees, in addition to complete XML documents, as described in Section 7.1. Such a requirement does not influence the common atomic transformation interface, since the structure of a document subtree is interchangeable with the structure of a document, because of the recursive nature of XML trees. However, document-specific operations must be avoided. For instance, generic transformation approaches provide document specific constructs, such as the XSL-T document root references. Such constructs are not well defined within the context of subtree processing. Such a constraint does not reduce the functionality of the individual transformations, because absolute constructs can be substituted by alternative relative constructs, which are adequate for subtree processing. Consequently, the semantics authors must not use document-specific constructs, in order to ensure the adequacy of the atomic transformations for the XMLPipe transformation model.

Atomic transformations do not produce an output subtree, as opposed to the majority of transformations. Specifically, an atomic transformation directly modifies its input, because *transformRev* transforms documents by substituting each subtree with the result of its corresponding transformation. Transformation implementations that are specific to XMLPipe can directly support such behaviour. However, existing transformation approaches produce an output document, instead of modifying their input. Consequently, atomic transformation wrappers are necessary for adapting the existing transformation functionality to the XMLPipe tree modifying behaviour.

Finally, atomic transformations must be able to both access the adaptation profile and introduce language specific adaptation statements. Adaptation profile access is essential for adaptive transformations, which customise their output according to the adaptation requirements. A straightforward way to introduce language specific adaptation statements is also essential, because it allows the necessary intra-language communication for identifying and terminating infinite transformation loops. In a similar manner to above, XMLPipe specific transformation implementations can straightforwardly access and modify the adaptation requirements. In contrast, the wrappers of existing transformation technologies must provide technology specific methods to access and introduce the adaptation statements.

The atomic transformation interface can consist of a single function. Each atomic transformation processes a document subtree, according to an adaptation profile and can optionally introduce a set of language specific statements. Therefore, a single two-parameter function that accepts a pair of a document subtree and an adaptation

profile and returns a list of adaptation statements is sufficient. Such an interface does not restrict the functionality of a transformation and does not require more transformation functionality than the transformation of document subtrees. Consequently, it is adequate for encapsulating the multitude of current and future transformation technologies. The pilot XMLPipe implementation, which will be described in Chapter 10, uses such a function for the atomic transformation interface.

However, we will define the atomic transformations as functions that map an adaptation profile to a transformation, in order to be consistent with the introduced transformation notation. Specifically, $\mathcal{A}$ will represent the set of all XMLPipe atomic transformations. Each atomic transformation $A \in \mathcal{A}$ is a function that maps an adaptation profile pr to a transformation, which transforms a document subtree to its corresponding transformation output. For each atomic transformation $A \in \mathcal{A}$ and adaptation profile pr, $d \xrightarrow[I]{A(pr)} d'$ represents the transformation of d to d', according to the adaptation profile pr and the external input I.

Atomic transformations ($\mathcal{A}$): The set of *atomic transformations* $\mathcal{A}$ contains all functions $A : \mathit{Profiles} \rightarrow \mathcal{T}^{\mathcal{L}:Im_X}_{\mathcal{L}:Im_X(L)}$.

7.7.2 Transformation wrappers

Atomic transformation wrappers are necessary for using existing transformation technologies. The atomic transformation interface allows direct modification of document subtrees, access to the adaptation requirements and introduction of language specific adaptation statements. Semantics authors can define atomic transformations by either directly implementing the atomic transformation interface or using an existing transformation technology, such as XSL-T. XMLPipe specific transformation implementations can directly access the atomic transformation functionality. However, atomic transformation wrappers are necessary for the integration of existing transformation technologies. Such wrappers are responsible for modifying the input subtree, according to the output of the underlying transformation technology, and for providing technology specific means to access the adaptation requirements and introduce language specific adaptation statements. Additionally, it is beneficial if the transformation wrapper developers map the XMLPipe transformation design principles, summarised in Table 7.3, to the corresponding technology specific design guidelines. Such design guidelines assist the semantics authors to create transformation specifications that are adequate for the XMLPipe, without being aware of the underlying XMLPipe transformation model.

For instance, consider an XSL-T atomic transformation wrapper, which uses an existing XSL-T implementation. The wrapper must pass its input subtree to the XSL-T transformer and replace it with the transformation's result. XSL-T stylesheet specifications do not include a mechanism for accessing sets of adaptation requirements. The wrapper can insert an adaptation profile representation to the transformation's input, using XMLPipe specific elements. An XSL-T stylesheet will be able to use XPath expressions to access the adaptation requirements information. A similar method can be used for the introduction of language specific adaptation statements. For instance, if the transformation output contains such XMLPipe specific elements, the wrapper can interpret them as language specific adaptation statements.

Table 7.4 describes the XSL-T design guidelines, which correspond to a subset of the Table 7.3 assumptions (2, 3 and 4). Specifically, each stylesheet must include top

Guideline	Description	Example
Multiple top-level templates (based on Assumption 2)	There must be a top-level XSL-T template for each handled construct that the transformation processes.	For language L_{doc}, there must be a top level template for the `doc:document`, `doc:em`, `doc:img` and `doc:p` constructs. For instance: `<xsl:template match=''doc:p''>`
Relative XPath expressions (based on Assumption 2)	XPath expressions must use relative paths instead of absolute document root references.	Use `*/doc:section` instead of `//*/doc:section`. Such changes might require stylesheet restructuring.
Copy unknown content (based on Assumption 3)	Include identity transformations, for all content, and null transformations for content that must not be copied to the output. Apply them, at a minimum, to places where content is expected.	Use `<xsl:apply-templates select=''*''/>` instead of `<xsl:apply-templates select=''doc:p''/>`
Copy unknown context (based on Assumption 4)	For attribute or *FOC* handled constructs, apply the copy templates at the root of the tree, instead of directly using the language constructs	Use `<xsl:apply-templates select=''*''/>`, instead of `<xsl:apply-templates select=''./@xl:href''/>` in combination with generic content copying templates.

Table 7.4: XSL-T transformation design guidelines

level transformation templates for all the handled constructs that it processes, in order to allow the processing of handled construct rooted subtrees. XSL-T uses XPath expressions for the selection of document node sets. XSL-T stylesheets must solely use relative XPath expressions, because absolute XPath expressions are not well defined, within the context of subtree processing. Additionally, all stylesheets must copy foreign namespace content that occurs at places where content is expected and at as many other places as possible. Stylesheets that correspond to *FOC* or attribute handled constructs must also copy their foreign namespace parent and its descendants, because the subtree separation process for *FOC* and attribute handled constructs includes their parent element. The semantics authors that use XSL-T must only be aware of the identified design principles, and they are not required to know the details of the XMLPipe transformation model.

7.7.3 Transformation pipelines composition

XMLPipe transformation pipelines are composed out of atomic transformations and validation commands. Existing alternative pipeline approaches consist of transformers, sources, mergers and sinks, as described in Section 7.1. However, only transformers are relevant to XMLPipe. Specifically, sources and sinks are not necessary, because the document input and output are a priori defined by the subtree separation process. In a similar manner, mergers are not necessary, because there is a single document input. The validation commands are necessary, because they allow the interoperation

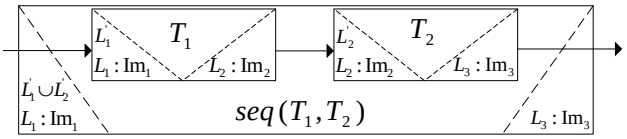

Figure 7.7: XMLPipe transformation pipelines: sequence pipeline

Sequence pipeline (*seq*): The sequence *seq* pipeline is a function

$$seq : \bigcup_{\forall \mathcal{L}_1, \mathcal{L}_1', \mathcal{L}_2, \mathcal{L}_2', \mathcal{L}_3, Im_1, Im_2} \mathcal{T}^{\mathcal{L}_2:Im_2}_{\mathcal{L}_1:Im_1(\mathcal{L}_1')} \times \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_2:Im_2(\mathcal{L}_2')} \rightarrow \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_1:Im_1(\mathcal{L}_1' \cup \mathcal{L}_2')}$$

where $d \xrightarrow{seq(T_1,T_2)} d'$ iff $d \xrightarrow{T_1} d_1$ and $d_1 \xrightarrow{T_2} d'$.

between the pipeline driver and the validation model. A pipeline based interoperation approach between the transformation and validation models is adequate for XMLPipe, as described in Section 7.1. Consequently, the XMLPipe transformation pipelines are solely composed out of atomic transformations and validation commands.

The XMLPipe pipeline composition uses a recursive application of three composition methods: transformation sequence, transformation selection and dynamic transformation. XMLPipe allows the recursive composition of transformation pipelines, because a transformation pipeline is also a transformation, as described in Section 2.3.2. Furthermore, it is beneficial, because it allows the arbitrary complex nesting of transformation pipelines.

The *transformation sequence pipeline* is a sequential composition of two transformations, where the second transformation is applied to the result of the first. Sequential composition is essential for the modular design of transformation specifications, because it allows their partitioning into sequences of simpler steps. Figure 7.7 illustrates the sequence pipeline. $seq(T_1, T_2)$ represents the sequential composition of T_1 and T_2: it firstly applies T_1 to the document input and subsequently applies T_2 to the output of the first transformation. The sequence pipeline input and output document types are identical to the input of T_1 and the output of T_2, respectively. A sequence pipeline only combines two transformations, but nested sequence pipelines can create arbitrarily long transformation chains. For instance, $seq(seq(T_1, T_2), T_3)$ is the sequential composition of T_1, T_2 and T_3.

The *transformation selection pipeline* enables the selection of alternative transformations, according to the adaptation requirements. A selection pipeline consists of a set of transformations that are associated with optional adaptation expressions. For each adaptation profile, the optimal transformation alternative can be chosen, in a similar manner to the transformation semantics selection by *bestHCTS* function. Figure 7.8 illustrates a transformation selection pipeline that chooses between three transformations: T_1, T_2 and T_3. For an adaptation profile pr, $sel((T_1, B_1), \ldots, (T_n, B_n))$ is equivalent to the optimal transformation of $\{T_1 \ldots T_n\}$, according to pr. The input

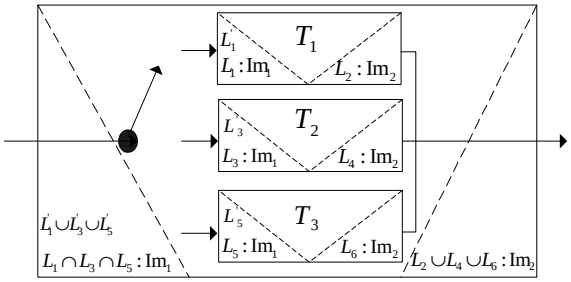

Figure 7.8: XMLPipe transformation pipelines: transformation selection

Selection pipeline (*seq*): The selection *sel* pipeline is a function

$$seq: \bigcup_{\substack{\forall n, \forall i \in [1,n], \\ \forall \mathcal{L}_{i_1}, \mathcal{L}_{i_1}', \mathcal{L}_{i_2}, Im_1, Im_2}} \left(\begin{array}{c} (\mathcal{T}^{\mathcal{L}_{1_2}:Im_{1_2}}_{\mathcal{L}_{1_1}:Im_{1_1}(\mathcal{L}_{1_1}')} \times (\mathcal{B} \cup \{\epsilon\})) \times \cdots \times \\ \times (\mathcal{T}^{\mathcal{L}_{n_2}:Im_{n_2}}_{\mathcal{L}_{n_1}:Im_{n_1}(\mathcal{L}_{n_1}')} \times (\mathcal{B} \cup \{\epsilon\})) \rightarrow \\ \rightarrow \mathcal{T}^{\mathcal{L}_{1_2} \cup \cdots \cup \mathcal{L}_{n_2}:Im_2}_{\mathcal{L}_{1_1} \cap \cdots \cap \mathcal{L}_{n_1}:Im_1(\mathcal{L}_1{}' \cup \cdots \cup \mathcal{L}_{n_1}{}')} \end{array} \right)$$

where for an adaptation profile pr, $sel((T_1, B_1), \ldots, (T_n, B_n)) = T_k$ where

$$measure(pr, B_k) = \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) \neq 0$$

OR

$$B_k = \epsilon, \text{ if } \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) = 0$$

and output document types of a selection pipeline depend on *all* the transformations, as opposed to the the sequential composition. Specifically, its input is the intersection of the inputs of its individual transformations, because the selection mechanism can choose any of the alternative transformations. In a similar manner, its output document type is the union of the output types of its individual transformations.

The *dynamic transformation pipeline* enables the customisation of transformations, according to dynamically produced information. Specifically, dynamic pipelines feed the output of a transformation to the external input of another transformation. If an atomic transformation uses its external input to acquire its transformation stylesheet, a dynamic transformation pipeline enables the dynamic generation of a transformation. Dynamic pipelines can significantly simplify some transformation specifications. For instance, consider the `imp:import` construct, illustrated in the driving example. The `select` attribute of the `imp:import` element specifies an XPath expression, which selects the imported nodes. XSL-T stylesheets do not allow dynamic XPath expressions. Therefore, an XSL-T stylesheet cannot reuse its built-in XPath engine, in order to perform the selection of the imported content. A dynamic pipeline enables the dynamic creation of an XSL-T stylesheet that statically contains the necessary XPath expression.

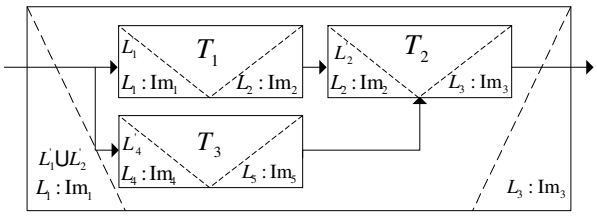

Figure 7.9: XMLPipe transformation pipelines: dynamic transformation

Dynamic pipeline (*dyn*): The dynamic pipeline *dyn* pipeline is a function

$$dyn : \bigcup_{\substack{\forall \mathcal{L}_1, \mathcal{L}_1', \mathcal{L}_2, \mathcal{L}_2', \\ \mathcal{L}_3, \mathcal{L}_4, \mathcal{L}_4', \mathcal{L}_5, Im_1, \\ Im_2, Im_3, Im_4, Im_5}} \mathcal{T}^{\mathcal{L}_2:Im_2}_{\mathcal{L}_1:Im_1(\mathcal{L}_1')} \times \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_2:Im_2(\mathcal{L}_2')} \times \mathcal{T}^{\mathcal{L}_5:Im_5}_{\mathcal{L}_4:Im_4(\mathcal{L}_4')} \rightarrow \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_1:Im_1(\mathcal{L}_1' \cup \mathcal{L}_2')}$$

where $d \xrightarrow{dyn(T_1,T_2,T_3)} d'$ iff $d \xrightarrow{T_1} d_1$ and $d \xrightarrow{T_3} d_2$ and $d_1 \xrightarrow[d_2]{T_2} d'$.

Figure 7.9 illustrates the dynamic transformation composition. $dyn(T_1, T_2, T_3)$ represents the application of T_2 to the output of T_1, according to the output of T_3. Specifically, the input document d is firstly transformed by T_1, which produces d_1: $d \xrightarrow{T_1} d_1$. Additionally, d is also transformed by T_3, which produces d_1': $d \xrightarrow{T_3} d_1'$. Finally, the application of T_2 to d_1, according to d_1', results in the output of the dynamic pipeline d': $d_1 \xrightarrow[d_1']{T_2} d'$. The output of T_3 only customises T_2, and it does not explicitly process the resulting document. Consequently, the input and the output of a dynamic transformation are the same as the input and the output of a sequential composition of T_1 and T_2.

The transformation pipeline driver is responsible for the interoperation between validation and transformation, because a pipeline based interoperation approach is adequate for XMLPipe, as described Section 7.1. Specifically, a semantics author would be aware of which transformation specifications require validated input, because semantics authors are the entities that define the individual transformation pipelines and atomic transformations. Consequently, XMLPipe transformation pipeline specifications allow the introduction of explicit validation steps, in addition to the atomic transformations. Such validation steps enable a semantics author to liberally introduce all the necessary validation processing.

Two types of pipeline initiated validation methods are necessary: the *complete subtree validation* and the *handled construct subtree validation*. The former tests the validity of all the nodes of a subtree, and it is necessary for ensuring the validity of a subtree, according to the XMLPipe validation model. The latter only tests the validity a subtree's top level nodes, which belong to the same namespace as its root element. Handled construct validation is essential for processing semantically correct but invalid

documents, because it allows the validation of only the necessary constructs for transforming a subtree. For instance, consider the driving example languages L_{doc} and L_{alt}. A transformation pipeline that is associated with L_{doc} may require a complete subtree validation step, in order to ensure the validity of the transformed subtrees. In contrast, a transformation pipeline that is associated with L_{alt} can use the handled construct subtree validation, in order to allow the transformation of semantically correct documents. Specifically, the contents of the `alt:case` element do not have to be valid, in a semantically correct document, as described in Section 7.2. The handled construct validation overcomes this problem by only testing the usage of the `alt:alt` and `alt:case` constructs, which are necessary for the application of the transformation.

Both validation methods can be used as atomic transformations, within a pipeline specification. Specifically, a subtree validation process can also be considered as a transformation process, because it process a subtree, introduces the necessary default attribute values and either returns successfully or fails. The XMLPipe pipeline specifications allow the use of two predefined constructs, which correspond to the two subtree validation methods, in the same places that they allow atomic transformations. The pipeline driver interoperates with the XMLPipe validation model, described in Chapter 8, in order to perform each validation step.

Summarising, an XMLPipe transformation pipeline can be an atomic transformation, a subtree validation step or their composition, using one of the tree pipeline composition methods. Section 7.5 introduced an abstract definition of XMLPipe transformation pipeline specifications, in order to allow the definition of the transformation semantics. The above description of the XMLPipe pipelines allow a more fine grained definition of transformation pipelines.

XMLPipe pipelines (*Pipelines*): The set of all XMLPipe pipelines *Pipelines* contains all transformations in $\mathcal{T}^{\mathcal{L}:Im_X}_{\mathcal{L}:L(Im_X)}$, that correspond to all atomic transformations, subtree validation constructs and pipeline compositions of pipelines. For a profile pr,

$$p \in Pipelines \text{ iff } \begin{cases} p = A(pr) & , A \in \mathcal{A} \\ p = \texttt{validateSubtree} & \\ p = \texttt{validateHC} & \\ p = seq(p_1, p_2) & , p_1, p_2 \in Pipelines \\ p = dyn(p_1, p_2, p_3) & , p_1, p_2, p_3 \in Pipelines \\ p = sel(p_1, B_1, \cdots, p_n, B_n) & , \forall i \in [1, n] p_i \in Pipelines, B_i \in \mathcal{B} \end{cases}$$

7.8 The complete transformation model

Figure 7.10 combines all introduced components, into the complete XMLPipe transformation model. Specifically, the integration model transformation driver drives the transformation process and implements the revised transformation algorithm, illustrated in Figure 7.5. For each document subtree $d_1 \dots d_n$, it interacts with the binding component, in order to retrieve the relevant pairs of transformation pipeline specifications and adaptation binding specifications $\{(B_{i1}, p_{i1}), \dots, (B_{ik_i}, p_{ik_i})\}$. After interacting with the adaptation measure evaluator and choosing the most adequate specification p_i, it passes the control to the transformation pipeline driver.

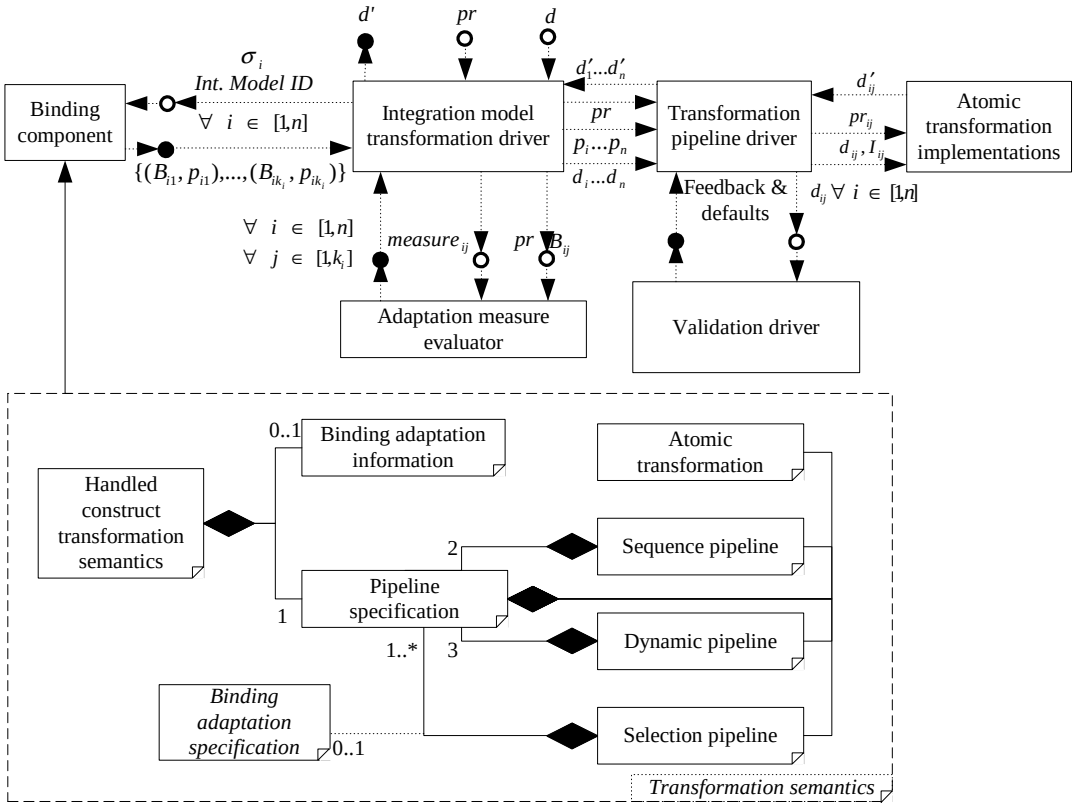

Figure 7.10: XMLPipe transformation model

The transformation pipeline driver is responsible for driving the transformation of the individual subtrees, according to the corresponding pipeline specifications. The pipeline driver must interoperate with the adaptation measure evaluator, in order to choose the optimal alternative transformation for the selection pipelines.[3] Additionally, the pipeline driver also interoperates with the validation component to support the pipeline initiated subtree validations.

7.9 Discussion

This chapter described the XMLPipe transformation model that is the core XMLPipe sub-model, because it enables the adaptive transformation of presentation documents that combine an open set of languages.

The XMLPipe transformation model allows the processing of all necessary presentation documents. Specifically, the proposed transformation algorithm was based on the initial set of assumptions that only allowed the processing of valid XMLPipe documents. Nevertheless, the subsequent algorithm investigation illustrated that the proposed subtree processing order enables the processing of all semantically correct

[3]Figure 7.10 does not illustrate the interoperation between the pipeline driver and the binding component, in order to avoid a cluttered layout.

documents, independently of their validity. Consequently, the XMLPipe transformation model enables the processing of all presentation documents that have a well defined interpretation.

The preprocessing framework requires that the transformation model defines all the necessary interoperation with other sub-models for transforming a document. The XMLPipe transformation model interoperates with the binding, adaptation and validation models. The *bestHCTS* function defines its interoperation with the binding and the adaptation model, in order to retrieve the required transformation semantics and choose the optimal alternative, respectively. The XMLPipe pipeline specifications allow the introduction of two subtree validation alternatives. The transformation pipeline driver interoperates with the validation model, in order to perform each of these alternatives. The details of the validation model interoperation will be described in the following chapter.

The preprocessing framework also requires that XMLPipe ensures the validity of a subtree prior to its transformation, because the result of a transformation is only well defined if its input is valid. A separate validation invocation for each document subtree can be considered as unnecessary processing overhead, if a document is a priori known to be valid. Consequently, the XMLPipe transformation model does not automatically validate each transformed subtree. Nevertheless, a semantics author or an XMLPipe implementation can straightforwardly force the validation of all subtrees. For instance, the semantics authors can insert a deep subtree validation statement as the first step of each transformation pipeline. Such statements would result in the validation of each subtree, prior to its transformation. Alternatively, an XMLPipe implementation can automatically perform the same validation steps, if an implementation specific invocation option is specified. Therefore, the default transformation model provides the necessary functionality for fulfilling the subtree validation requirement. However, it focuses on reducing the transformation computational complexity, as opposed on ensuring the document validity.

Regarding the individual subtree transformations, the preprocessing framework requires the use of an open set of transformation technologies. The well defined common interface of atomic transformations and its simplicity enable the seamless integration of a multitude of transformation technologies, because any transformation can implement it. Specifically, a semantics author can create an XMLPipe specific transformation by directly implementing the atomic transformation interface. Alternatively, existing and future transformation technologies, such as XSL-T, can be supported by the introduction of an atomic transformation wrapper.

A transformation model must provide built-in transformation pipelines, because they allow the integration of the existing and future transformation technologies and enable the enhancement of their functionality. The XMLPipe transformation pipelines are a core component of the proposed transformation model, because the transformation semantics use pipeline specifications to describe the processing of the language constructs. Each pipeline is a recursive composition of atomic transformations, which allows the specification arbitrary complex transformations. Furthermore the use of atomic transformations allows the seamless integration of a multitude of transformation approaches.

The preprocessing framework requires that all transformation components allow the transformation of both complete documents and document portions. The atomic transformations, the transformation pipelines and the transformation driver enable such processing. The subtree data structure is interchangeable from a document, because of its

recursive nature, as described in Section 7.7.1. Only the atomic transformations require special consideration, in order to ensure that the transformation specifications do not use document specific constructs and are adequately designed for subtree processing. The established transformation design principles, in Table 7.3, and their mapping to the individual transformation technologies, such as in Table 7.4, assist the development of transformation specifications that are adequate for subtree processing. Consequently, the design of the XMLPipe transformation model allows subtree processing and also assists the design of adequate transformation semantics.

The preprocessing framework requires a well defined recursive algorithm that uses independently developed transformation specifications. The *transformRev* function enables the orchestration of the individual transformation specifications. Specifically, it defines the separation of a document into its individual subtrees, their transformation and the composition of the transformation results. The transformation specifications do not require inter-language relationships, which would prohibit the adequate processing of documents that combine an open set of languages. In contrast, a specification, which corresponds to a language L, only depends on the XMLPipe transformation design guidelines and the set of L handled constructs. Consequently, the proposed transformation model provides the necessary recursive orchestration of independently developed specifications, for processing mixed namespace presentation documents.

The proposed mixed namespace processing is more powerful than existing transformation approaches, because there is no existing alternative that enables the generic processing of mixed namespace documents. Specifically, XSL-T allows modular stylesheet composition, but it introduces similar drawbacks to the adaptation profiles. The XEBRA browser enables the sequential composition of multiple transformation specifications, but predefined processing sequences are not adequate for generic document processing, as described in Section 7.1. In contrast, *transformRev* uses the XMLPipe integration model, in order to combine independently developed transformation specifications for transforming mixed namespace documents.

The preprocessing framework requires timely transformation execution. However, the computational efficiency of *transformRev* cannot be evaluated. Firstly, a realistic evaluation of its computational complexity is not feasible, because it depends on several factors that cannot be appropriately formulated: the input constructs, their nesting, their corresponding processing semantics, the number of alternative specifications and the computational complexity of the adaptation measure evaluator. Additionally, each transformation can be arbitrarily complex. Consequently, an approximation of the average transformation complexity is not useful, because it is conceptually as valid as an approximation of the complexity of an average Java program. Secondly, assessing the efficiency of an approach requires an efficiency benchmark. Such a benchmark does not exist, because no existing approach offers comparable transformation functionality.

However, experimental measurements can provide an indication of the overall computational complexity, which is more reliable than theoretical approximations. Worst and best case computational complexity approximations are feasible, but they are of limited use, because of the multitude of required assumptions. For instance, consider that a fixed number of simple transformations are applied to each document node. Given the recursive nature of *transformRev*, a worst case scenario is that each document node is a handled construct that can only contain a single child node (in which case the tree becomes a sequence of nodes). In such a case, the computational complexity of *transformRev* is n times the computational complexity of the atomic transformations,

where n represents the number of document nodes. For instance, if the atomic transformations belong to $O(n)$, then *transformRev* belongs to $O(n^2)$. In a similar manner, if the atomic transformations belong to $O(n^2)$, then *transformRev* belongs to $O(n^3)$. Such computational complexity can be prohibitive for the efficient transformation of large documents. However, for the majority of presentation documents, the computational complexity of *transformRev* is of the same order as the atomic transformations complexity, as it will be illustrated by the experimental efficiency evaluation in Chapter 10. Specifically, the structural properties of most presentation documents significantly simplify their transformation. However, such structural properties cannot be formally defined, because they depend on the syntax of the individual languages. Consequently, the definition of *transformRev* does not guarantee the timely execution of transformations. However, the experimental observation of equivalent computational complexity to the atomic transformations illustrates that there are no fundamental complexity issues.

Regarding the document adaptation, the preprocessing framework requires that all transformation processing steps are adaptation requirement sensitive. The integration model transformation driver interoperates with the adaptation component, in order to choose the optimal specification for each document subtree, according to the externally provided adaptation profile. Additionally, the introduced selection pipeline allows the specification of adaptation requirement sensitive pipelines. Furthermore, all atomic transformations can access the adaptation profile. Specifically, XMLPipe specific transformation implementations can directly access the adaptation profile information. Generic transformation specifications can use a set of technology specific methods, which must by provided by the corresponding wrappers.

Additionally, a transformation model must enable the processing of documents that contain both high and low level presentation languages, for a variety of $\mathcal{L}_p$ sets (sets of natively supported languages). The XMLPipe transformation model fulfils this requirement by not restricting the set of input and output languages. Specifically, *transformRev* can adapt a document for any set of natively supported language sets, if the necessary transformation semantics exist. The input of *transformRev* can contain any presentation language that has associated transformation semantics, independently of its abstraction level. Additionally, *transformRev* can transform natively supported languages, which correspond to the lowest possible presentation level for a given adaptation profile, because they are the interface of the target browser. Consequently, the XMLPipe transformation model covers the required adaptation spectrum, and it can adapt both high and low level presentation specifications, according to an unrestricted variety of natively supported language sets.

Adaptation for a variety of adaptation requirements also necessitates the support of an open set of presentation integration models. However, *transformRev* can only support presentation integration models that are less generic than the XMLPipe integration model, according to the expressed assumptions. Nevertheless, such an assumption is not significantly restrictive, because current browser integration models are less generic than the XMLPipe integration model, because they support either no integration model or very limited integration. Therefore, the XMLPipe transformation model does not entirely fulfill the integration model adaptation requirement, but its adaptation capabilities are not significantly harmed, because of the XMLPipe integration model's generality.

The XMLPipe transformation model compares favourably to existing generic adaptation approaches. The CC/PP based stylesheet selection, proposed in [OH02], is the

most prominent existing approach for generic adaptation. The proposed adaptation model allows more precise transformation selection, as described in Section 6.6. Additionally, the handled construct based association of the XMLPipe transformation semantics assists the adaptation of handled constructs that may require significantly different processing. Device independent authoring approaches can offer more powerful adaptation than the XMLPipe transformation model, because they can use the foundation of a well defined presentation model or a constrained set of languages. However, XMLPipe cannot adopt similar techniques, because they would harm its generality. In contrast, the XMLPipe transformation model could be considered as a good balance between adaptation functionality and generality, because it allows powerful adaptation, but it does not restrict the set of supported languages or target platforms.

Finally, the preprocessing framework requires that for a fixed set of adaptation requirements, the document presentation remains consistent, between separate processing instances. The XMLPipe transformation model cannot guarantee such consistency, because the resulting document interpretation depends on a set of freely evolving distributed semantics and the relative adequacy of its members. Any semantics modification or introduction can result in a different processing output. Nevertheless, if there is not malicious modification of the processing semantics, any modification will introduce new functionality or enhance the existing functionality. Therefore, any presentation inconsistencies can be considered as similar to browser presentation inconsistencies that result from presentation engine updates. Consequently, XMLPipe cannot guarantee the consistency of document processing, but the possible inconsistencies would result from infrequent updates, and they can be considered as acceptable for an evolving set of processing semantics.

Summarising, the XMLPipe transformation model fulfils most relevant framework requirements and it compares favourably to existing transformation approaches. Specifically, it fulfils all framework requirements, apart from the adaptation for arbitrary presentation integration models and for efficient document transformation. However, the sole support of presentation integration models that are less generic than the XMLPipe integration model is not significantly restrictive, because XMLPipe integration is more generic than all existing integration models. Additionally, efficient execution cannot be guaranteed, but experimental measurements can illustrate that there are no fundamental algorithmic efficiency issues.

7.10 Summary

This chapter presented the XMLPipe transformation model, which is the core submodel of XMLPipe, because it is responsible for producing a document's interpretation. Specifically, it creates the optimal interpretation of semantically correct presentation documents according to an adaptation profile, using an open set of independently developed transformation specifications.

The XMLPipe transformation model is based on a well defined set of assumptions, which are necessary for its feasibility. Our investigation firstly established a sufficiently restrictive set of assumptions for proving the feasibility of mixed namespace transformation and developing the corresponding algorithm. Subsequently, it provided a looser set of assumptions and a revised algorithm that extend the practical applications of the proposed transformation model. Additionally, a subset of the assumptions provide the necessary design guidelines, for developing transformation specifications that are

adequate for the proposed processing.

The integration model transformation driver implements the core transformation algorithm, which consists of a recursive postorder separation, transformation and composition of the individual document subtrees. The transformation algorithm is based on the XMLPipe integration model, which allows the separate processing of the document subtrees. For each subtree, the integration model transformation driver interoperates with the binding and adaptation models, in order to choose the optimal transformation specification. Subsequently, it delegates the transformation request to the transformation pipeline driver, which is responsible for transforming the individual subtrees.

The proposed transformation pipelines allow the seamless integration of a multitude of transformation technologies. Specifically, each transformation pipeline is a recursive composition of atomic transformations and subtree validation processing steps. The atomic transformations provide a common interface, which allows the integration of separate transformation technologies. Atomic transformation wrappers allow the reuse of existing transformation implementations, such as XSL-T transformers. The validation processing steps enable a semantics author to validate a document subtree prior to its transformation. The proposed pipelines offer three transformation composition methods, which allow sequential transformation composition, runtime transformation creation and adaptation requirements sensitive transformation. The resulting recursive combination of atomic transformations and validation steps allows the composition of arbitrary complex transformations out of simpler specifications.

The XMLPipe transformation model enables the generic preprocessing of presentation documents, because it fulfils most relevant framework requirements and is more powerful than existing approaches. The subsequent chapters will introduce the remaining XMLPipe sub-models, which are necessary for supporting the transformation model.

Chapter 8

XMLPipe validation model

The XMLPipe validation model is responsible for providing the necessary validation functionality to the transformation model and the document authors. The transformation model requires two types of subtree validation: complete subtree validation and handled construct validation. Additionally, the document authoring process necessitates an authoring validation method that evaluates whether a document can be successfully processed. Therefore, authoring validation must evaluate the semantic correctness of a document, because semantically correct documents have a well defined interpretation and can be processed by the proposed transformation model. In a similar manner to the proposed transformation model, a generic validation model must utilise the XMLPipe integration model, in order to enable the composition of independently developed validation semantics for the validation of mixed namespace documents. Consequently, the XMLPipe validation model must provide subtree validation and semantic correctness evaluation of mixed namespace documents, according to an independently developed set of validation semantics.

NRL and NVDL are the most prominent validation approaches, for mixed namespace documents. The seamless integration of a multitude of validation technologies is necessary, because no existing validation is more generic than the others, as described in Section 2.2.1. Neither NRL nor NVDL constrain the used set of schema specifications, since separate validation technologies can be used for the validation of each document subtree. Additionally, the XMLPipe integration model defines the validity of a document according to the validity of its individual subtrees. Consequently, the NRL and NVDL subtree-based validation can be adopted by the XMLPipe validation model. However, it must be extended to accommodate the the concept of semantic correctness and the relationships between the individual subtrees, according to the classification of their root handled constructs.

This chapter separately introduces the validation driver and the validation model interface. The former is responsible for validating document subtrees, according to the XMLPipe integration model. The latter provides the required functionality to the document authors and transformation model.

8.1 Validation driver

The validation driver must address similar issues to the transformation driver, but a more thorough investigation is feasible, because document validation is better defined than document transformation. Section 8.1.1 initiates the validation driver discussion

by proving the equivalence between subtree validation methods and the XMLPipe valid documents definition, under a set of assumptions. Section 8.1.2 proceeds to the investigation of the proposed methods for identifying the places where content is expected (described in Chapter 5), since they influence the valid composition of *COC* rooted subtrees. Section 8.1.3 describes the adequate processing order and subtree separation process. Section 8.1.4 describes the XMLPipe atomic validations, which cover the validation of the individual subtrees and enable the seamless integration of several validation technologies. Section 8.1.5 introduces the necessary validation semantics and their binding. Finally, Section 8.1.6 combines all introduced concepts into the XMLPipe validation driver and algorithm.

8.1.1 Adequacy of subtree validation

According to the XMLPipe integration model, a validation model can process a mixed namespace document by processing its individual subtrees. Such an approach can only be adequate, if it is equivalent to the integration model definition of valid documents. This section establishes the adequacy of subtree-based validation approaches by proving their equivalence to the valid documents definition, under a set of assumptions.

Document validation can be considered as a form of transformation, because both validation and transformation processes can modify a document and return processing feedback. Specifically, a validation process modifies a document when it introduces default attribute values, and it returns feedback on the document validity. A transformation also modifies a document by definition, and it can return feedback on its success or failure. The following validation notation will reflect its relationship with transformation. If the validation of a document d by a validation process V results in d', we will write $d \xrightarrow{V} d'$. The validation process can either succeed or fail, if the document is valid or invalid, respectively.

A usable validation process must provide rich validation feedback that covers most invalid construct occurrences within a document. Consequently, the validation model must rely on a minimal set of assumptions, which do not restrict the processing of well formed documents. Table 8.1 summarises the minimum necessary set of validation assumptions, which are a subset of the transformation assumptions. Firstly, there must be at least one schema specification for each presentation language, in order to allow the separate processing of each document subtree. Secondly, schema specifications must not require that all subtrees are rooted at a single document element, but they must allow handled construct rooted subtrees, since such subtrees are valid XMLPipe documents. Finally, the schema specifications of a language L that contains *FOC* and/or attribute handled constructs must declare a "`foc`" element that contains all such constructs. The predefined `foc` element is similar to transformation model concept of a foreign namespace parent for *FOC* and attribute handled constructs. The validation model cannot reuse the foreign namespace parents of handled constructs, because not all validation technologies allow the declaration of document constructs within arbitrary foreign namespace elements. For instance, the XML Schema does not allow the `<any>` wildcard construct as a top level element. The following validation investigation will use the minimal foundation of the described assumptions and introduce additional assumptions/guidelines as necessary.

According to Corollary 2 (page 123), d is a valid document if and only if there is sequence of single namespace valid documents $d_1, \ldots, d_n$, where d_1 is rooted at either

ID	Assumption	Description
12	Schema existence	For each $L \in \mathcal{L}$, there is at least an associated schema specification.
13	Subtree validation	The design of schema specifications allows the validation of handled construct rooted subtrees.
14	Context validation	Schema specifications that contain *FOC* and attribute handled constructs must declare a `foc` element of the same language, which contains all such constructs

Table 8.1: Validation assumptions

a *COC* or an *SMC* element, so that $d \in (d_n \cdots \underset{Im_X}{\overset{+}{\leftarrow}} (d_3 \underset{Im_X}{\overset{+}{\leftarrow}} (d_2 \underset{Im_X}{\overset{+}{\leftarrow}} d_1)) \cdots)$. In order to prove the adequacy of a subtree validation process V, it is sufficient to show that a document d is valid if and only if $d \xrightarrow{V} d'$ is successful.

A subtree validation process is adequate for single namespace documents ($n = 1$). Consider that d is a single namespace document of a language L. According to Assumption 12, there is a validation process V_1 that can validate d. Additionally, consider that V_1 does not allow stand alone attribute or *FOC* construct rooted subtrees. If the validation process $d \xrightarrow{V_1} d'$ is successful, then the nesting of d constructs is valid and d is rooted at a *COC* or an *SMC* element handled construct. Therefore, if the validation process is successful, the document is valid ($d \in \mathcal{V}_L^{Im_X}$). Conversely, if $d \in \mathcal{V}_L^{Im_X}$, it validly combines the constructs of L and it is rooted at either a *COC* or an *SMC* element construct. According to the subtree validation assumption 13, the validation process $d \xrightarrow{V_1} d'$ must be successful. Consequently, for single namespace documents a subtree validation process V is equivalent to the XMLPipe integration model definition, if V does not allow stand alone attribute or *FOC* construct rooted subtrees.

Regarding mixed namespace documents, if d is valid, then the subtree validation process is successful. Specifically, if $d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$, there is a sequence of $n \geq 2$ single namespace valid documents $d_1, \ldots, d_n$, so that $d \in (d_n \cdots \underset{Im_X}{\overset{+}{\leftarrow}} (d_3 \underset{Im_X}{\overset{+}{\leftarrow}} (d_2 \underset{Im_X}{\overset{+}{\leftarrow}} d_1)) \cdots)$. $\forall i \in [1, n]$ $\mathcal{L}_{d_i} = \{L_i\}$, $d_i \in \mathcal{V}_{L_i}^{Im_X}$ and d_1 is rooted at either a *COC* or an *SMC* element handled construct. According to Assumption 12, there is a validation process V_1 for each subtree d_i. If d_1 is rooted at a *COC* or an *SMC* element construct, its corresponding validation V_i will be successful, in a similar manner to the above single namespace case. If a subtree d_i is rooted at an attribute or a *FOC* construct, its validation within the predefined `foc` element must also be valid, according to Assumption 14. Consequently, the subtree validation of a valid mixed namespace document is successful, if it processes each *FOC* or attribute handled construct rooted subtree within the predefined `foc` element.

Conversely, if d is invalid, the validation process will be unsuccessful. Specifically, if d is invalid, it cannot be validly composed out of a sequence of valid single namespace subtrees. Therefore, it either consists of invalid single namespace subtrees and/or their nesting is not valid. In the former case the subtree validation process will fail, because if a subtree d_i is invalid and V_i corresponds to L_i, $d_i \xrightarrow{V_i} d_i'$ will fail. If the nesting of the subtrees is invalid, either there a *COC* rooted subtree that occurs at a place

ID	Assumption	Description
15	`foc` introduction	The separation of subtrees that are rooted at *FOC* or attribute handled constructs must introduce the predefined `foc` element.
16	Context test	The validation of *FOC* and attribute handled construct rooted documents must fail.
17	Handled construct test	The subtree separation process must fail, if the root of a subtree is not a handled construct.
18	*COC* test	The subtree separation process must fail, if a *COC* rooted subtree appears at a place where no content is expected.

Table 8.2: Subtree separation design principles

where no content is expected and/or the root of d is an attribute or a *FOC* handled construct. A subtree validation process will fail, if it ensures the *COC* nesting validity and that attribute and *FOC* constructs do not occur as document roots. Consequently, subtree validation process will fail for invalid documents, if it tests the valid occurrence of *COC*, *FOC* and attribute handled constructs.

Consequently, under the introduced assumptions, a subtree validation process is equivalent to the integration model definition of valid documents, for both single and mixed namespace documents. Table 8.2 summarises the additional introduced assumptions, which express the subtree separation design principles. Specifically, a subtree separation process must introduce the necessary predefined `foc` constructs, when it separates subtrees that are rooted at either *FOC* or attribute handled constructs. Additionally, it must fail when the nesting of the individual subtrees is not valid: when a subtree is not rooted at a handled construct, when *FOC* or attributes appears as document root constructs and when *COC* constructs occur at places where no content is expected. Summarising, if a subtree validation process adheres to the principles in Table 8.2 and the language schemas follow the assumptions of Table 8.1, the validation of a document is successful if and only if the document is valid.

8.1.2 *COC* placeholders identification

A validation process must be aware of the places where content is expected, because they determine the valid nesting of *COC* rooted subtrees. The defined integration model semantics do not specify these places, because their identification is closely related to the syntax of the presentation languages. Section 5.3.3 described two methods for identifying the places where content is expected: the explicit identification method and the heuristic method. A validation process must incorporate at least one of these methods, in order to adequately validate a document.

The explicit identification method can be straightforwardly applied, if the subtree separation process substitutes each *COC* rooted subtree with a predefined construct. According to the explicit identification method, each schema must denote the places where content is expected. If the subtree separation process substitutes each *COC* rooted subtree with the predefined `coc` element, it is sufficient that the validation schemas allow a valid occurrence of `coc` at all places where content is expected. In a similar manner to the predefined `foc` element, `coc` must belong to the same namespace

as its parent element. For instance, consider a document d of a language L that contains a COC rooted subtree d' of another language L'. Additionally, consider that the subtree separation process maps d to d'' by the substitution of d' with the predefined construct `coc`. If d is valid, the validation of d'' will be successful, because `coc` will occur at a place where content is expected and the schema of L must allow `coc` at all places where content is expected. Otherwise, the validation will fail, because the subtree separation process will introduce a `coc` element at an invalid place, according to schema of L. Consequently, the explicit identification method is a good candidate, because it only requires an adequate subtree separation process and minimal schema modifications.

The heuristic method provides a fallback mechanism when there are no XMLPipe specific schemas, but its incorporation would significantly complicate the reuse of existing validation approaches. The application of the heuristic method requires information on which language constructs can contain COC constructs of the same language. The language schemas provide such information, but most validation technology implementations, such as XML Schema validators, do not provide the necessary interfaces to obtain it. Consequently, the XMLPipe specific validators would be unable to reuse existing implementations, because they would have to implement specific interpreters of the corresponding schemas.

Consequently, the XMLPipe validation model will only adopt the explicit identification method, in order to avoid complicating the incorporation of existing validation technologies.

8.1.3 Subtree separation and processing order

Modular validation technologies, such as XML Schemas, can validate mixed namespace documents without requiring subtree separation. For instance, XML Schema can validate a mixed namespace document, if there is a schema specification for each language and each schema allows valid occurrences of foreign namespace subtrees, according to the XMLPipe integration model. Such approaches are more computationally efficient than subtree separation approaches, because they do not require the separate processing of each subtree, the introduction of predefined constructs and the separate invocation of validator implementations. However, they restrict the usage of validation technologies, because all combined schemas must use the same validation technology. Moreover, the introduction of the necessary integration placeholders impedes the development of the schemas, because it significantly increases their complexity. Consequently, such approaches are feasible, but they cannot form the basis of the XMLPipe validation model.

An adequate subtree separation process for validation must create separate subtrees at the namespace boundaries of a document, unlike the *transformation* subtree separation. The transformation model separates all handled construct rooted subtrees, independently of the namespace of their context. Such a separation assists the use of significantly different transformers, for the processing of subtrees that belong to the same language. Moreover, it impedes the introduction of intra-language transformation relationships that can harm the uniform and context independent processing of handled construct rooted subtrees. None of these issues applies to document validation. Specifically, syntax specifications of a language's constructs are unlikely to require separate validation technologies, because each validation technology must allow the syntax specification of all the constructs of language. The majority of languages should not require intra-language handled construct constraints, because the proposed

transformation processing separately addresses each handled construct subtree. However, prohibiting any class of syntax constraints should be avoided, because the more syntax constraints a validation method can express the more powerful syntax specifications it allows. Consequently, a validation subtree separation process must only create separate subtrees at a document's namespace boundaries.

A validation subtree separation process must also introduce the predefined `foc` and `coc` constructs. The validation of both attribute and *FOC* element handled constructs must occur within the context of the predefined `foc` element, according to Assumption 15. Additionally, the explicit identification method of the places where content is expected requires the introduction of `coc` elements, as indicators of all separated *COC* rooted subtrees.

The order of the subtree separation and validation is irrelevant, but a preorder processing sequence is beneficial. The processing order is irrelevant because subtree validation only introduces minor tree modifications, as opposed to transformation. Such minimal modifications do not influence the validation of a subtree's content or context; consequently, the processing order is insignificant. However, a preorder processing process allows the straightforward support of the handled construct subtree validation, which only processes the topmost constructs that belong to the same namespace as a subtree's root. If a preorder validation process is used, handled construct subtree validation can be achieved by only performing the first validation iteration. Consequently, the XMLPipe validation subtree separation creates separate subtrees at the namespace boundaries of a document, in a preorder manner.

8.1.4 Atomic validations

A common validation interface allows the seamless integration of multiple validation technologies. In a similar manner to the atomic transformations, the XMLPipe atomic validations provide such an interface. A validation process is also a transformation process, as described in Section 8.1.1. Consequently, the XMLPipe atomic validations can be defined as the transformation processes that map a document to its validated form, according to an optional external input. Atomic validation wrappers are responsible for the incorporation of generic validation technologies, such as XML Schema, and they can use the external transformation input to retrieve the schema specifications. The set $\mathcal{A}^{\mathcal{V}}$ will represent the set of all atomic validations.

XMLPipe atomic validation processes ($\mathcal{A}^{\mathcal{V}}$): The set of all *XMLPipe atomic validation processes* $\mathcal{A}^{\mathcal{V}}$ is the subset of all transformations that map an XMLPipe document to its validated form. For each $V \in \mathcal{A}^{\mathcal{V}}$ the validation process $d \xrightarrow[I]{V} d'$ is successful if and only if d is a valid document, according to the external input I. d' is the validated result.

Each atomic transformation must follow a set of design guidelines, in order to be adequate for the XMLPipe validation model. Table 8.3 summarises the atomic validation design guidelines. Specifically, according to the last two assumptions of Table 8.1, each schema specification must allow the validation of handled construct rooted subtrees by providing top level declarations of all *SMC* and *COC* element handled constructs and declaring all *FOC* and attribute rooted subtrees within the predefined `foc` element. Additionally, atomic validations must allow the occurrence of the `coc` construct at all places where content is expected, according to the adopted explicit

ID	Description	XML Schema
13	Declaration of the valid element *SMC* and *COC* rooted subtrees.	Top level handled construct declarations, such as `<xsd:element name=''p''>`
14	Declaration of *FOC* and attribute handled construct rooted subtrees, within the context of the predefined `foc` element.	Introduce a `foc` top level element: `<xsd:element name=''foc''>` *FOC elements/attributes* `</xsd:element>`
19	Allow the occurrence of a `coc` element at places where content is expected.	Include an `<xsd:any maxOccurs=''unbounded''>` rule at places where content is expected.
20	No explicit document root references or other absolute path expressions.	XML Schema does not contain such constructs.

Table 8.3: Design guidelines and their XML Schema mapping

identification method. Furthermore, schema specifications must not contain absolute references, in a similar manner to the atomic transformations.

The atomic validation wrapper developers must map the above design guidelines to the validation technology specific equivalents, in order to assist the schema authoring process. The rightmost column of Table 8.3 illustrates equivalent XML Schema guidelines. The first two are direct applications of their descriptions, using the XML Schema syntax. An XML schema can denote the places where content is expected by either explicitly declaring the predefined `coc` element or using the `<xsd:any>` construct, if its top level declarations only include handled constructs. The latter is beneficial, because it allows the occurrence of both the `coc` element and all language handled constructs. No XML Schema specific guideline is necessary for avoiding document specific constructs, because XML Schema does not contain such constructs.[1]

8.1.5 Validation semantics

The XMLPipe validation semantics of a language consist of multiple pairs of a schema specification and an atomic validation implementation reference. *VSemantics* will represent the set of all validation semantics and Figure 8.1 illustrates the validation semantics organisation. In addition to the schema specification, the atomic validation implementation reference is necessary, because it allows the use of an unrestricted set of validation technologies. Each language is associated with one or more instances of validation semantics, because the syntax of a language can be defined in several schema languages. As opposed to the transformation semantics, handled construct based binding is not necessary, because document validation is a more constrained process that does not require separate validation technologies for the individual constructs of a language. The function $locateVS()$ is a part of the binding component interface, and it maps a language's URI to its corresponding set of validation semantics.

[1]However, other schema languages may contain document specific constructs. For instance, Schematron[Jel03] defines the syntax of XML languages, using XPath structure assertions that can be document specific.

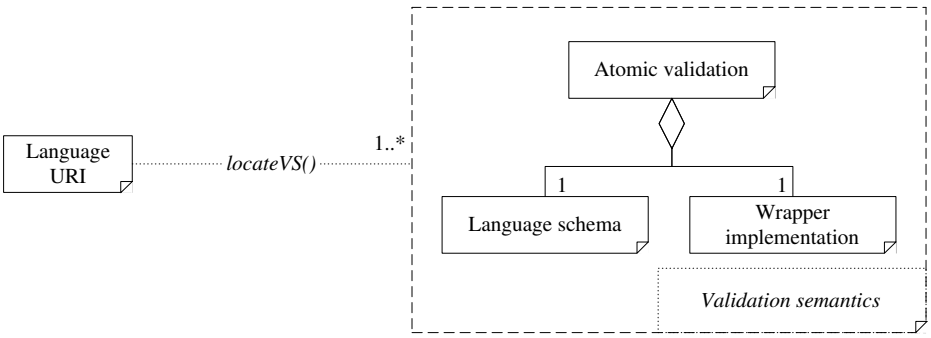

Figure 8.1: Language validation semantics

Validation semantics (*VSemantics*): *VSemantics* contains all XMLPipe validation semantics. Each member of *VSemantics* is a pair of references to a schema specification and to the corresponding atomic validation implementation.

Validation semantics location function (*locateVS*): The validation semantics location function $locateVS : URI \rightarrow \wp(VSemantics)$ maps an XML language URI to the set of its corresponding validation semantics. $\forall uri \in URI, locateVS(uri) = \{vs_1, vs_2, \ldots, vs_n\}$, where $\forall i$, vs_i is one of the alternative validation semantics for the language that corresponds to *uri*.

The selection of the optimal validation alternative is not well defined, because all alternatives are equivalent, since they must define the same syntax. However, XMLPipe implementations can use implementation specific parameters to choose between multiple validation alternatives. For instance, consider a language that has both XML Schema and Relax-NG syntax specifications. An implementation that has a cached XML Schema wrapper implementation, might choose the former schema, in order to avoid the extra overhead of remotely retrieving a Relax NG wrapper. Alternatively, it can choose the Relax NG specification, because Relax NG is more expressive than XML Schema, as described in Section 2.2. Such implementation specific issues are outside the scope of the XMLPipe model; consequently, the optimal validation selection process is delegated to the design of its implementations.

8.1.6 The validation algorithm

The aforementioned validation discussion established the necessary foundation for an algorithmic definition of the subtree validation process. Figure 8.2 illustrates the algorithmic definition of function *validate*. *validate* validates a subtree of a document d that is rooted at a node n. If the boolean parameter *deep* is false, it inhibits the processing of foreign namespace subtrees, in order to allow the application of *validate* for both the complete subtree validation and handled construct validation.

validate is recursively defined according to the subtree design principles, summarised in Table 8.2, and the proof of the subtree validation adequacy, illustrated in Section 8.1.1. Specifically, it firstly ensures that the subtree root node is a handled

Validation algorithm (*validate*): *validate* validates the subtree of d that is rooted at the node n and results in the validated document.

function *validate*(D d, *Node* n, *Boolean deep*) $\rightarrow D$
  let $\sigma = (uri, s)$ be the construct that corresponds to n
  let L be the language that corresponds to σ
  //Instantiate the atomic validation for the identified subtree
  let $VS = locateVS(uri)$
  if ($\sigma \notin langConstructs^{HC}(L)$ OR $VS == \emptyset$) the validation **fails**; **end if**
  let *vs* be the most appropriate member of VS (implementation specific)
  let $V \in \mathcal{A}^{\mathcal{V}}$ be the validation process that corresponds to *vs*
  //Separate the subtrees
  for each subtree d_i of n rooted at a construct σ_i of a language $L_i \neq L$
    separate d_i from d
    if ($\sigma_i \in COC(L_i)$) AND ($deep == true$)
      add a $(uri, \texttt{coc})$ element at its place
    end if
  end for
  //apply the validation
  let d' be the subtree of d, which is rooted at n
  if $\sigma \in langConstructs_a(L)$ OR $\sigma \in FOC(L)$
    if n has an ancestor n''
      let $d' = (uri, \texttt{foc}) \xleftarrow{+} d'$ *//add the predefined* `foc` *construct*
    else
      the validation **fails**
    end if
  end if
  apply V to d': $d' \xrightarrow{V} d''$
  if the application of V was unsuccessful
    the validation **fails**
  end if
  //re-compose the subtree and recurse
  remove all $(uri, \texttt{foc})$ and $(uri, \texttt{coc})$ elements from d''
  replace d' with d'' within d
  for each separated subtree rooted at n_i
    add the subtree at its corresponding place in d
    if $deep == true$
      call *validate*$(d, n_i, deep)$
    end if
  end for
  the validation is **successful**; **return** d
end function

Figure 8.2: XMLPipe validation algorithm

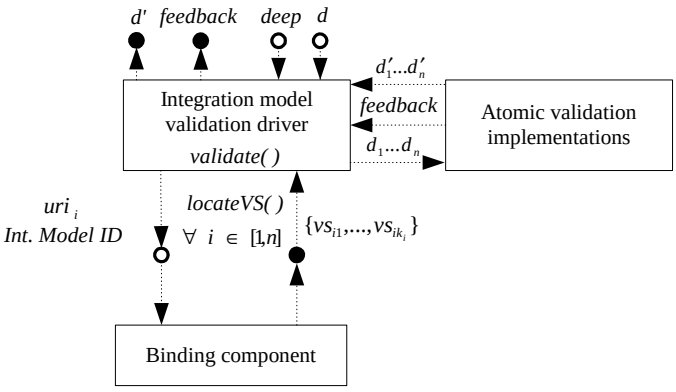

Figure 8.3: Integration model specific validation: top level

construct and that there is a corresponding validation semantics specification. Subsequently, it chooses the optimal semantics alternative, in an implementation specific way, and it instantiates the corresponding atomic transformation process V. Prior to the application of V, *validate* prepares the document subtree by separating any foreign namespace subtrees and adding the predefined `foc` and `coc` constructs. The `coc` constructs are not added when *deep* == *false*, because the shallow validation process does not test the nesting of foreign namespace subtrees. The prepared subtree only contains constructs of L and the predefined `coc` and `foc` elements. The application of V tests the validity of the L constructs and the valid nesting of COC rooted subtrees, because of the introduced `coc` elements. If *deep* == *false* the validation process concludes with the removal of all introduced `coc` and `foc` elements and the recomposition of the subtrees. Otherwise, *validate* also recurses into the foreign namespace subtrees. If *validate* terminates successfully the specified subtree is valid, because it consists of valid single namespace subtrees that are nested in a valid way.

Figure 8.3 illustrates the XMLPipe validation driver that drives a document's validation process by implementing *validate* and interoperating with the binding component and the atomic validation implementations. Specifically, the validation driver maps an input document d to its validated version d'. The *validate* input parameters are a document and a subtree's root node, but the illustrated validation driver omits the node information, in order to be consistent with the transformation model illustrations. For each document subtree $d_1, \ldots, d_n$, the validation uses the binding component to retrieve the corresponding set of validation specifications. Subsequently, it instantiates the optimal atomic validation processes, which are responsible for validating the individual subtrees.

8.2 Validation model interface

The validation model interface is a set of wrapper components that use the validation driver functionality to serve the validation requests of the transformation model and the document authors. Specifically, there are two processing validation wrappers and an authoring validation wrapper. The former are thin wrappers, because they only

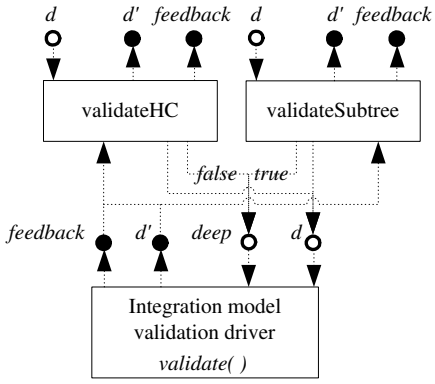

Figure 8.4: XMLPipe processing validation interface

delegate the validation requests to the validation driver, since *validate* provides all necessary functionality for both complete subtree and handled construct validation. In contrast the authoring validation wrapper requires more substantial processing, because it must orchestrate both the validation and the transformation drivers, in order to overcome the *SMC* validation issues.

8.2.1 Processing validation interface

The processing validation interface consists of two separate wrappers, illustrated in Figure 8.4, that correspond to the two transformation pipeline validation constructs: `validateSubtree` and `validateHC`. The transformation pipeline driver invokes both components as atomic transformations. For element *SMC* and *COC* rooted subtrees, the document subtrees provided by the transformation model can be directly processed by *validate*. Regarding the subtrees that are rooted at either a *FOC* or an attribute handled construct, *transformRev* provides a subtree that is rooted at the parent of the corresponding handled construct. The parent element inclusion allows correct subtree validation, because *validate* tests the valid nesting of *FOC* and attribute rooted subtrees by ensuring their occurrence within a parent element. Both wrappers must provide the handled construct node to *validate*, as opposed to its parent, because the n parameter of *validate* must be the root of a valid subtree. Consequently, it is sufficient that `validateSubtree` and `validateHC` call $validate(d, n, deep)$, where d is their input document subtree, n is the topmost subtree handled construct and *deep* is either true, for `validateSubtree`, or false, for `validateHC`.

8.2.2 Authoring validation

Authoring validation could evaluate several document characteristics, but semantic correctness is the most adequate feasible alternative. The other alternatives are the document's validity, according to the XMLPipe integration model, and the feasibility of its transformation. The former consists of a single *validate* function call, but it is not adequate for semantically correct but invalid documents. The latter is the most beneficial, because its provides an indication of whether the document and the corresponding language semantics follow the XMLPipe design principles. However, it would require

to transform a document for all possible adaptation profiles, because each profile can result to different transformation invocations. Such exhaustive processes are not feasible, because there are infinitely many combinations of adaptation requirements and sample space reduction optimisations are ineffective. For instance, a validation model can attempt to processing a document for all processing semantics alternatives, which may be less numerous than the adaptation profile alternatives. The value of such optimisations is limited, because they do not guarantee successful document processing and they are also prohibitively inefficient, because of the open set of languages and processing semantics. Consequently, the evaluation of semantic correctness can be considered as an good balance between the feasibility and adequacy of a validation process.

The semantic correctness of a document can be evaluated by a two step process, which eliminates the *SMC* rooted subtrees before validating the document. A document is semantically correct, if it is either a valid XMLPipe document or an invalid document that becomes valid after the processing of its *SMC* rooted subtrees, as described in Section 7.6. Therefore, if a document is semantically correct, a pre-validation processing step that transforms all *SMC* subtrees must produce a valid document. The validity of the resulting document can be evaluated by a deep subtree validation.

The correct usage of the *SMC* constructs is essential for both the document's validity and the success of their transformation. Therefore, an additional validation step is necessary for the *SMC* subtrees, because they are not visible to the above deep subtree validation process. Deep subtree validation is not adequate for the *SMC* rooted subtrees, because the nesting of their foreign namespace subtrees is irrelevant to their processing and it is validated by the subsequent deep subtree validation. For instance, the nesting of the `alt:alt` subtree constructs, in lines 2–9 of the transformation driving example (page 116), is not valid, but it becomes valid after its processing. In contrast, handled construct validation is both adequate and sufficient. Specifically, it ensures the valid usage of the *SMC* construct and its top level descendants that are interpreted by the subtree transformation. Additionally, it does not validate the foreign namespace subtrees, which are simply copied by the *SMC* transformation. Consequently, the authoring validation process must perform a handled construct validation step for each *SMC* rooted subtree, before its transformation.

The introduction of a transformation processing step can harm the adaptation requirements independence of the authoring validation process. Adaptation requirements independence is essential, because exhaustive evaluation for all adaptation requirement combinations is not feasible, as described above. However, the transformation of most *SMC* constructs either does not depend on the adaptation requirements or it does in validation insignificant ways. For instance, the processing of the driving example `imp:import` construct is adaptation requirements independent. In contrast, the processing of the L_{alt} depends on the adaptation requirements, because they influence the `alt:case` element selection. However, the alternative transformation results do not differ in validation significant ways. For example, the transformation of the `alt:alt` construct, in line 2 of the driving example, results in separate document title definitions, according to the adaptation requirements. However, both alternatives produce a `doc:title` element that contains a child text node, and they are equivalent in terms of their validation. Consequently, under the assumption that the processing of *SMC* constructs does not differ in validation significant ways for separate adaptation requirement sets, the authoring validation process can remain adaptation requirements independent by using a dummy adaptation profile for the *SMC* subtree transformations.

The dummy profile can be minimal and only state support for the input document's

languages. Specifically, the binding adaptation specifications of the *SMC* transformation semantics can use the value of any adaptation term. However, the dummy profile does not have to contain a value for each adaptation term, because each term has a default value that can be used for evaluating adaptation expressions. Additionally, the specification of the supported languages is not necessary. However a set that includes all document languages is consistent with the subsequent document processing by a validation process that can process all document languages. Consequently, the dummy profile will consist of a single adaptation statement that states support for all document languages.

The dummy profile can partially drive the *SMC* transformation process, but transformation algorithm modifications are necessary. Consider a document d and the corresponding dummy profile pr that states support for all languages in $\mathcal{L}_d$. Most transformation specifications associated with *COC* or *FOC* require support for languages that existing browsers support, as opposed to the typically more abstract languages in $\mathcal{L}_d$. Consequently, the above dummy profile would prohibit the processing of most non-*SMC* rooted subtrees. However, the *SMC* constructs transformation can introduce content from additional languages, which may not belong to $\mathcal{L}_d$. If the introduced content consists of *SMC* rooted subtrees, *transformRev* may process them successfully, since their associated semantics may be adaptation requirements independent. However, if the introduced content is *COC* or *FOC* rooted subtrees of languages not in $\mathcal{L}_d$, the transformation process will most probably fail, because it is unlikely that adequate transformation semantics will exist. Consequently, the dummy profile can drive the *SMC* transformation process, if no low level presentation constructs are used, no semantics specifications are adequate for high level constructs and no inclusion of foreign language *COC*/*FOC* subtrees takes place. Such constraints are both overly restrictive for a validation process and ambiguous, because the distinction between low and high level constructs cannot be well defined. Consequently, the dummy profile is not sufficient for driving the *SMC* transformation process.

A set of transformation algorithm modifications address all the above issues and enable the elimination of *SMC* subtrees without restricting the set of processed documents. Figure 8.5 illustrates the necessary modifications: the underlined statements denote new or modified statements and the arrows ($\Longleftarrow$) denote the omission of one or more statements. The first modification introduces the necessary dummy profile, which states support for all document languages. Subsequently, the selection of all handled construct rooted subtrees has been substituted with the selection of all *SMC* handled constructs that have an adequate transformation specification. Prior to the transformation application, a handled construct validation step is performed, in order to ensure the validity of the topmost subtree constructs. The transformation is only performed, if the validation is successful. *transformAuth* never fails, because it only assists the subsequent validation process, which is responsible to producing the validation feedback. Consequently, it does not fail when a transformation is unsuccessful or if there are no adequate transformation specifications for an *SMC* rooted subtree.

The algorithm modifications are not backwards compatible with *transformRev*, because it is only adequate for *SMC* elimination and not for generic document transformation. Therefore, the transformation driver must implement both *transformRev* and *transformAuth*, in order to provide the necessary functionality for both document transformation and validation.

The authoring validation can be defined as a two step transformation. *validateAuth* will represent the composite transformation process that firstly calls *transformAuth*,

Authoring validation transformation (*transformAuth*): *transformAuth* is a transformation that eliminates as many *SMC* rooted subtrees as possible.

function *transformAuth*$(D\ d) \rightarrow D$
    let pr be the dummy profile that states support for all languages in $\mathcal{L}_d$
    let $d' = d$
    let n be the first node of d', according to a postorder tree traversal
    while (true)
        let σ be the XML construct that corresponds to n
        let L be the language that corresponds to σ
        let n' be the the next postorder tree traversal node of n
        if $L \in SMC(L)$ AND $bestHCTS(\sigma, pr) \neq \epsilon$
⟸ **if** $\sigma \in langConstructs_e^{HC}(L)$
                Separate d_1 from d', where d_1 is rooted at n.
            else
                Separate d_1 from d', where d_1 is rooted at the parent of n.
            end if
            let pr' be all adaptation statements associated with L
            apply a handled construct validation to d_1
            if the validation was successful
                apply T' to d_1: $d_1 \xrightarrow{T'} d_1'$ using both the initial pr and pr'
            else $d_1' = d_1$
            end if
            let pr'' be the set of all L-specific statements introduced by T'
            if $d_1' \neq d_1$ AND d_1' is not an empty tree
                Associate pr'' with L and with the parent of n
                Replace d_1 with d_1' within d'
                let n be the first node of d_1', according to a postorder traversal
            else
                if there is a language specific pr_1 for the parent of n'
                    discard pr_1
                end if
                let $n = n'$
            end if
⟸ **else** *//the n rooted subtree does not require separate processing*
            if n the root of d'
                the transformation is **successful**; **exit**
⟸ **else**
                let $n = n'$
                if there is a language specific pr_1 associated with the parent of n'
                    discard pr_1
    end if; **end if**; **end if**; **end while**
end function

Figure 8.5: Authoring validation transformation algorithm

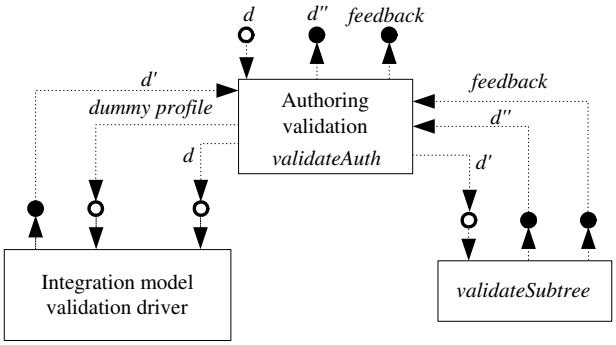

Figure 8.6: XMLPipe authoring validation

which removes as many *SMC* rooted subtrees as possible, and subsequently calls `validateSubtree`, which performs a deep subtree validation. Figure 8.6 illustrates the authoring validation wrapper, which interoperates with both the transformation driver and the `validateSubtree` wrapper, in order to perform the *validateAuth* transformation.

Authoring validation (*validateAuth*): The authoring validation is a transformation that maps a document d to its validated output d'', and it is composed out of a pre-validation transformation step and a subsequent subtree validation step: $d \overset{validateAuth}{\longrightarrow} d''$ iff $d \overset{transformAuth}{\longrightarrow} d'$ and $d' \overset{validate}{\longrightarrow} d''$

8.3 The complete validation model

Figure 8.7 illustrates the complete XMLPipe validation model, which consists of the integration model validation driver, the two processing validation wrappers and the authoring validation wrapper. The integration model validation driver is the core validation component and it combines independently developed validation semantics to validate mixed namespace documents, according to the XMLPipe integration model. The `validateHC` and `validateSubtree` wrappers translate the corresponding pipeline validation requests to validation driver requests. The authoring validation wrapper allows the evaluation of a document's semantic correctness. In addition to the validation driver, it also uses the transformation driver to eliminate the *SMC* rooted subtrees.

8.4 Discussion

A generic validation model must provide the necessary authoring and processing validation to fulfil the preprocessing framework requirements. Specifically, it must allow the document author to validate a presentation document, in a device independent manner. Additionally, it must fulfill the transformation pipeline validation requests by performing both deep subtree validation and handled construct validation.

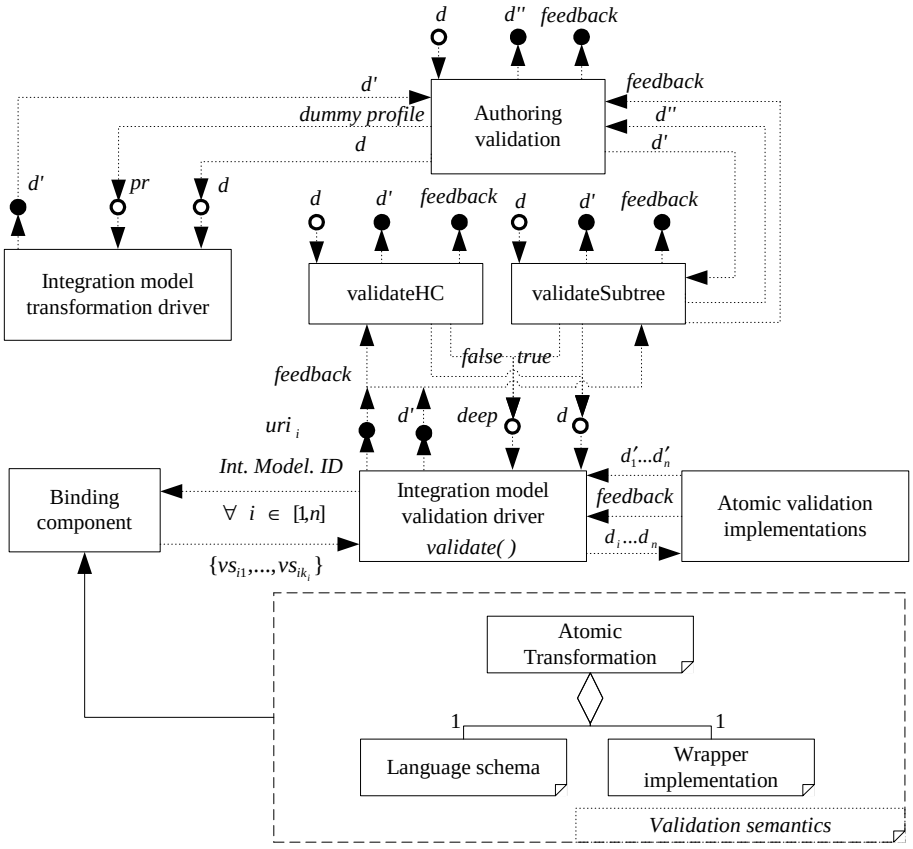

Figure 8.7: The XMLPipe validation model

The three proposed wrappers use the validation driver to provide all necessary functionality. The validation driver evaluates the validity of mixed namespace documents, according to the XMLPipe integration model. The validation driver processes each document subtree separately, in a process that is proven to be equivalent to the XMLPipe valid documents definition. The two processing validation wrappers offer handled construct and deep subtree validation by forwarding the pipeline driver requests to the validation driver. The authoring validation wrapper uses both the transformation and the validation drivers, in order to evaluate the semantic correctness of a document. Semantic correctness is more adequate for authoring validation than validity, because the XMLPipe transformation model can process semantically correct but invalid documents. The authoring validation process uses the transformation driver, but it uses a dummy adaptation profile to remain adaptation requirements independent. This independence is illustrated by the *validateAuth* definition that does not require an adaptation profile parameter. Therefore, the XMLPipe validation model provides the necessary functionality and fulfils the preprocessing framework requirement for authoring validation.

The preprocessing framework requires that both languages and their corresponding validation semantics can be independently defined. The schema design guidelines

and the predefined `foc` and `coc` constructs allow such independent definitions. The proposed guidelines use these predefined constructs, as opposed to foreign namespace references. The semantics authors can use them to specify the places where content is expected and to provide well defined context for the context-dependent *FOC* and attribute handled constructs. Specifically, a schema specification must allow valid occurrences of `coc` at all places where arbitrary presentable content is expected. Moreover, all top level attribute and *FOC* handled constructs must be declared within the `foc` element. Such validation semantics definitions do not introduce inter-language relationships, which would impede the independent development of languages and semantics definitions.

The preprocessing framework requires document subtree validation according to an unrestricted variety of validation technologies. The XMLPipe atomic validation interface does not restrict the individual validation processes, and it can be implemented by any autonomous validation technology. Additionally, the validation design guidelines, summarised in Table 8.3, are sufficient for the development of atomic validations that are adequate for subtree processing. If atomic validation wrapper developers map these guidelines to their validation technology equivalents, such as the XML Schema guidelines, they further assist the development of adequate schemas by the semantics authors. In addition to the atomic validations, both the validation driver and the wrappers can process document subtrees. Consequently, the XMLPipe validation model allows the validation of document subtrees, using a multitude of autonomous validation technologies.

The preprocessing framework requires that each subtree is validated prior to its transformation. As described in Section 7.9, the XMLPipe transformation model can fulfill this requirement, but it does not validate subtrees prior to their transformation, in order to avoid the additional processing overhead. Authoring validation is an approximate alternative that overcomes the processing overhead problem, because the document must only be validated once. Specifically, authoring validation ensures that most *SMC* subtrees can be successfully processed and result to a valid document. If the transformation specifications follow the transformation design principles, the processing of each subtree will result to a valid output subtree that occurs at a valid place, because the input subtree is valid and occurs at a valid place. Consequently, if all *SMC* trees can be validated and all design principles are followed, an authoring validation success is a good indication that every transformed subtree will be valid. Therefore, in addition to the explicit subtree validation, authoring validation provides an approximate method to ensure the validity of the subtrees, which does not depend on the adaptation requirements and does not slow down the transformation process.

The preprocessing framework requires timely validation execution, in a similar manner to the transformation execution. Document validation is better defined than transformation, and the computational complexity of *validate* can be approximated.

The proposed atomic validation interface does not restrict the individual validation processes, which can be arbitrarily complex. However, tree-automata can cover all required validation functionality, as described in [MLM01], and their investigation can indicate the upper necessary computational complexity. [BML+04] investigates tree-automata and suggests that the computational complexity of validating a DOM tree with n nodes is $O(n \log(n))$. Consequently, atomic validations can be arbitrarily complex, but the upper necessary computational complexity is $O(n \log(n))$.

Function *validate* also processes the individual document subtrees in $n \log(n)$ time.

Specifically, consider a document subtree, which contains n nodes, after the separation of its foreign namespace subtrees. *validate* processes each subtree by locating and instantiating the appropriate atomic transformation, separating and recombining any foreign namespace subtrees, applying the atomic validation and removing any added `coc` and `foc` constructs. The atomic transformation location and instantiation processes can be considered to take a fixed amount of time c, because they do not depend on n and the number of validation alternatives is relatively insignificant. Additionally, the separation of the foreign namespace subtrees requires $2n-1$ iterations, which is the number of iterations required by a preorder traversal. The subsequent atomic validation application processes the n subtree nodes in $O(n\log(n))$ time, as described above. Finally, the tree recomposition and the removal of the predefined elements can approximately be performed in $O(n)$ time, because their number can be considered to be proportional to the subtree size. Consequently, the computational complexity of each subtree iteration is $O(c+2n-1+n\log(n)+n+n) = O(n\log(n))$.

validate combines the individual subtree validations, and it validates a document in $O(nlog(n))$ time. Consider document with n nodes that consists of k subtrees that contain $n_1, n_2, \ldots, n_k$ nodes. The described subtree validation process occurs exactly once for each document subtree. Therefore, its computational complexity is $O(n_1\log(n_1)+n_2\log(n_2)+\cdots+n_k\log(n_k)) \subseteq O(n\log(n))$. Consequently, the proposed validation algorithm belongs to the same computational complexity order as existing validation approaches. Its execution can be slower than single namespace subtree validation, because, in addition to the subtree validation, each iteration must locate the necessary semantics, instantiate the atomic validations and perform tree modifications. However, the diminished performance is offset by the functionality gains, since *validate* enables the validation of documents that combine an open set of languages.

NRL and NVDL are the most prominent existing approaches for generic mixed namespace validation. Both approaches can use a multitude of validation technologies to validate the individual subtrees of a mixed namespace document. The XMLPipe validation model adopts their core concepts and allows the independent subtree validation by atomic validations, which can be implemented by any validation technology. However, it is also based on the XMLPipe integration model that provides a well defined document interpretation and enables better presentation document validation, which overcomes the NRL/NVDL erroneous validation problems (described in Section 2.2.2). Additionally, the XMLPipe validation model allows the evaluation of a document's semantic correctness, which is a better indication, than simple validity, of whether a document can be successfully processed. Consequently, the proposed XMLPipe validation model adopts the most prominent existing concepts and extends them to provide further validation functionality.

Summarising, the proposed validation model fulfils all relevant processing framework requirements and provides both authoring validation and the necessary processing validation functionality. Consequently, it is an adequate validation model for a generic preprocessing model.

8.5 Summary

A validation model is an essential component of a preprocessing model, because it can provide the required validation functionality for the transformation model and the document authors. This chapter described the XMLPipe validation model, which offers the

required validation functionality and is adequate for the Web, because it fulfils the relevant preprocessing framework requirements. Specifically, it consists of an integration model specific validation driver, which provides the core validation functionality, and a set of validation wrappers, which provide the necessary interface to the transformation model and document authors.

The validation driver enables the use of independently developed validation semantics to validate mixed namespace documents that use an open set of languages. It performs a preorder validation of a document's subtrees. The subtrees are validated by atomic validations that implement the common atomic validation interface. Such a common interface enables the integration of a multitude of current and future validation technologies. The subtree based validation is equivalent to the integration model definition of valid documents, under a well defined set of assumptions. The well defined assumptions assist the independent development of validation specifications that are adequate for the proposed processing, because it is sufficient that they are obeyed by the semantics authors. Furthermore, they provide a well defined method to denote the places where content is expected by the introduction of the `coc` construct. Such a method is necessary because the validation driver adopts the explicit method of identifying the places where content is expected.

Three validation wrappers provide the necessary validation functionality to the transformation model and the document author. Specifically, the `validateSubtree` and `validateHC` wrappers perform the deep subtree and handled construct validation processing steps, which are required by the transformation pipeline driver. They both delegate the validation requests to the validation driver, which provides both types of validation. The authoring validation wrapper is more substantial and it evaluates the semantic correctness of a document by firstly transforming its *SMC* subtrees and subsequently validating the transformation result.

Chapter 9

XMLPipe binding model

All introduced XMLPipe sub-models defined a corresponding set of processing semantics and used URI-based associations to define the conceptual organisation of their semantics. This chapter describes the XMLPipe binding model, which maps the conceptual semantics organisation to their physical organisation and distribution.

9.1 Binding considerations

The binding model interface consists of the set of semantics access functions that have been defined by the previous sub-models. Specifically, the integration model introduced five access functions: $langConstructs^{HC}$, $langConstructs_e^{HC}$, $langConstructs_a^{HC}$, COC, SMC and FOC. The adaptation model introduced the *locateTermSem* function, which maps the qualified name of an adaptation term to its type. The transformation model introduced the *locateHCTS* function, which maps the qualified name of a handled construct to its corresponding transformation semantics. Finally, the *locateVS* function maps a language's URI to a set of alternative validation semantics. An adequate binding model must encapsulate the physical organisation and distribution of the processing semantics, under the interface of these functions.

The physical organisation and distribution of processing semantics are interrelated but separate problems. The former specifies the information structure within a physical medium. The latter defines their distribution into multiple physical media and the corresponding location and retrieval mechanisms. A binding model must address both problems, in order to ensure a well defined way to retrieve the processing semantics. Moreover, since the semantics authors must be able to provide semantics specifications, they also require well defined semantics distribution.

Existing semantics organisation approaches are either document-based or URI-based. Document-based organisation proposals, such as the DTD declarations and the `xml-stylesheet` processing instruction, are not adequate for XMLPipe. They share the same drawbacks as profile based integration approaches, because they require predefined integration of all document languages and impede the liberal integration of XML languages into mixed namespace documents. URI-based approaches, such as the XML Schema instance attributes and the NRL schema associations, are more adequate. Specifically, all XMLPipe resources, such as handled constructs and adaptation terms, are uniquely identifiable by URIs. Furthermore, the explicit association between such resources and their processing semantics assists the liberal semantics composition, for processing mixed namespace documents. Consequently, the XMLPipe binding model

must provide a URI-based organisation of the processing semantics.

Existing distribution methods can be classified into three categories: document based, local repository based and URI based. Document based approaches, such as the DTD and the XML Schema document links, are not adequate for XMLPipe, because they force the document author to provide document processing semantics. Additionally, local repository methods are not adequate for an open set of languages and semantics, because they cannot ensure the wide availability of a continuously evolving set of languages and semantics specifications. However, local repositories can be the basis of a secondary location mechanism, because they can allow more efficient retrieval of the commonly used processing semantics. URI-based distribution methods are adequate for XMLPipe, because they ensure the wide availability of processing semantics and allow the author of a resource to define its authoritative interpretation. Consequently, an adequate binding model must define a URI-based semantics distribution and it may include a secondary location mechanism that is based on local repositories.

The RDDL[BB02] approach provides a well defined URI-based mechanism for distributing any type of human and machine oriented semantics. Specifically, an XHTML document that corresponds to the URI of a resource can contain the necessary human oriented semantics. Such a document can also contain multiple RDDL links to the corresponding processing semantics. Each RDDL link has a "nature" and "purpose" that denote the type of the corresponding semantics. RDDL does not restrict the set of valid "natures" and "purposes"; consequently, it allows the description of all necessary XMLPipe semantics. Therefore, a binding model can use RDDL links for distributing and locating its processing semantics.

9.2 Semantics organisation

The organisation of the XMLPipe semantics must allow the coexistence of all introduced semantics specifications. However, it must not significantly restrict their physical representation, because each distribution method may require custom semantics representations.

Figure 9.1 illustrates the proposed semantics organisation, which consists of a list of semantic links. Each semantic link maps the URI of a resource to its corresponding semantics. The *nature* attribute can take one of four predefined values that correspond to the four types of XMLPipe processing semantics. Such links can be represented by several technologies, such as RDDL links and RDF associations.

Each URI can be associated with multiple transformation, validation and adaptation term semantics, but with only a single handled constructs specification. The handled construct specifications must be unique, because the handled constructs of each presentation language must be unambiguously defined. In contrast, each URI can be associated with a *list* of validation, transformation and adaptation semantics. The validation semantics consist of a list of atomic validations, because there can be multiple atomic validations for each presentation language. Both adaptation and transformation semantics require an additional association level, in order to identify the local adaptation term names and the language handled constructs, respectively.

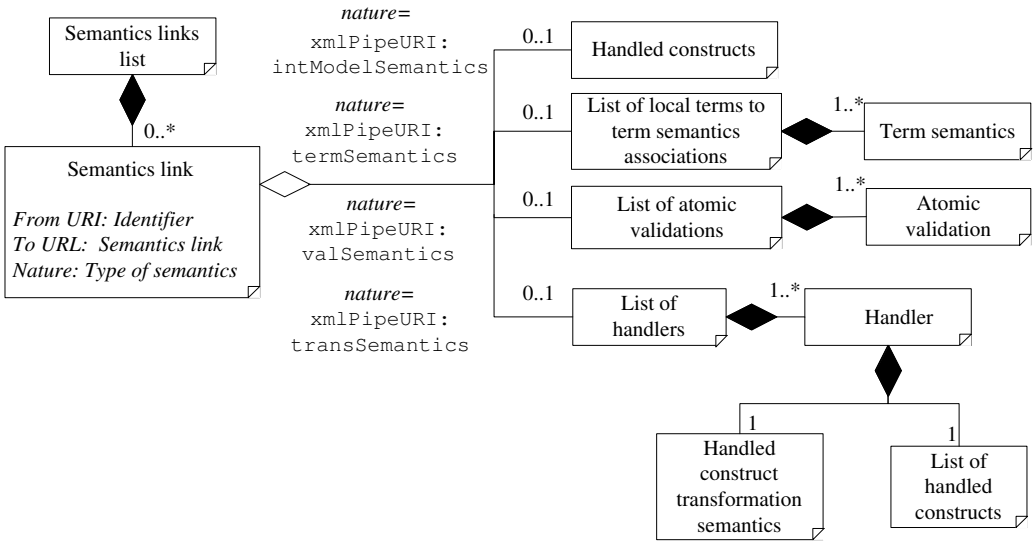

Figure 9.1: XMLPipe semantics organisation

9.3 Semantics distribution

A semantics distribution method must define how to locate the necessary semantics, how to orchestrate the several location mechanisms and how to map the semantics organisation to their physical representation. The XMLPipe binding model combines a well defined location mechanism with an open set of secondary location mechanisms. A caching process uses a set of trust levels to orchestrate the multiple location mechanisms.

9.3.1 Principal location mechanism

The principal location mechanism is based on the introduction of embedded RDDL links into the Web page that corresponds to each resource's URIs. Figure 9.2(a) illustrates

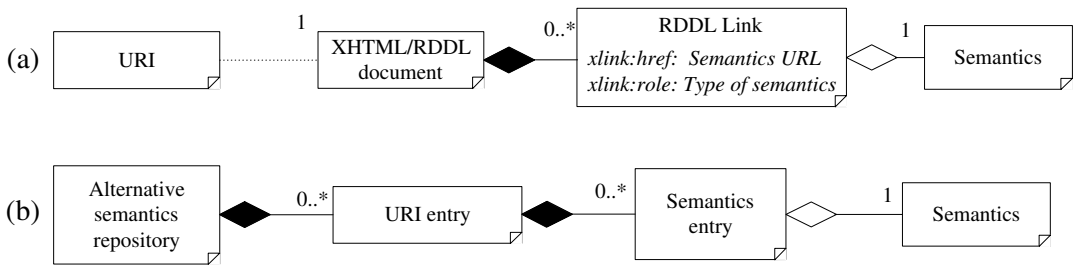

Figure 9.2: Location mechanisms information organisation for the (a) principal and (b) secondary location mechanisms

the corresponding physical semantics representation. A URI that points to an RDDL enabled Web document provides a well defined link between itself and the corresponding resource's RDDL description. Each RDDL link represents a semantics link (described in the previous section) and uses the predefined XMLPipe nature URIs to differentiate between the four types of XMLPipe processing semantics. The RDDL links do not specify the "*From URI*" attribute, because it is the same as the Web page's URL.

The principal location mechanism allows the straightforward specification and location of both processing and human oriented semantics, for an open set of presentation languages. Specifically, the author of resource, such as an XML language, can incorporate RDDL processing semantics links into the Web page that describes the resource. A human entity who knows the resource's URI, can use it to view the descriptive Web page in a Web browser. In a similar manner, a processing model can use the resource's URI to retrieve all RDDL processing semantics links. The linked processing semantics express the resource's authoritative interpretation.

9.3.2 Secondary location mechanisms

A generic preprocessing model benefits from a set of secondary location mechanisms, as described in Section 4.4. Firstly, the combination of multiple location mechanisms can eliminate central points of failure. Secondly, secondary location mechanisms assist the independent definition of processing semantics, because a semantics author is not necessarily related to the author of a resource, who is responsible for its associated RDDL links. Finally, Web pages that correspond to existing XML languages do not include XMLPipe specific RDDL links. Secondary semantics repositories can assist the initial adoption of a preprocessing approach, such as XMLPipe, by providing the corresponding processing semantics. Consequently, the XMLPipe binding model must allow the incorporation of secondary location mechanisms.

The XMLPipe binding model solely defines the high level semantics representation, in order to avoid restricting the secondary location mechanisms. The details of the interoperation between the binding model and the semantics repositories depend on the individual repositories and are delegated to the binding model implementations. Examples of secondary semantics repositories are a local semantics database and a remote Web-based semantics repository. A binding model implementation can use database specific functionality to access the former and a Web service[1] based interface to access the latter.

Figure 9.2(b) illustrates the semantics organisation within a secondary semantics repository. Secondary semantics repositories can cache semantics specifications of several other repositories. Therefore, they can contain multiple entries that correspond to the same pair of URI and nature. Consequently, in addition to the initial semantics organisation, illustrated in Figure 9.1, the *Semantics entry* information is necessary for differentiating between semantics specifications that are associated with the same URI and nature pair.

[1]Web services are the "programmatic interfaces" that enable application to application communication, within the Web. Information about the W3C Web services activity can be found in: `http://www.w3.org/2002/ws/`.

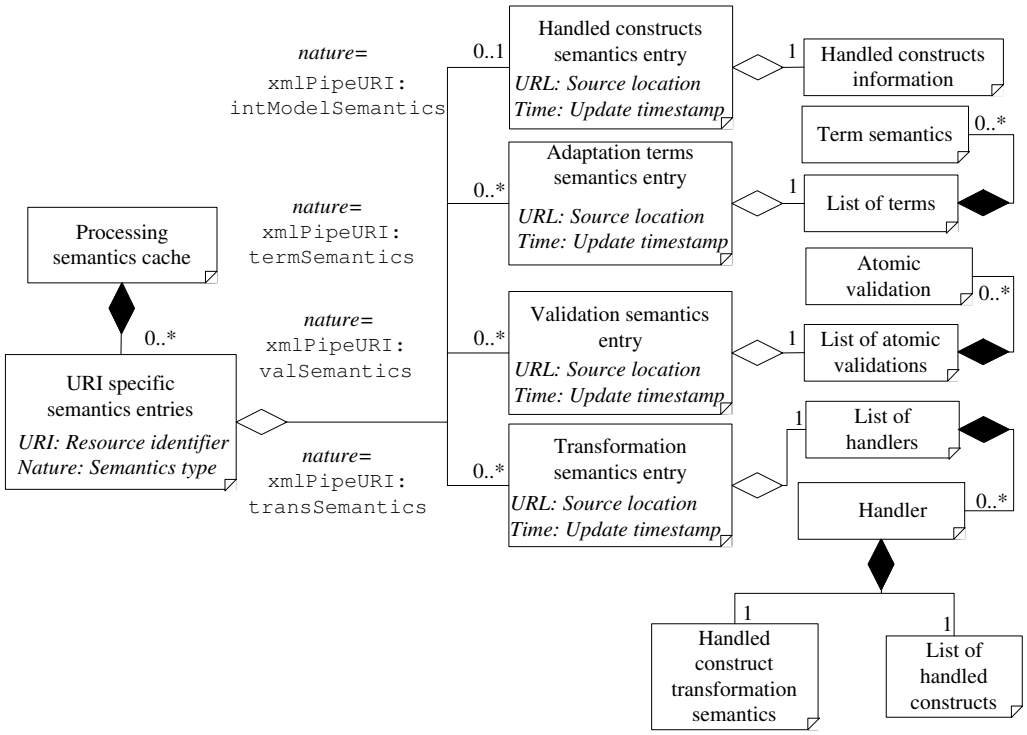

Figure 9.3: Semantics cache physical representation

9.3.3 The semantics cache

The XMLPipe binding model includes a local semantics cache that accelerates the process of semantics retrieval, during the processing of a document. It minimises duplicate remote retrievals by caching the recently used semantics.

Figure 9.3 illustrates the semantics cache organisation. It follows the secondary repository organisation, illustrated in Figure 9.2(b), because the semantics cache can be considered as a secondary semantics repository. For each cached pair of a URI and processing semantics type there is a semantics link that points to one or more semantics entries. The semantics cache utilises the "semantics entry" field to track the original semantics source and keep the the cache up-to-date.

An unambiguous set of processing semantics, which does not include conflicting or redundant entries, is essential for well defined document processing. The processes that insert or update the cached data are responsible for resolving any semantics conflicts and omitting duplicate entries. Specifically, there can be a single handled construct information entry for each URI, because each presentation language can only have one specification of its handled constructs. The adaptation term types must also be unambiguously defined, but different terms of the same namespace can be defined in separate semantics repositories. Therefore, the cache organisation allows more than one adaptation term entries for each URI. Additionally, multiple transformation and validation entries can exist for each URI, because each language and handled construct can be associated with several alternative transformation and validation semantics.

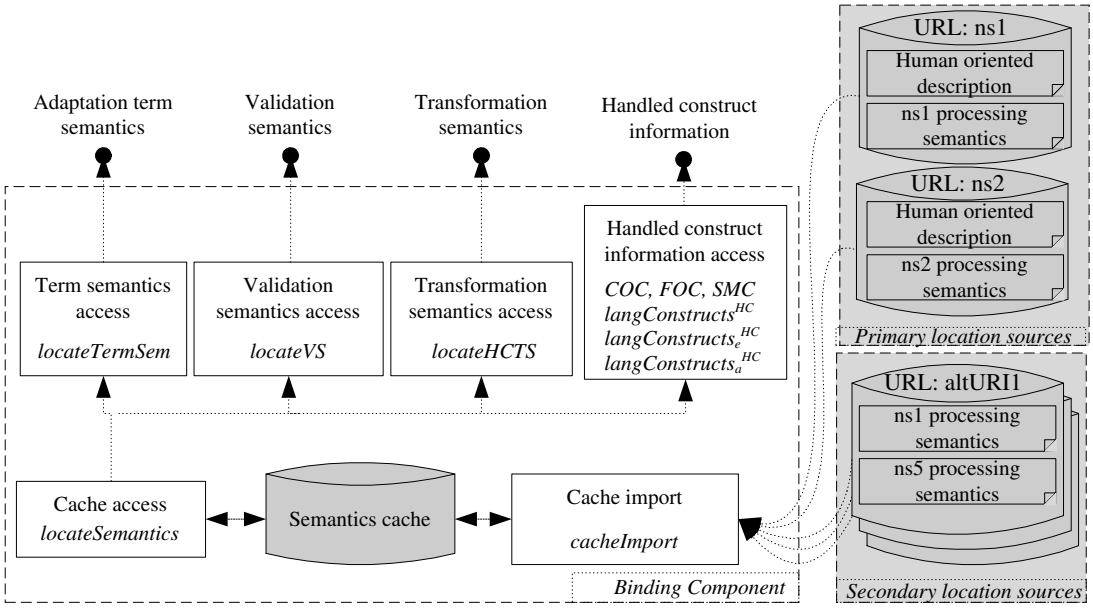

Figure 9.4: XMLPipe binding model

9.3.4 Orchestrating the location mechanisms

The XMLPipe binding model combines all remotely retrieved processing semantics into the semantics cache and uses the cached information to fulfill all processing semantics requests. Figure 9.4 illustrates the XMLPipe binding model. *Cache import* manipulates the *semantics cache* by implementing the location mechanisms and resolving any conflicting semantics specifications. *Cache access* is responsible for accessing the semantics cache and notifying *cache import*, when additional processing semantics are required. The remaining four access components use *cache access* to implement the binding model interface.

The *cache access* component implements the *locateSemantics* function, which maps a pair of a URI and a type of processing semantics to the corresponding cached semantics entries. If no relevant entries exist, *locateSemantics* calls *cacheImport* (cache import component), to ensure that all relevant processing semantics have been imported into the cache.

The *cache import* component implements the *cacheImport* function, which is responsible for orchestrating the location mechanisms and resolving conflicting semantics specifications. *cacheImport* uses a predefined set of secondary repositories and a corresponding set of trust levels. Both lists can be specified by either the preprocessing initiation entity or the default configuration of an XMLPipe implementation. Firstly, *cacheImport* creates an empty list of trust level and semantics specification pairs. Subsequently, it uses the principal location mechanism and assigns the predefined trust level of "1" to all retrieved semantics specifications. "1" is the fixed trust level of a resource's authoritative interpretation. Subsequently, *cacheImport* iterates trough all trusted secondary repositories, which are the repositories that correspond to a greater trust level than a predefined minimum value. Conflicting specifications are

Semantics location function (*locateSemantics*): The *locateSemantics* function is defined by the following algorithm:

function $locateSemantics(URI\ uri, URI\ nature) \rightarrow$
$ISemantics \cup (\Sigma \times TermSemantics) \cup$
$\wp(\wp(VSemantics)) \cup \wp(\wp((\wp(\Sigma) \times HCTSemantics)))$
let *cache* be the semantics cache
if there is no $c \in cache$ that corresponds to *uri* and *nature*
$cacheImport(uri, nature)$
end if
let $ret = \emptyset$
for each $c \in cache$ that corresponds to *uri* and *nature*
Add the processing semantics described by c to *ret*
end for
return *ret*
end function

resolved according to the trust level of the corresponding repositories: specifications that correspond to higher trust levels are always preferred. The interoperation with the secondary semantics repositories is encapsulated by function *altSem* and purposely left undefined, in order to allow an open set of secondary location mechanisms.

The binding model interface consists of four access components: *term semantics access*, *validation semantics access*, *transformation semantics access* and *handled construct information access*. It is sufficient that they return all matching cache entries, since *cacheImport* ensures that the cache has no conflicting or redundant entries. Specifically, *handled construct information access* implements the COC, FOC, SMC, $langConstructs^{HC}$, $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$ functions by returning the corresponding members of the 5-tuple *locateSemantics* return value. The *term semantics access* implements *locateTermSem* and returns the semantics entry that corresponds to the local name of the requested term. The validation and transformation semantics access components implement the *locateVS* and the *locateHCTS* functions, respectively. Both functions must combine the sets returned by *locateSemantics*, because there can be multiple validation and transformation semantics specifications for each language. In a similar manner to *locateTermSem*, *locateHCTS* function must only use the specifications that correspond to the local name of the input handled construct.

9.4 Evaluation

A binding model that is adequate for generic document preprocessing must fulfill the requirements of both the preprocessing framework and the other XMLPipe components. The latter is necessary, because the proposed preprocessing framework did not include the adaptation term and handled construct semantics. Their omission does not indicate a deficiency of either the preprocessing framework or the XMLPipe model, because they are XMLPipe specific and do not apply to all preprocessing models.

Cache import (*cacheImport*): The *cacheImport* function updates the semantics cache information, using both principal and secondary location mechanisms.

```
function cacheImport(URI uri, URI nature)
    let cache be the semantics cache
    let alt be the URI list of the alternative retrieval mechanisms
    let trust ∈ URI × ℝ be the predefined URI trust associations
    let trustMin be the minimum trust threshold
    let foundSemantics = false
    let semList = ∅ be an empty set of pairs of semantics and trust levels
    //attempt to use the principal location mechanism
    sem = RDDLParse(uri, nature)
    if sem ≠ ∅; add (sem, 1) to semList; end if
    //Use the alternative location mechanisms
    for each uri′ ∈ alt
        if ∃(uri′, x) ∈ trust AND x ≥ trustMin
            sem = altSem(uri′, uri, nature)
            if sem ≠ ∅
                if nature == intModelSemantics
                    if semList == ∅ OR ((x′, sem′) ∈ semList AND x > x′)
                        semList = ∅
                        add (x, sem) to semList
                    end if
                else if nature == termSemantics
                    add (x, sem′) to semList, where sem′ includes all term
                            semantics not declared in semList for higher priorities.
                    remove old duplicate term declarations from semList
                else if nature == valSemantics
                    add (x, sem′) to semList, where sem′ includes all atomic
                            validations, which do not have the same wrapper
                            implementation to a higher priority semList entry.
                    remove old duplicate (same wrapper implementation) atomic
                            validations from semList
                else if nature == transSemantics
                    add (x, sem) to semList
    end if; end if; end if; end for
    if semList ≠ ∅
        replace cache entry for uri and nature with semList
    end if
end function
```

Figure 9.5: Cache import algorithm

Handled construct information access functions: $\forall uri \in URI$, where $locateSemantics(uri,$ `XMLPipeURI : intModelSemantics`$) = (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5)$:
$langConstructs^{HC}(uri) = \Sigma_4 \cup \Sigma_5$, $langConstructs_e^{HC}(uri) = \Sigma_4$,
$langConstructs_a^{HC}(uri) = \Sigma_5$, $COC(uri) = \Sigma_1$, $SMC(uri) = \Sigma_2$, $FOC(uri) = \Sigma_3$

Adaptation term semantics location function: $\forall uri \in URI, s \in S$,
$locateTermSem((uri, s)) = ts_{ij}$
iff $locateSemantics(uri,$ `XMLPipeURI : termSemantics`$) =$
$\{\{(s_{11}, ts_{11}), \ldots, (s_{1n_1}, ts_{1n_1})\} \ldots, \{(s_{k1}, ts_{k1}), \ldots, \{s_{kn_k}, ts_{kn_k}\}\}$ and $s_{ij} = s$.
Otherwise, $locateTermSem((uri, s)) = \epsilon$

Transformation semantics location function:
$\forall \sigma = (uri, s) \in \Sigma$, where $locateSemantics(uri,$ `XMLPipeURI : transSemantics`$) =$
$\{QH_1, \ldots, QH_n\} \in \wp(\wp((\wp(\Sigma) \times HCTSemantics)))$,

$$locateHCTS(\sigma) = \bigcup_{i \in [1,n]} \{hcts : (\sigma, hcts) \in QH_i\}$$

Otherwise, if $locateSemantics(uri,$ `XMLPipeURI : transSemantics`$) = \emptyset$,
then $locateHCTS(\sigma) = \emptyset$

Validation semantics location function: $\forall uri \in URI$,
if $locateSemantics(uri,$ `XMLPipeURI : valSemantics`$) = \{VS_1, \ldots, VS_n\} \neq \emptyset$,

$$locateVS(uri) = \bigcup_{i \in [1,n]} VS_i$$

otherwise, $locateVS(uri) = \emptyset$

The proposed binding model provides access to all necessary processing semantics, and it covers the semantics access requirements of both the preprocessing framework and the other XMLPipe sub-models. Specifically, the implementations of the *locateHCTS*() and *locateVS*() functions provide the necessary validation and transformation processing semantics. The handled construct information and adaptation term functions cover the requirements of the XMLPipe adaptation and integration model, respectively. Additionally, they follow the same binding principles as the validation and transformation functions, in order to allow the processing of an open set of languages, according to an open set of adaptation requirements.

The preprocessing framework requires that both semantics organisation and principal location mechanism are URI-based. According to the proposed semantics organisation, the semantics of each resource are associated with its URI. Additionally, the URI of a resource is sufficient for locating its principal semantics using the RDDL-based principal location mechanism. Therefore, the XMLPipe binding model fulfils the organisation and primary location requirements.

Additionally, the location of human oriented resource descriptions must also be URI-based, according to the preprocessing framework. The RDDL-based principal

location fulfils this requirement, because it integrates both human oriented and machine processible descriptions under a resource's URI. It is sufficient that a person uses the URI of a resource within any Web browser to retrieve its human-oriented description.

The preprocessing framework requires the use of processing and integration model identifiers, within the processing semantics associations. Such identifiers enable the coexistence of processing semantics, for multiple processing models. The proposed semantics organisation does not use processing model identifiers, because all semantic links use URI qualified natures that also uniquely identify the XMLPipe processing model. Furthermore, no integration model identifier is necessary, because the XMLPipe processing model is tightly coupled with the XMLPipe integration model. Consequently, the proposed binding model does not use the required identifiers, but the URI qualified natures are adequate alternatives.

A binding model that is adequate for the Web must follow the decentralisation principle and not rely on centralised semantics repositories. The *cacheImport* function allows the combination of an open set of location mechanisms that can use multiple semantics repositories. Multiple repositories allow document processing when the principal location mechanism is unavailable or inadequate, such as when a resource's author does not provide RDDL links to the corresponding processing semantics. Furthermore, they ensure that semantics authors can define a resource's processing semantics independently of its author.

The preprocessing framework requires that no inline processing information is necessary, within a presentation document. The proposed organisation and principal location mechanisms ensure that a construct's URI is sufficient for locating all necessary processing semantics. Therefore, the XMLPipe binding model enables document processing without requiring any inline processing information.

The binding model must assist the timely execution of document processing, in a similar manner to all other sub-models. The XMLPipe transformation and validation models can require several semantics specifications, in order to process a presentation document. For instance, a document's transformation requires a set of adaptation term semantics, the handled construct information for each document language, the transformation semantics for each document handled construct and, possibly, a set of validation semantics. The distributed nature of the proposed location mechanism can introduce considerable processing delays, because most such processing semantics must be remotely retrieved. The proposed local semantics cache addresses this problem, because once the necessary processing semantics are cached, remote retrieval is no longer necessary.

Summarising, the proposed binding model is adequate for a generic preprocessing model, because it fulfils the requirements of both the preprocessing framework and the other preprocessing sub-models.

9.5 Summary

This chapter described the XMLPipe binding model, which links the XMLPipe processing components to their corresponding distributed processing semantics. It encapsulates the physical organisation and distribution details under the interface of the semantics location functions.

Both proposed semantics location and organisation are URI-based. The semantics organisation is URI-based, because all XMLPipe processing semantics describe URI

identifiable resources. The principal semantics location uses the URI-based RDDL approach, because it provides a straightforward method to publish and locate the authoritative processing semantics, for an open set of languages.

The XMLPipe binding model combines the principal RDDL-based location mechanism with an open set of secondary location mechanisms. The secondary location mechanisms are essential for avoiding central points of failure and ensuring the independence of the semantics definition processes. Specifically, semantics authors that are not also the authors of a resource, such as language authors, can use secondary semantics repositories to specify its processing semantics. Furthermore, secondary location mechanisms ensure the semantics availability, when the principal RDDL-based mechanism is either unavailable or inadequate. A predefined set of trust levels is used to orchestrate the location mechanisms and produce a concise set of processing semantics, which is stored in a local cache. The set of access components that provide the binding model interface use the local cache to serve the requests of all XMLPipe sub-models.

The proposed binding model is an adequate foundation for generic document processing, because it can support an open set of languages without requiring document specific processing information. Furthermore, it encapsulates the physical semantics organisation and distribution, under the well defined interface of the predefined location functions.

The combination of the XMLPipe binding sub-model with the previously introduced sub-models covers all XMLPipe preprocessing functionality. The next chapter will introduce the necessary components to combine all sub-models into the complete XMLPipe model.

Chapter 10

The complete XMLPipe model

The sub-models introduced in the previous chapters described all aspects of XMLPipe preprocessing, but a set of additional components are necessary for completing the XMLPipe model. The core preprocessing functionality is covered by the validation and transformation sub-models, which use the proposed integration model to process a presentation document, according to the independently developed semantics of its languages. The adaptation model provides the necessary functionality for composing adaptation profiles and selecting the optimal transformation specification, for each document subtree. The binding model bridges all sub-models with their corresponding processing semantics. The necessary additional functionality must cover document parsing and the interface to the preprocessing entities. The former has not been covered by the previous sub-models, because parsing is already well defined for all XML documents. The latter is necessary for orchestrating the parser and the individual sub-models, in order to fulfill the preprocessing entity requests.

This chapter composes the complete XMLPipe model, illustrates the feasibility of generic preprocessing and supports our hypothesis. Specifically, Section 10.1 introduces the necessary additional processing components and combines them with the introduced sub-models into the complete XMLPipe model. Sections 10.2 and 10.3 focus on implementing the XMLPipe preprocessing model and describe the core implementation issues and a pilot XMLPipe implementation, respectively. Section 10.4 illustrates the feasibility of generic document preprocessing by describing a comprehensive case study that uses the XMLPipe pilot implementation to process a presentation document, according to three separate adaptation profiles. Finally, Sections 10.5 and 10.6 illustrate the feasibility of generic document preprocessing and the soundness of our hypothesis, by investigating the case study and the complete XMLPipe model.

10.1 Composing the XMLPipe model

The XMLPipe sub-models defined all preprocessing functionality, but not the interface to the external entities which interact with a preprocessing model. This section will investigate the necessary interactions and introduce the required processing components for completing the XMLPipe model.

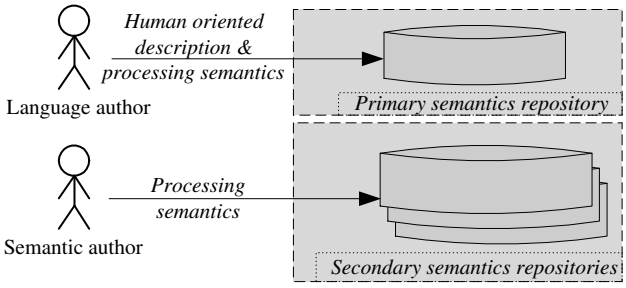

Figure 10.1: Semantics definition process

10.1.1 Interface to the semantics and language author

The language and semantics authors define all processing semantics. However, they only interact with a preprocessing model indirectly through the principal and secondary semantics repositories, as illustrated in Figure 10.1. The XMLPipe binding model has defined the processing semantics organisation, for all semantics repository types, and the principal location mechanism details. In contrast, it has purposely left the secondary location mechanisms undefined, in order to avoid constraining them. Therefore, no additional processing components are necessary and the proposed binding model is sufficient for the semantics definition process.

10.1.2 Interface to the document user and author

The document users and authors directly interact with a preprocessing model. The document user can use XMLPipe to transform a document to its optimal interpretation, according to a composite adaptation profile. The document author can use XMLPipe to perform authoring validation, in order to ensure the semantic validity of a document. The proposed XMLPipe sub-models covered all necessary processing functionality, apart from document parsing and the validation/transformation interfaces to the document author/user.

Document parsing is well defined for all XML documents, as described in Section 2.1.1 (page 13). Consequently, XMLPipe can use any standards compliant XML parser. Function *parse* maps the URL of a document to its corresponding DOM tree, and it represents the interface of standards compliant DOM parsers.

DOM parser (*parse*): Function $parse : URI \rightarrow D$ represents the interface of a standards compliant DOM parser, and it maps a URI *uri* to its corresponding DOM representation $d \in D$.

The *validation driver* component provides the preprocessing interface to the document authors. It implements the *XMLPipeVal* function, which drives the authoring validation of a document that corresponds to an input URL. Firstly, *XMLPipeVal* uses the document parser to retrieve the DOM tree of the specified URL. Subsequently, it uses the authoring validation driver (*validateAuth*), to perform the authoring validation of the parsed document.

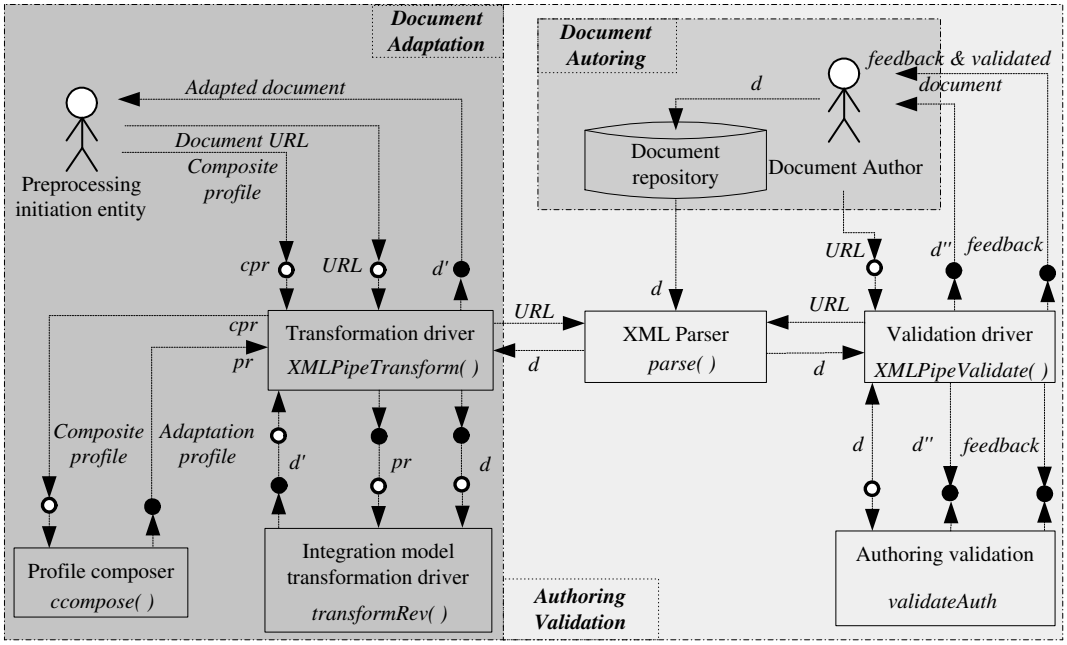

Figure 10.2: Document transformation and authoring validation

XMLPipe validation driver (*XMLPipeVal*): $XMLPipeVal : URI \rightarrow D$ is a function that represents the authoring validation XMLPipe interface. For a $uri \in URI$, $XMLPipeVal(uri) = d'$ where $parse(uri) = d$ and $d \stackrel{validateAuth}{\longrightarrow} d'$

The *transformation driver* provides the XMLPipe interface to the document users. The transformation driver implements the *XMLPipeTrans* function, which is responsible for transforming the document to its optimal interpretation, according to a composite adaptation profile. Firstly, *XMLPipeTrans* uses the parser to parse the input document and, subsequently, uses the integration model transformation driver (*transformRev*) to adapt the parsed document. Prior to the document transformation, it also interoperates with the adaptation model's profile composer (*ccompose*), in order to convert the input composite profile to an adaptation profile.

XMLPipe transformation (*XMLPipeTrans*): $XMLPipeTrans : URI \times CProfiles \rightarrow D$ represents the XMLPipe transformation interface. For a URI $uri \in URI$ and a composite profile $cpr \in CProfiles$, $XMLPipeTrans(uri, cpr) = d'$ where $parse(uri) = d$, $ccompose(cpr) = pr$ and $transformRev(d, pr) = d'$.

Figure 10.2 illustrates the interaction between XMLPipe and its external entities. Initially, the document author creates a document d and associates it with a URL, by adding it to a document repository, such as a Web server or a local storage medium. The document author can provide the URL of the document to the validation driver, in order to request its authoring validation. The validation driver interacts with the XML parser to retrieve the DOM tree d and, subsequently, delegates the validation request to the authoring validation component. The document user can request the

transformation of a document by providing its URL and a composite adaptation profile to the transformation driver. The transformation driver firstly uses the XML parser, to retrieve the DOM tree that corresponds to d. Subsequently, it uses the profile composer to convert the composite profile cpr to an adaptation profile pr. Finally, it delegates the transformation request to the integration model transformation driver and returns the transformed document d' to the document user.

10.1.3 The complete XMLPipe model

The complete XMLPipe model consists of the aforementioned transformation and validation drivers, the document parser and the previously introduced XMLPipe submodels. Figure 10.3 illustrates the complete XMLPipe model. An external entity can initiate the processing of a document by interacting with either the transformation driver or the validation driver. Subsequently, the driver components orchestrate the corresponding process and return any feedback to the preprocessing initiation entity.

Upon a transformation request, the transformation driver firstly uses the document parser to map the input URL to a DOM document tree d. It also uses the profile composer to map the composite profile cpr to an adaptation profile pr. Subsequently, it delegates the transformation request to the integration model transformation driver, which is responsible for transforming the parsed document d, according to the adaptation profile pr. The remainder of the transformation process consists of a postorder separation and transformation of all document subtrees. For each handled construct σ_i, the integration model transformation driver interacts with the binding component, in order to locate the corresponding set of transformation semantics. The returned transformation semantics consist of several transformation pipelines p_{ij}, which are associated with corresponding binding adaptation specifications B_{ij}. Each B_{ij} consists of an optional applicability expression and a set of adequacy expressions. The adaptation measure evaluator assigns an adaptation measure to each specification B_{ij}, according to the adaptation profile pr. The integration model transformation driver uses the assigned measures to identify the optimal transformation pipeline specification, for each document subtree. After the transformation of all d subtrees, the integration model transformation driver returns the resulting document d' to the transformation driver.

The transformation pipeline driver is responsible for the transformation of each document subtree. Specifically, for each transformation pipeline specification, it combines the specified set of atomic transformations to create the corresponding transformation pipeline. The created transformation pipelines map each document subtree d_i to its corresponding transformation result d'_i. If a pipeline specification requests subtree validation, the pipeline driver uses the `validateHC` and `validateSubtree` functions to interoperate with the validation model.

Upon a validation request, the validation driver uses the XML parser to map the input URL to the corresponding DOM document d and delegates the validation request to the authoring validation component. Subsequently, the authoring validation component produces d' by interacting with the integration model transformation driver, which eliminates as many *SMC* rooted subtrees as possible. Specifically, the integration model transformation driver applies a postorder separation and transformation of all *SMC* handled construct rooted subtrees, according to their associated transformation pipeline specifications. Subsequently, the authoring validation driver uses the `validateSubtree` function to perform a deep subtree validation of d'.

Both `validateHC` and `validateSubtree` functions delegate the validation requests

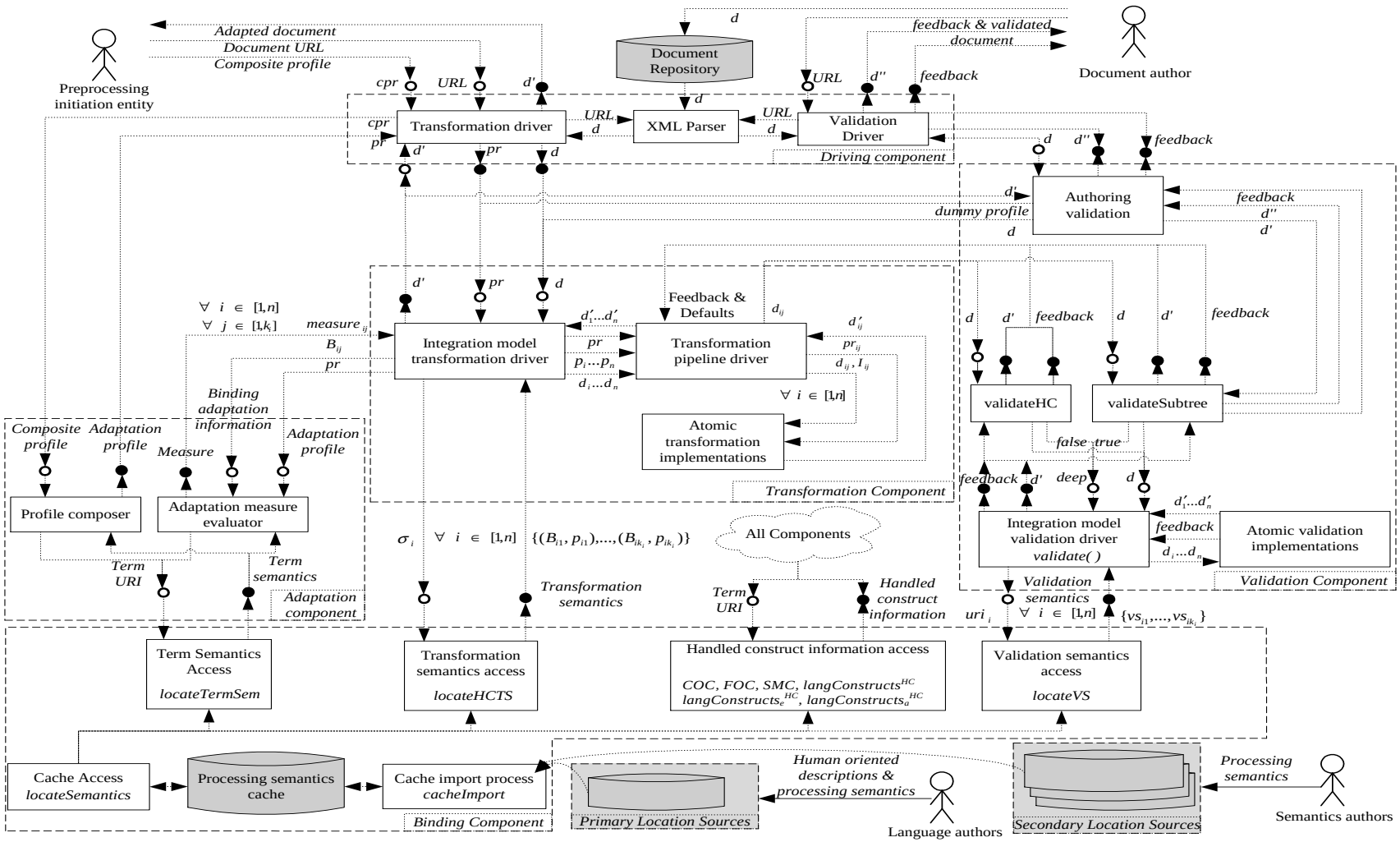

Figure 10.3: The XMLPipe preprocessing model

to function *validate*, which is implemented by the integration model validation driver. As opposed to the integration model transformation driver, function *validate* separates and processes the document subtrees in a preorder manner. It separates document subtrees, if they are rooted at a handled construct that belongs to a separate namespace than their parent. For each subtree belonging to a namespace uri_i, the integration model validation driver uses the binding component to locate the corresponding collection of validation specifications $vs_{i1}, \ldots, vs_{ik_i}$. Subsequently, it chooses the most appropriate specification, in an implementation specific manner. Afterwards, it creates the atomic validation that corresponds to the chosen specification and validates the document subtree.

Both the authoring validation and transformation processes require access to the XMLPipe processing semantics, illustrated in Figure 10.4. The binding component bridges the individual XMLPipe components with their corresponding semantics and it is responsible for their efficient retrieval. The primary location mechanism uses RDDL links, which are embedded in the Web pages that correspond to the URI of each resource. XMLPipe also allows an open set of secondary location mechanisms. When a component requests a set of processing semantics, the binding component updates its internal cache with all relevant information, resolves any conflicting entries and returns the relevant entries. The local processing semantics cache acts a local repository and, in addition to accelerating the processing, enables the processing of documents within environments that lack constant network connectivity.

10.2 XMLPipe implementation issues

The described XMLPipe model is sufficiently detailed to outline the design of an XMLPipe implementation. However, it does not cover a set of fine grained design details and implementation specific issues. This section describes the core set of implementation issues we addressed, during the development of the pilot XMLPipe implementation.

10.2.1 Presentation integration model

The initial set of transformation assumptions required that all presentation integration models are equivalent to the XMLPipe integration model. Such a requirement is not always necessary, because the transformation process of a document subtree can access all descendant subtrees and enforce their integration according to the target presentation integration model. Consequently, the revised set of assumptions loosened the integration models equivalence assumption to only require that the presentation integration model is less generic than the XMLPipe integration model (Assumption 7, Table 7.3).

Most existing presentation approaches do not support generic integration models and significantly restrict the structure of valid documents. Specifically, they can only process either single namespace documents or mixed namespace documents that use a predefined integration profile. Most languages and integration profiles specify a single document root element that introduces well defined context independent content. The corresponding valid documents can only be rooted at such an element.

Independent subtree transformations are not sufficient for adapting XMLPipe documents to such restrictive representations, unless the input documents are overly constrained. According to the XMLPipe integration model, each handled construct rooted

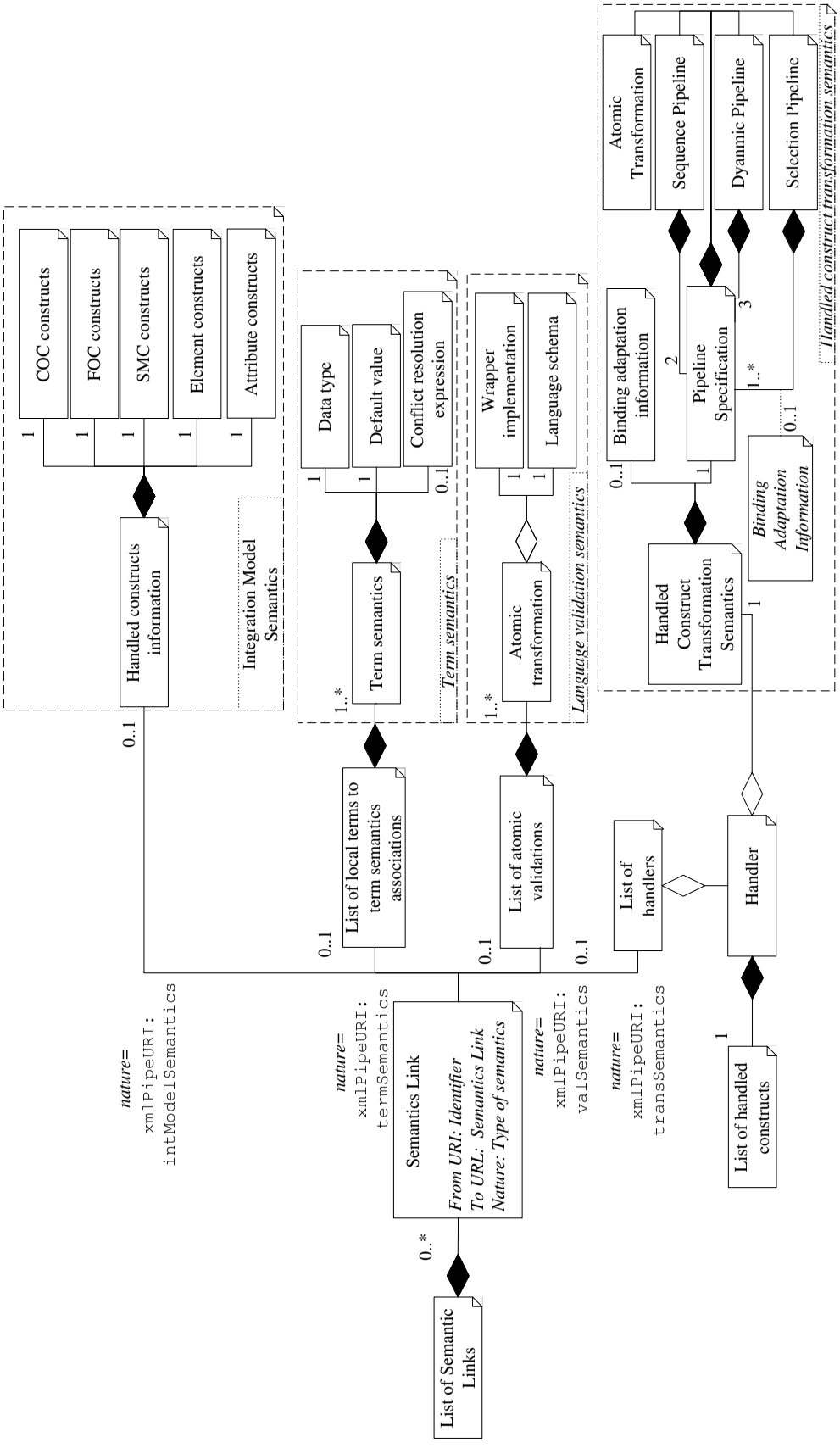

Figure 10.4: XMLPipe processing semantics

subtree d has a well defined presentation. Therefore, the result of a subtree transformation d' must also have a well defined presentation, because it is an interpretation of d. Independently of the individual transformations, the output document is only guaranteed to be valid, if the source document contains a single handled construct and its associated transformation maps it to the root construct of a natively supported language. Otherwise, the resulting document would either be rooted at not the predefined document root element or contain multiple such elements.

For instance, consider the processing of the L_{doc} constructs (illustrated in the driving example, in Section 7.2) for a browser that supports XHTML. The `html` construct is the only valid root element of an XHTML document, and it must only appear as a document's root. Consequently, the transformation semantics for L_{doc} should map all L_{doc} handled constructs (`doc:document`, `doc:p`, `doc:em`, `doc:imh`) to an `html` rooted tree. Such a mapping is necessary to guarantee that all L_{doc} handled construct rooted subtrees are mapped to valid XHTML documents. If an input document uses multiple handled constructs, such as in the driving example document, its XMLPipe processing would result in an invalid XHTML document that contains multiple `html` elements.

Well defined interoperation between the individual subtree transformers can assist the adaptation of a document for restrictive integration models, but it is not adequate for XMLPipe. For instance, interoperation between subtree transformation instances can allow them to choose between generating a document root construct or another construct, which is adequate for the integration of a subtree within its context. However, such interoperation relies on restrictive assumptions about the set of natively supported languages. Consequently, it is not adequate for a generic preprocessing model.

Without transformation interoperation there can be no validity guarantees, but the transformation specifications can be designed to produce valid documents, in the majority of integration cases. Most existing languages only allow documents to be rooted at a single predefined element. However, they contain constructs that introduce subtrees with well defined interpretation and can appear in most document places. For instance, the `xhtml:p` element always introduces a paragraph and it can appear in most places within an XHTML document. The semantics authors can utilise such language constructs, in order to define transformation specifications that result in valid documents. For instance, if the transformations of handled construct rooted subtrees map their source subtrees to target subtrees that are rooted at such constructs, the resulting document will be valid in most cases. Moreover, unsupported language combinations can be avoided, if the document users provide minimal adaptation profiles, which only include languages that can be combined within a single document. Consequently, most documents can be adapted to the current presentation integration models, if the transformation specifications utilise the design of the presentation languages and document users provide minimal adaptation profiles.

A side effect of the absence of generic presentation integration models is the necessity of `DOCTYPE` document type declarations. Generic integration models define a document's interpretation, according to its constructs. Without such models, document type declarations are necessary for defining the type of a document and linking to the corresponding syntax specifications. Therefore, in addition to creating valid presentation documents, an XMLPipe implementation must also produce document type declarations.

Choosing the optimal resulting document type and producing the corresponding declaration requires interoperation between the transformers and a fixed set of language and integration profile document types. The former is necessary, because the individual

transformers must interoperate, in order to choose a commonly supported document type. The latter is necessary for a well defined interoperation between the transformers, because the necessary form of interoperation relates to the nature of the individual document types. In a similar manner to above, such a solution is not adequate for a generic preprocessing model.

The XMLPipe pilot implementation avoids restrictive solutions, by partially addressing the `DOCTYPE` generation problem. Specifically, the transformation that corresponds to the document root handled construct has access to all transformed subtrees. Consequently, it can choose the most appropriate `DOCTYPE` declaration. Therefore, the pilot implementation allows the document root transformation to specify the type of a document. Such a solution does not allow the individual transformers to interoperate, in order to choose the optimal document type. However, it does not harm the XMLPipe's processing generality and it only requires minimal adaptation profiles. Minimal profiles are necessary for ensuring that all transformations use languages that belong to the same integration profile.

The aforementioned solutions to the presentation integration model and document type issues are temporary solutions, because they cannot guarantee optimal optimal document interpretations and the validity of the processed documents. We do not attempt to provide permanent solutions, because the current design of natively supported presentation languages is problematic. Integration profiles and the associated document type declarations are against the Web design principles, and they are not adequate for the continuously evolving set of XML languages. Consequently, it is reasonable to assume that future presentation components will adopt generic integration models and not require restrictive integration profiles. The XMLPipe transformation specifications would then be able to utilise these integration models to generate versatile mixed namespace documents.

10.2.2 Semantics representation

A stable, concise and easy to use processing semantics representation is necessary for the adoption of a preprocessing model, because it encourages the semantics authors to specify the processing semantics for a variety of XML languages. Conciseness and ease of use assist the creation of the specifications. Stability minimises the inconsistencies between separate XMLPipe implementations and ensures the wide applicability of the semantics specifications.

Notwithstanding the importance of the XMLPipe processing semantics representation, this thesis primarily focuses on the semantics organisation and the corresponding document processing. Consequently, the XMLPipe binding model established the processing semantics organisation and their distribution, but not their representation.

The pilot XMLPipe implementation uses a representation that is adequate, but not necessarily optimal. Appendix E outlines the XMLPipe pilot semantics representation and provides a reference for interpreting the subsequent semantics descriptions. The proposed representation focuses on reducing the XMLPipe implementation complexity by allowing straightforward parsing and processing. Moreover, it defines the representation of all XMLPipe semantics and it is self explanatory. However, it should not be considered as a proposal or a guide of the optimal semantics representation.

10.2.3 Node context information issues

Information on each document node's source and processing is necessary for meaningful error reporting. During the proposed document transformation, each document node might originate from a separate source and might result from multiple transformation applications. A processing error can originate in either the document sources or the transformation specifications. Consequently, an error reporting mechanism, which assists the document authoring and semantics development, must report all document sources and transformations that relate to an offending document node.

All processing components benefit from accessing such rich contextual information. Since most processing components can already access the adaptation requirements information, the XMLPipe pilot implementation reuses the adaptation profile representation to encode each node's location and processing information. Specifically, it associates each node with a *node context* specification, which consists of a set of adaptation statements that describe the node's original location and processing history. Each processing component can access a node's context specification to either report an error or to amend it.

Creating and updating node context specifications is not sufficient, because most subtree transformations discard them. Specifically, atomic transformations can use third party implementations, such as XSL-T transformers. Most non XMLPipe specific implementations do not directly modify the input subtree. Instead, they create new nodes that have no associated context information. The application of such transformations to a document subtree discards all its node context information.

The pilot XMLPipe implementation ensures that all document nodes are always associated with approximately correct error reporting information by a combination of node context serialising and cascading. Specifically, it provides utility functions that enable the transformation wrappers to serialise and parse node context information, using an XML representation. Prior to a subtree's transformation, a wrapper can serialise its context information within the subtree. The transformation will preserve most context information, because XMLPipe transformations must copy as many foreign namespace subtrees as possible to their corresponding place in the transformation output. After the transformation application, the wrapper can parse the retained context information and re-attach it to the subtree nodes. Such a context serialisation technique preserves most of the context information. The remaining nodes, inherit their parent's context information, because a node's parent is likely to have the same origin and processing. The combination of the applied node context serialisation and cascading ensures that all document nodes have approximately correct node context information.

The XMLPipe pilot implementation uses the node context information to resolve relative URLs, in addition to reporting errors. Specifically, the resolution of relative URLs depends on the initial source document of their corresponding XML construct. For instance, the URL resolver for a link can use the document source of the link handled construct. In contrast the URL resolver of an XSL-T wrapper must use the location of both the input document and the transformation stylesheet, because a URL reference can originate from both. The node context provides the necessary information to such resolvers. The XML Base recommendation[Mar01] provides similar functionality, but it is not necessary for XMLPipe, because the existing node context mechanism is sufficient.

The XMLPipe implementation also uses the node context information to keep track

Implementation issue	Resolution
Restrictive presentation integration models	Minimal adaptation profiles and semantics specifications that ensure the validity of most output documents by utilising the design of natively supported languages.
`DOCTYPE` declarations	The document root transformation can specify the output document type.
Semantics representation	Concise, easy to author and well defined. Current implementation focuses on ease of processing, and it defines an adequate, but not necessarily optimal, semantics representation.
Node location information	Association of node context information to each node, cascading location information and serialisation of location information, prior to the application of atomic transformations.
Language specific adaptation statements	Adaptation statements insertion to the node context of each subtree's parent element.
Relative URL resolution	Use the location information of the most relevant node(s).

Table 10.1: XMLPipe implementation issues

of the language specific adaptation statements. As illustrated by the *transformRev* algorithm (page 137), the integration model transformation driver keeps track of the language specific adaptation statements, which are introduced by a subtree's transformation, by associating them with the subtree's parent node n. The pilot implementation reuses the node context information, and it is sufficient to add the language specific adaptation statements to the node context information that is associated with the parent n.

10.2.4 Summary

Table 10.1 summarises the addressed implementation issues and the corresponding proposed resolutions, which enable the creation of an XMLPipe implementation.

10.3 The pilot implementation

This section describes our pilot implementation of the XMLPipe preprocessing model. The core purpose of the pilot implementation is to illustrate the feasibility of preprocessing presentation documents that use an open set of XML languages, according to a variety of adaptation requirements and independently developed processing semantics. In its current status, the pilot implementation does not contain all XMLPipe functionality, but it covers a sufficient functionality subset for fulfilling its purpose. Section 10.3.1 investigates the sufficiency of the implemented functionality subset and Section 10.3.2 outlines the pilot implementation's design. Finally, Section 10.3.3 summarises the implementation's command line interface.

10.3.1 Sufficient functionality subset

The current pilot implementation covers a subset of the XMLPipe sub-models: the integration model, the transformation model and the adaptation model. The transformation model implementation is complete and includes the proposed atomic transformations, transformation pipelines and transformation driver. The adaptation model implementation coves all adaptation functionality, but it does not incorporate the applicability and custom conflict resolution expressions. In contrast, it uses the adequacy expressions to derive the default applicability expression and always uses the default conflict resolution mechanism. The binding model implementation provides the necessary semantics location interfaces and processing semantics parsing functionality, but it currently supports only a local document secondary repository.

The implemented functionality subset is sufficient for illustrating the feasibility of the proposed XMLPipe processing. A transformation model implementation can illustrate the feasibility of orchestrating independently developed transformation semantics to transform mixed namespace documents. The XMLPipe validation model is based on the same integration principles as the XMLPipe transformation model. Therefore, the latter implicitly illustrates the feasibility of orchestrating independently developed validation semantics to validate presentation documents. Regarding the adaptation model implementation, the absence of applicability and conflict resolution expressions does not harm the core adaptation functionality, because they are only necessary when the default applicability and resolution are not adequate. Therefore, their support is beneficial, but they are not necessary for illustrating the feasibility of adaptive document transformation. Finally, the simplistic location mechanism implementation does not combine processing semantics from multiple sources. Nevertheless, if the semantics representation and organisation are location independent, they can sufficiently illustrate the feasibility of using independently developed semantics. Consequently, the partial pilot implementation fulfils its purpose, because it can illustrate the feasibility of processing presentation documents that use an open set of languages, according to a variety of adaptation requirements and independently developed processing semantics.

10.3.2 Implementation outline

An XMLPipe implementation must provide a well defined preprocessing interface to a document preprocessor of an adequate design, using an appropriate programming language.

A combination of a command line interface and a well defined preprocessing API is adequate for XMLPipe. Preprocessing approaches can be used at either the server or the client processing side. A command line interface can be used for both client and server side processing by a person or a script, respectively. In a similar manner, the preprocessing interface can be used by both a browser or a server side application.

We chose the Java programming language for the pilot XMLPipe implementation. Its wide applicability and device independence is beneficial for XMLPipe, because XML document preprocessing can be used in a multitude of environments. Additionally, the Java virtual machine enables the straightforward integration of remotely retrieved atomic transformations and validations, without requiring their pre-compilation for multiple platforms or runtime compilation. Furthermore, the existing applet security mechanisms can assist security extensions to the XMLPipe model, because they address a similar problem: execution of automatically retrieved, possibly malicious, processing

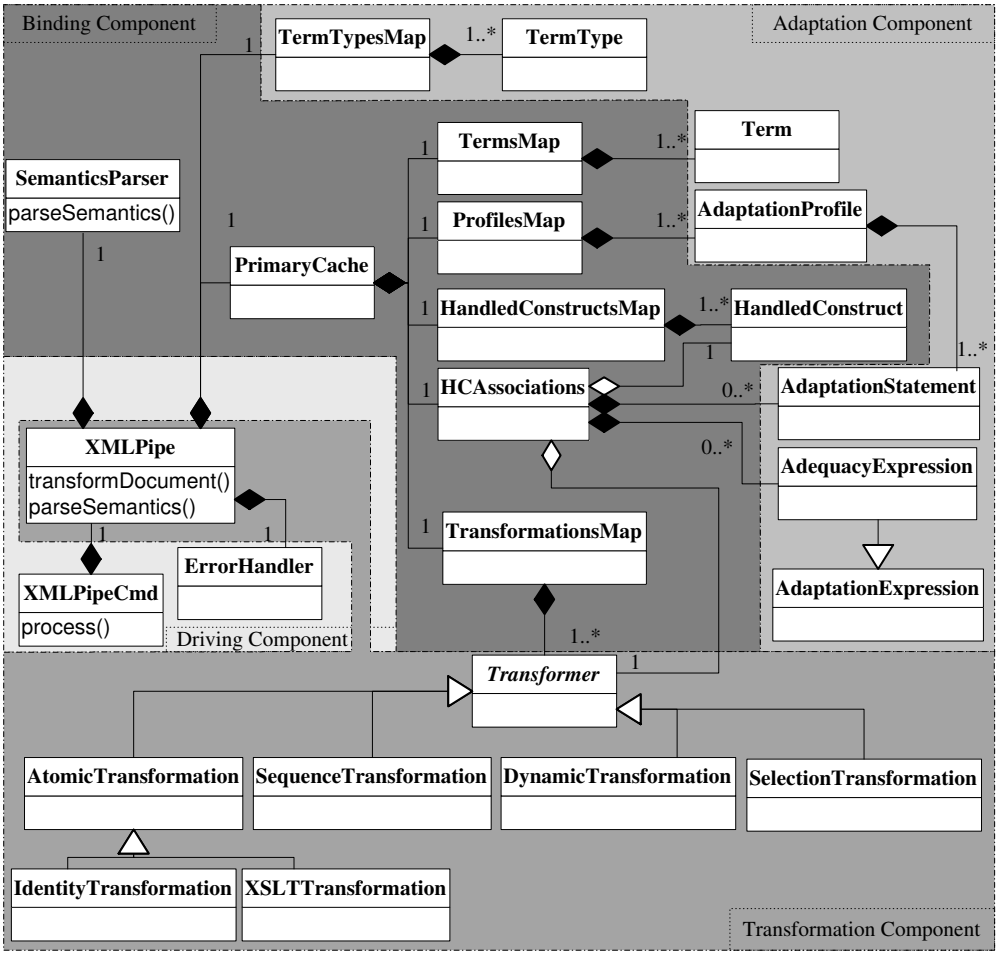

Figure 10.5: Pilot implementation class hierarchy

components.

Figure 10.5 illustrates the top level class hierarchy of the pilot implementation. The separate background compartments group the individual classes, according to their corresponding XMLPipe sub-models. The pilot XMLPipe class hierarchy follows the organisation of the processing semantics. Most classes represent a processing semantics concept and they cover both the corresponding internal information representation and the relevant functionality.

The driving component consists of the `XMLPipeCmd`, `XMLPipe` and `ErrorHandler` classes. `XMLPipeCmd` implements the transformation driver, interacts with the preprocessing initiation entity and orchestrates the use of the other classes. Upon a transformation request, it parses the input document and instantiates the `XMLPipe` class, which is responsible for performing the document transformation. Subsequently, it serialises the transformation output. Moreover, `XMLPipeCmd` initiates the parsing of all necessary processing semantics, keeps track of their internal representation and instantiates an `ErrorHandler` object. The latter controls all error and information processing messages and utilises the node context information to produce context rich error messages.

The pilot implementation contains the minimum necessary binding functionality, as described in Section 10.3.1. The implemented binding component consists of the internal processing semantics representation and the `SemanticsParser`. The latter is responsible for converting the XML processing semantics representation to the corresponding internal representation. Specifically, prior to document processing, `XMLPipe` interoperates with the `SemanticsParser` to add all available processing semantics to a `PrimaryCache` object. `PrimaryCache` contains five sets of associations that map the individual named resources to their corresponding classes. For instance, `TermsMap` maps the qualified names of all cached terms to `Term` class instances. `HCAssociations` represents the transformation handlers. Each `HCAssociations` object maps a handled construct to a transformation pipeline (`Transformer`), according an optional set of adequacy expressions. Additionally, a `HCAssociations` object can also include a set of adaptation statements that provide a declarative method for introducing temporary adaptation profile modifications, within a transformation pipeline.

The adaptation component consists of a set of classes that represent the term data types, adaptation statements and adaptation expressions. The `TermTypesMap` maps the data type names, such as `SetOfStrings`, to their corresponding subclasses (not illustrated) of `TermType`. The `TermType` class and its subclasses cover all built-in adaptation term data types. Each `TermType` subclass contains the necessary functionality for parsing, serialising and performing operations. Consequently, the `TermType` subclasses contain the core functionality for evaluating adaptation expressions. `AdequacyExpression` class, which subclasses `AdaptationExpression`, contains an `evaluate` function that uses the individual `TermTypes` to evaluate adequacy expressions, according to an adaptation profile.

Finally, the transformation model implementation consists of the driving transformation methods, included in the `XMLPipe` class, the `Transformer` abstract class and its subclasses. Specifically, the `XMLPipe transformDocument` member function implements the integration model transformation driver that is responsible for the depth first transformation of the input document. The abstract `Transformer` class contains the common interface and functionality of both atomic and pipeline transformations. Each proposed transformation pipeline corresponds to a subclass of the `Transformer` class. A transformation pipeline class is responsible for orchestrating a set of transformations, which are also instances of the `Transform` class, according to a pipeline specification. The `AtomicTransformation` class contains the interface and common functionality of all atomic transformations. For instance, the built-in XSL-T atomic transformation (`XSLTTransformation` class) inherits `AtomicTransforamtion`. `IdentityTransformation` is a predefined identity atomic transformation, which is used when a transformation pipeline requires an atomic transformation, but the processing semantics omit it.

10.3.3 The invocation

The `XMLPipeCmd` class implements the XMLPipe command line interface. The preprocessing initiation entity must specify the name of an adaptation profile and the locations of both the input document and the processing semantics file. For instance,

```
java XMLPipeCmd -c config.xml aMobile document.xml
```

will use the processing semantics in `config.xml` to transform `document.xml`, according to the adaptation profile `aMobile`. The processing semantics file specification is

necessary, because of the partially implemented binding component. Future XMLPipe extensions, which fully implement the proposed binding model, will not require such explicit semantics repository specifications.

10.4 A case study

This thesis has described the XMLPipe processing model and the XMLPipe implementation, but it has not explicitly illustrated the feasibility of the proposed XMLPipe processing. This section will use the XMLPipe pilot implementation to illustrate the feasibility of the proposed document preprocessing, by describing a case study that coves all preprocessing aspects. The pilot XMLPipe implementation covers a subset of the XMLPipe model functionality. However, the following case study will seamlessly cover all preprocessing aspects, in order to illustrate the complete spectrum of the proposed document preprocessing.

Sections 10.4.1 and 10.4.2 set the foundation of the case study by introducing the input document and a set of adaptation profiles, which enable the illustration of XMLPipe adaptation capabilities. Sections 10.4.3 to 10.4.8 describe the necessary processing semantics and their corresponding binding. Section 10.4.9 describes the combination of all processing semantics by the processing sub-models, during the authoring validation and transformation of the input document. Finally, Section 10.4.11 describes the processing of an additional XML language and illustrates how semantics authors can reuse existing constructs to simplify the processing semantics definitions.

10.4.1 The input document

The case study will reuse the transformation driving example document, introduced in Section 7.2 (page 115). The driving example document is adequate for a preprocessing case study, because it includes multiple XML languages, combines their constructs in several ways and is sufficient for illustrating all XMLPipe preprocessing functionality. Additionally, it is adequate for evaluating XMLPipe's ability to process semantically correct but invalid documents, because it combines the constructs of presentation languages in a semantically correct way, but the integration of the *SMC* constructs is invalid.

This section will outline the semantics and syntax of the document languages, which were precisely described in Section 7.2. Figure 10.1 illustrates the case study document. `document.xml` is the main input document. `authors.xml` and `imp.xml` contain the included author information and textual content, respectively. L_{doc} constructs define the layout of a document that includes a title, a set of authors and nested sections, which can contain multiple paragraphs. L_{alt} introduces adaptive content by associating a sequence of content alternatives to a set of adequacy expressions. L_{imp} allows the inclusion of arbitrary XML content, according to a URI and an XPath expression. Finally, L_{xl} represents the simple links subset of XLink. The `xl:type` attribute introduces a link and the `xl:href` attribute specifies the link's target.

Both document validation and transformation require well defined classification of all language handled constructs. Listing 10.2 illustrates the handled constructs information representation, according to their initial classification in Table 7.1 (page 115). Each language has a corresponding `constructs` element. The `constructs` elements do not have to appear in a single document, and they can occur in separate repositories

```
<doc:document>
 <alt:alt>
  <alt:case test="uri#deviceType=mobile">
   <doc:title>Mobile example</doc:title>
  </alt:case>
  <alt:case>
   <doc:title>Desktop example</doc:title>
  </alt:case>
 </alt:alt>
 <imp:import href="authors.xml" select="**/[@id='MP_DHS']"/>
 <doc:section>
  <doc:title>The doc language</doc:title>
  <doc:p>The root language allows <doc:em>emphasized</doc:em> text,
    images <doc:img href="xmlPipe.gif"/> and nested sections.</doc:p>
  <doc:section>
   <doc:title>Nested section</doc:title>
  </doc:section>
 </doc:section>
 <doc:section>
  <doc:title>Mixed namespace support</doc:title>
  <doc:p>A foreign namespace SMC construct to import textual content:
    <imp:import href="imp.xml" select="**/text/text()"/>, an FOC XLink
    attribute for <doc:em xl:type="simple" xl:href="...">links</doc:em>
    and an SMC subtree that allows adaptation sensitive content:
  </doc:p>
  <alt:alt>
   <alt:case test="http://.../#deviceType=mobile">
    <doc:p>This is a mobile</doc:p>
   </alt:case>
   <alt:case>
    <doc:p>This is NOT a mobile</doc:p>
   </alt:case>
  </alt:alt>
 </doc:section>
</doc:document>
```

document.xml

```
<c:collection>
 <doc:authors id="MP_DHS">
  <doc:author first="M" last="Ped"
      mail="mp49@kent.ac.uk"/>
   <doc:author first="D" last="Shr"
      mail="dhs@kent.ac.uk"/>
 </doc:authors>
</c:collection>
```

authors.xml

```
<root>
  <text>Text node 1</text>
  <text>Text node 2</text>
</root>
```

imp.xml

Listing 10.1: The case study input document

```
<constructs ns="doc URI">
    <hc name="document" class="coc" node="el"/>
    <hc name="em" class="coc" node="el"/>
    <hc name="img" class="coc" node="el"/>
    <hc name="p" class="coc" node="el"/>
</constructs>
<constructs ns="alt URI">
    <hc name="alt" class="smc" node="el"/>
</constructs>
<constructs ns="imp URI">
    <hc name="import" class="smc" node="el"/>
</constructs>
<constructs ns="XLink URI">
    <hc name="href" class="foc" node="at"/>
    <hc name="type" class="foc" node="at"/>
</constructs>
```

Listing 10.2: Handled constructs information

that correspond to the different languages. Each handled construct is specified by a `hc` element that specifies its classification and whether it is an element or an attribute.

10.4.2 The adaptation profiles

The preprocessing case study will use three adaptation profiles that span a broad adaptation requirements spectrum, in order to illustrate and evaluate the XMLPipe adaptation capabilities. The three adaptation profiles correspond to a desktop browser, a limited early mobile phone and a conceptual XSL-FO printer. The desktop browser profile covers the top end of the capabilities spectrum, and the corresponding document interpretation must be a rich interactive presentation. In contrast, the early mobile profile describes a device with limited presentation and interaction capabilities. The corresponding document interpretation must be simple, but retain sufficient browsing functionality. The conceptual XSL-FO printer profile covers the non-interactive set of Web devices. The corresponding document interpretation must be rich, in a similar manner to the desktop browser profile, but it must be optimised for the non-interactive printer medium.

Adaptation profiles that describe all the capabilities/preferences of the above three cases would require a multitude of adaptation statements. Nevertheless, in order to maintain a manageable case study complexity level, the corresponding specifications will only include the necessary statements for processing the example document.

The desktop browser profile states native presentation support for XHTML and binary images, because they are sufficient for a rich presentation of the example document. Listing 6.1 (page 97) contains all necessary statements for such a desktop browser profile. It states that the device type is a “desktop” and the natively supported representations include XHTML and binary media content. The optimal document interpretation for the desktop browser profile must be optimised for desktop browsing and map all document constructs to XHTML. Specifically, the resulting document must be continuous, contain all document information and use full size images. The constructs of all document languages must be mapped to their XHTML equivalents.

```
<profile name="XSLFOPrinter">
   <statm ns="XMLPipeURI/Terms"
          name="deviceType">printer</statm>
   <statm ns="XMLPipeURI/Terms"
          name="supported">
      <item>http://www.w3.org/1999/XSL/Format</item>
   </statm>
</profile>
```

Listing 10.3: Printer adaptation profile

Listing 10.3 illustrates the case study printer profile. Existing printers do not directly support XSL-FO, but the case study will use an XSL-FO representation, because it provides the necessary paged media presentation abstractions and it can be considered as a valid XML alternative to the customary printer languages. The optimal XSL-FO printer document interpretation must contain all content and images, in a similar manner to the desktop browser interpretation. However, it must be partitioned into a sequence of pages, and it must not include any interactive content, such as interactive links.

The mobile phone profile purposely describes a low-end mobile device, in order to enable the illustration of the XMLPipe adaptation capabilities. Listing 6.6(a) (page 102) has illustrated a composite profile for a low-end device, which corresponds to the adaptation profile illustrated in Listing 6.2 (page 97). The described mobile device only supports WML (Wireless Markup Language) and WBMP (Wireless Bitmap) images that are no larger than 100×96 pixels. The optimal document interpretation, for such a device, must only use the supported WML and WBMP representations and use an adequate layout for a limited display size. Furthermore, since WML 1.1 parsers rarely support XML namespaces, the document interpretation must also not contain namespace declarations.

10.4.3 Validation semantics

The XMLPipe implementation does not cover the XMLPipe validation model, as described in Section 10.3. Nevertheless, this section will describe the validation semantics, in order to illustrate their design and assist the subsequent description of validation processing.

The validation semantics consist of the necessary atomic validation declarations and schema specifications, for all document languages. A schema specification must be adequate for the proposed validation model and follow the XMLPipe validation design principles (summarised in Table 8.3, page 158). This section describes the individual schemas, but most of their corresponding listings are only included in the case study appendix.

The L_{doc} schema specification (Listing F.25, page 332) includes a separate top level element declaration for each handled construct and allows `coc` occurrences at all places where content is expected. The four top level element declarations, which correspond to the four handled constructs `doc:document`, `doc:em`, `doc:img` and `doc:p`, enable the validation of all the handled construct rooted subtrees. The
`<xsd:any processContents="strict"/>` construct is introduced at all places where

```
1  <?xml version="1.0"?>
2  <xsd:schema xmlns:xsd="XML Schema URI"
3              xmlns:ns="XLink URI"
4              targetNamespace="XLink URI"
5              elementFormDefault="qualified">
6    <xsd:attribute name="type" type="xsd:string"
7                   fixed="simple"/>
8    <xsd:attribute name="href" type="xsd:string"/>
9
10   <xsd:element name="foc">
11    <xsd:complexType>
12      <xsd:attribute ref="ns:type" use="required"/>
13      <xsd:attribute ref="ns:href" use="required"/>
14    </xsd:complexType>
15   </xsd:element>
16 </xsd:schema>
```

Listing 10.4: L_{xl} schema specification

content, is expected, which are within the `doc:p` and `doc:em` elements. The `any` XMLSchema element is used, because it allows valid occurrences of both the L_{doc} handled constructs and the predefined `coc` element, which is introduced by the subtree separation process at all places that foreign namespace *COC* constructs occur. The schema specification is independent of the other languages and, apart from the top level declarations and the `any` element, does not differ from a non XMLPipe-specific schema.

L_{imp} contains a single handled construct that does not accept any child elements. Consequently, the corresponding schema (Listing F.13, page 323) consists of a single element declaration.

The schema specification for language L_{alt} (Listing F.19, page 326) also contains a single top level declaration that corresponds to the `alt` handled construct. `alt` can contain an arbitrary number of `case` elements, which can contain *COC* child elements. Therefore, the `any` XMLSchema construct is used within the `alt:case` declaration, in a similar manner to the aforementioned `doc:p` and `doc:em` declarations.

The L_{xl} schema specification, illustrated in Listing 10.4, declares two *attribute FOC* handled constructs under the predefined `foc` element. The XMLPipe validation model separates all attribute *FOC* handled constructs from their context and adds them to a root `foc` element. Consequently, the handled constructs declaration within `foc` is adequate for validating the two L_{xl} attributes.

The proposed separation of attribute handled constructs is beneficial, because it enables their validation. Regarding the L_{xl} attributes, an element can contain either both or none of them. Simple top level declarations of the attributes cannot express such relationships. In contrast, their inclusion within the predefined `foc` element enables the precise definition of their relationship. Additionally, if XMLPipe used the actual parent element of the attributes, instead of the predefined `foc` element, the schema specification would have to include a top level wildcard declaration, which is not always feasible. For instance, XML Schema does not allow top level `xsd:any` declarations.

A language's validation semantics must also associate its URI with the corresponding schema specifications. Listing 10.5 illustrates the association of L_{doc} to its schema

```
1  <config xmlns="XMLPipeURI" name="DOC_validation">
2    <validators>
3      <validator
4          name="XMLPipe_Validators_XMLSchema"
5          implClass="..."
6          implSource="SourceURL"/>
7    </validators>
8
9    <validation ns="DOC_URI">
10     <atomVal ref="XMLPipe_Validators_XMLSchema"
11              src="Schema_URL"/>
12   </validation>
13 </config>
```

Listing 10.5: L_{doc} validation semantics declaration

specification. The `validator` declaration (lines 2–7) specifies a named atomic validation implementation. The subsequent atomic validation declaration (lines 9–12) introduces the relationship between the language namespace, the atomic validation implementation and the language schema.

10.4.4 Transformation semantics: L_{alt} language

Successful document processing requires that each handled construct is associated with at least one adequate transformation pipeline, for all three adaptation profiles. The case study transformation semantics will provide all necessary pipelines for the case study profiles. However, in practice, the semantics development process is independent of the adaptation profile development. The use of adequacy and applicability expressions, as opposed to explicit adaptation profile references, allows such independence.

A single atomic transformation is sufficient for processing the constructs of L_{alt}. Listing 10.6 illustrates the L_{alt} transformation semantics declarations. The atomic transformation declaration, in line 3, introduces a Java transformer that performs direct DOM manipulation. It substitutes the `alt` element with the content of the optimal `alt:case` element, according to the adaptation requirements. In a similar manner to the validation semantics, the atomic transformation declaration specifies a name, a Java implementation class and an implementation location. Additionally, it states that the declared atomic transformation does not accept dynamic external input, which is solely used in dynamic transformation pipelines.

The `handler` element associates a set of handled constructs to a transformation pipeline. The illustrated handler has no adequacy and applicability expressions, because the proposed transformation is adequate for all adaptation profiles, since it implements the content selection internally. Attribute `hcList` includes a single `hcRef` element that corresponds to the single handled construct of L_{alt}. The `pipe` element introduces a transformation pipeline that, in this case, consists of a single transformation step, which refers to the above atomic transformation.

```
1  <config xmlns="XMLPipeURI">
2    <transformers>
3      <transformer name="XMLPipe_XPEx_ALT"
4                   implClass="....AltHandler"
5                   implSrc=".../AltHandler.class"
6                   dynamic="false"/>
7    </transformers>
8
9    <handler name="Generic_Alt_Handler">
10     <hcList ns="ALT_URI">
11       <hcRef name="alt"/>
12     </hcList>
13     <pipe>
14       <transform ref="XMLPipe_XPEx_ALT"
15                  src=""/>
16     </pipe>
17   </handler>
18 </config>
```

Listing 10.6: L_{alt} transformation semantics declarations

10.4.5 Transformation semantics: L_{imp} language

L_{imp} transformation semantics also require a single transformation pipeline, because L_{imp} content inclusion is adaptation requirements independent. The transformation of the `imp:import` element must substitute itself with the referenced document's portion that is specified by the `select` attribute's XPath expression. It is beneficial to use the XMLPipe built-in XSL-T transformer to evaluate such expressions, because XSL-T has built-in XPath support.

XSL-T does not allow dynamic XPath expressions evaluation, but the XMLPipe dynamic pipelines can address this limitation. A dynamic pipeline can be used to firstly create an XSL-T stylesheet that statically contains the necessary XPath expression and, subsequently, apply it to the document subtree. Listing 10.7 illustrates the corresponding transformation declaration. The `option` element introduces a pipeline that dynamically creates an XSL-T stylesheet, which statically includes the `select` attribute's XPath expression. The last `transform` element introduces an atomic transformation that dynamically applies the generated XSL-T stylesheet to the document subtree. Consider that T_1 represents the `option` transformation and T_2 the dynamic XSL-T atomic transformation. The described transformation pipeline corresponds to $dyn(\epsilon, T_1, T_2)$, where T_2 applies the stylesheet generated by T_1. The proposed dynamic pipeline enables the substitution of a complex XPath evaluator with a minimal XSL-T stylesheet (Listing F.15, page 324).

For example, consider the `imp:import` construct in line 10 of the case study example. T_1 uses the `href` and `select` attributes to dynamically generate the stylesheet illustrated in Listing 10.8. The result of applying the generated stylesheet to the `imp:import` rooted subtree is the inclusion of the author information, as illustrated in Listing 10.9.

```
1  <config xmlns="XMLPipeURI">
2   <handler name="ImportHandler">
3    <hcList ns="Import_URI">
4     <hcRef name="import"/>
5    </hcList>
6    <pipe>
7     <dynamic>
8      <normal/>
9      <option>
10      <transform ref="XMLPipe_XSLT"
11            src=".../import.xsl"/>
12     </option>
13     <transform ref="XMLPipe_DXSLT"
14            src="XMLPipe:dynamic"/>
15    </dynamic>
16   </pipe>
17  </handler>
18 </config>
```

Listing 10.7: L_{imp} dynamic transformation pipeline

```
1  <xsl:stylesheet version="1.0">
2   <xsl:template match="n:import">
3    <xsl:copy-of
4     select="document(authors.xml)
          {**/[id='MP_DHS']}"/>
5   </xsl:template>
6  </xsl:stylesheet>
```

Listing 10.8: Dynamically generated stylesheet

```
1  <doc:authors id="MP_DHS">
2   <doc:author first="M" last="Ped"
3     mail="mp49@kent.ac.uk"/>
4   <doc:author first="D" last="Shr"
5     mail="dhs@kent.ac.uk"/>
6  </doc:authors>
```

Listing 10.9: Dynamic transformation result

10.4.6 Transformation semantics: L_{xl} language

The processing of L_{xl} requires adaptation requirement dependent semantics. Both XHTML and WML have built-in linking support that is equivalent to the XLink simple links. Therefore, the desktop and mobile transformation pipelines must map the XLink links to their corresponding XHTML and WML representations. XSL-FO has also built-in linking support, but interactive links are not adequate for printed media. Consequently, the corresponding transformation outputs a non-interactive link representation, which consists of the link's name followed by the parenthesised link target.

All L_{xl} transformations must create the corresponding link representation and copy the subtree root, its children and all non-XLink attributes. The latter is necessary, because, during the transformation of attribute handled constructs, the attributes are included within their parent elements. Consequently, the subtree transformation is responsible for copying their parent element and its contents to their corresponding places within its output.

Listing 10.10 illustrates the L_{xl} XSL-T stylesheet for XHTML browsers, which also outlines the structure of all three transformation specifications. The utility templates in lines 11 and 13 are responsible for removing the XLink attributes and copying the remaining elements and attributes, respectively. The template at line 1 is responsible for processing the XLink attributes; it encloses their parent element to an anchor `xhtml:a` element. The WML stylesheet is identical, apart from the namespace declarations, because WML and XHTML share a common links syntax.

Listing 10.11 represents the non-interactive L_{xl} stylesheet. It copies the linked element and encloses the link URL in parentheses, as opposed to using a natively supported link representation. The resulting non-interactive link interpretation does not depend on the set of natively supported languages and it can be used for any non-interactive device.

```
<xsl:template
   match="*[@xlink:href]">
   <a href="{@xlink:href}">
      <xsl:copy>
         <xsl:apply-templates
               select="@*|*|text()"/>
      </xsl:copy>
   </a>
</xsl:template>

<xsl:template match="@xlink:*"/>

<xsl:template match="@*|*">
   <xsl:copy-of select="."/>
</xsl:template>
```

Listing 10.10: XSL templates for the interactive interpretation of L_{xl}

```
<xsl:template
    match="*[@xlink:href]">
   <xsl:copy>
      <xsl:apply-templates
          select="@*|*|text()"/>
   </xsl:copy>
   (<xsl:value-of
        select="@xlink:href"/>)
</xsl:template>

<xsl:template match="@xlink:*"/>

<xsl:template match="@*|*">
   <xsl:copy-of select="."/>
</xsl:template>
```

Listing 10.11: XSL templates for the non-interactive interpretation of L_{xl}

```
<adequacy>
   <expr ns="XMLPipeURI/Terms"
         name="supports">
      <contains>
         <termVal/>
         <val>XHTML URI</val>
      </contains>
   </expr>
   <expr ns="XMLPipeURI/Terms"
         name="supports">
      <not>
         <contains>
            <termVal/>
            <val>XLink URI</val>
         </contains>
      </not>
   </expr>
</adequacy>
```

Listing 10.12: L_{xl} XHTML handler adequacy expressions

```
<adequacy>
   <expr ns="XMLPipeURI/Terms"
         name="deviceType">
      <equals>
         <termVal/>
         <val>printer</val>
      </equals>
   </expr>
</adequacy>
```

Listing 10.13: L_{xl} non-interactive handler adequacy expressions

The L_{xl} language's transformation associations require adequacy expressions, because the optimal processing of its constructs depends on the adaptation requirements. Listings 10.12 and 10.13 illustrate the adequacy expressions associated with the XHTML and the non-interactive transformation pipelines, respectively. The former ensures that the target browser supports XHTML but not XLink. The non-interactive adequacy expression does not depend on any natively supported languages and only ensures that the target device is a printer, which is a non-interactive device.

Explicit applicability expressions are not necessary. The default applicability expression is sufficient, because all defined adequacy expressions correspond to required conditions and it requires that all adequacy expressions evaluate to non-zero values.

For example, consider the link at line 23 of the driving example:

```
<doc:em xl:type="simple" xl:href="...">links</doc:em>
```

Its presentation interpretation is an emphasised text portion that links to the specified URL. The desktop and mobile adaptation profiles do not support XLink links, but they support XHTML and WML links, respectively. Therefore, XMLPipe will use the aforementioned XHTML and WML transformation pipelines to interpret the simple XLink links to their natively supported representation:

```
<a href="...">
  <doc:em>links</doc:em>
</a>
```

In contrast, XMLPipe will use the non-interactive pipeline for the printer adaptation profile, which results in the proposed non-interactive links representation:

```
<doc:em>links (URL)</doc:em>
```

All transformations use the `doc:em` construct to represent emphasised content, because its subsequent processing will result to the optimal emphasis representation, for each adaptation profile.

10.4.7 Transformation semantics: L_{doc} language

The L_{doc} transformation semantics are also adaptation requirement dependent, and multiple transformation pipelines are necessary for mapping the L_{doc} constructs to their optimal natively supported representation. The optimal representation must use the set of natively supported languages, for each adaptation profile. Additionally, it must have an adequate layout, because L_{doc} constructs define the overall structure of a document.

The L_{doc} transformation semantics will include four transformation pipelines. A separate XSL-T stylesheet for each adaptation profile is sufficient for interpreting all L_{doc} constructs. Additionally, an image converting transformation is beneficial for the WML/WBMP mobile processing, because it is unlikely that the document source's `doc:img` constructs will refer to WBMP images. Therefore, the transformation semantics for L_{doc} will consist of three transformation pipelines that are associated with all L_{doc} handled constructs, and an additional image conversion pipeline that is only associated with the `doc:img` construct.

The L_{doc} desktop transformation can directly map all L_{doc} constructs to their natively supported counterparts. The corresponding XSL-T stylesheet uses four top level templates, which correspond to the four L_{doc} handled constructs. Listing 10.14 outlines the resulting interpretation of the example document. The top level `document` template produces the necessary top level XHTML elements, such as `head` and `body`.

```
<html xmlns="XHTML URI">
 <head>
  <title>Desktop example</title>
 </head>
 <body>
   <h1>Desktop example</h1>
   <div>
     <a href="...">M Ped</a>
   </div>
   ...
   <h1>1 The doc language</h1>
   <p>... <em>emphasized</em>
     text, images
     <img src="xmlPipe.gif"/>
     and nested sections.</p>
    <h2>1.1 Nested section</h2>
    ...
  </body>
</html>
```

Listing 10.14: L_{doc} interpretation for a desktop browser

```
<applicability>
 <contains>
  <termVal ns="XMLPipeURI/Terms"
           name="supported"/>
   <val type="String">XHTML URI</val>
 </contains>
</applicability>
<adequacy>
 <expr ns="XMLPipeURI/Terms"
       name="supported">
  <contains>
   <termVal/>
   <val type="String">XHTML URI</val>
  </contains>
 </expr>
 <expr  ns="XMLPipeURI/Terms"
        name="deviceType">
  <equals>
   <termVal/>
   <val>desktop</val>
  </equals>
 </expr>
</adequacy>
```

Listing 10.15: L_{doc} XHTML desktop binding adaptation specification

The presentation of the document authors consists of their names, which are linked to their e-mail addresses. The `section` element's interpretation is a heading text that is numbered according to its original order and nesting. Such a numbering is feasible, because sections can only occur within the context of a `document`, since they are not handled constructs. Finally, the `doc:em`, `doc:p` and `doc:img` are interpreted as their identical XHTML constructs.

The binding of the L_{doc} transformation semantics requires an explicit applicability expression, which is illustrated in Listing 10.15. The described desktop browser transformation applies to all XHTML browsers, but it is more adequate for desktop computers that do not impose significant display size limitations and can display a variety of binary image representations. Therefore, the applicability expression only requires XHTML support, but adequacy expressions also express the transformation's adequacy for desktop computers.

The XSL-FO printer transformation specification interprets the L_{doc} constructs in a similar manner to the desktop browser transformation, but it uses XSL-FO constructs and an more adequate layout for printed media. It generates a paged document layout and optimises its content for fixed page dimensions. The `doc:document` construct is mapped to the top level XSL-FO constructs that define the static page layout, the document title and the author information. The document paragraph constructs are mapped to XSL-FO `block` elements, which represent individual flows of text. The formating constructs are mapped to their corresponding combinations of font sizes and faces. The `doc:img` construct interpretation uses the corresponding XSL-FO graphics import construct: `<external-graphic>`

The mobile transformation semantics must address the limitations of both the WML and the described device. Current high end mobile phones provide rich processing and interaction capabilities. However, as described in Section 10.4.2, the case study mobile profile describes a very limited device, in order to allow the evaluation of XMLPipe's adaptation capabilities. WML only supports primitive formating and scripting, and most of the earlier mobiles do not support text font variations. Moreover, they have low resolution monochrome screens and do not support the majority of desktop browsing functionality, such as scroll bars and navigation buttons. The corresponding stylesheet must map the L_{doc} constructs into an adequate interpretation for such limited capabilities.

The L_{doc} interpretation for the case study mobile profile consists of a hierarchical organisation of minimal text portions. A WML document consists of page *templates* and *cards*. The former define each page's browsing behaviour by attaching actions to the interactive components of the mobile, such as its buttons. The document content is divided among several interlinked *cards*. The WML mobile transformation stylesheet creates a separate card, for each document paragraph, and a contents table card, for each document section. The generated templates assist the document navigation by customising the mobile controls to navigate at either the next text portion or to the table of contents. WML contains an `em` construct, but it does not modify the text appearance in devices with minimal font support. Therefore, the proposed stylesheet encloses the emphasised text into asterisks, in order to ensure the explicit presentation of the `doc:em` construct.

```
<wml>
 <template>
 <do label="start" type="reset">
   <go href="#N400001"/>
 </do>
 <do label="back" type="prev">
   <prev/>
 </do>
 </template>
  ...
 <card id="N400024">
 <do label="next" type="accept">
   <go href="#N400030"/>
 </do>
   <p>A foreign namespace...
     <a href="...">*links*</a>
    ...</p>
 </card>
 <card id="N400030">
  <p>This is a mobile</p>
 </card>
</wml>
```

Listing 10.16: L_{doc} interpretation for a WML mobile

```
<handler name="WBMP_coverter">
 <hcList ...><hcRef name="img"/>
 </hcList>
 <adequacy>
  <expr ns="DOC_URI"
        name="doNotRecurse">
    </not><termVal/></not>
  </expr>
  <expr ns="..." name="supports">
    <contains>
     <termVal/><val>WBMP URI</val>
    </contains>
  </expr>
  <expr ns="..." name="supports">
    <not><contains>
      <termVal/><val>JPEG URI</val>
    </contains></not>
  </expr>
 </adequacy>
 <context>
  <statm ns="DOC_URI"
    name="doNotRecurse">true</statm>
 </context>
 ...
</handler>
```

Listing 10.17: L_{doc} WML mobile partial binding adaptation specification

Listing 10.16 outlines the example document interpretation for the WML mobile. The WML template at lines 2–9 programs the two primary mobile navigation buttons to navigate to the previous page and the top level index. Within cards that do not represent the last paragraph of a section, such as the card at line 11, an additional association programs the middle mobile button to navigate to the next paragraph. The "links" text, at line 16, is an emphasised link within the example document; The illustrated WML output encloses the corresponding text within asterisks and the WML `<a>` anchor element.

WML 1.1 does not support namespace declarations and requires a DOCTYPE declaration. The proposed transformation pipeline addresses the lack of namespaces support by including an XSL-T transformation that removes all namespace declarations. XMLPipe allows the document root's transformation (`doc:document` element) to specify the resulting document's `DOCTYPE` declaration. Therefore, it is sufficient that the transformation template that corresponds to the `doc:document` element produces a WML `DOCTYPE` declaration.

The transformation of the `doc:img` construct to its corresponding WML construct is not sufficient for a low-end mobile. The desktop and printer stylesheets mapped the `doc:img` construct to its corresponding natively supported construct, under the assumption that there is native support of the referenced image representation. Such an assumption does not apply to low-end WML mobile devices that are unlikely to support other image representations than WBMP. The case study processing semantics address the mobile image representation limitations by associating an image converter to the `doc:img` handled construct.

Listing 10.17 is a partial representation of the image converter's declaration. The binding adaptation specification consists of a set of adequacy expressions. The default applicability expression, which requires that all adequacy expressions evaluate to non-zero values, provides an approximate applicability measure. Specifically, it requires WBMP support and lack of JPEG support. A browser that supports WBMP and not JPEG images is likely to only support the simplistic WBMP format and no other more complex image representations. The WBMP converter is not specific to mobiles or WML, because it can be used for any adaptation profile that supports WBMP images. Consequently, the proposed applicability/adequacy expression does not include any device type requirement. The image converter atomic transformation produces a temporary WBMP representation of the original image, according to the image size constraints of the target device. It also maps the source `doc:img` construct to an output `doc:img` construct that refers to the temporary WBMP file.

The `doNotRecurse` adequacy expression, in line 5, ensures that the conversion transformation will not recurse indefinitely. Specifically, `doNotRecurse` is a language specific term that belongs to the same URI as L_{doc}. During the initial `doc:img` subtree processing the image conversion transformer is applicable, because all illustrated expressions are true, since the default value of `doNotRecurse` is *false*. The `context` statement, in line 21, instructs the pipeline driver to introduce a language specific statement that changes `doNotRecurse` value to *true*. Since `doNotRecurse` is a language specific term, XMLPipe retains its value during the processing of the `doc:img` rooted subtree. Therefore, the image converter will not be applicable when XMLPipe traverses the newly generated `doc:img` construct.

The handled construct based association of transformation semantics is essential for modular transformation semantics specifications and independent introduction of

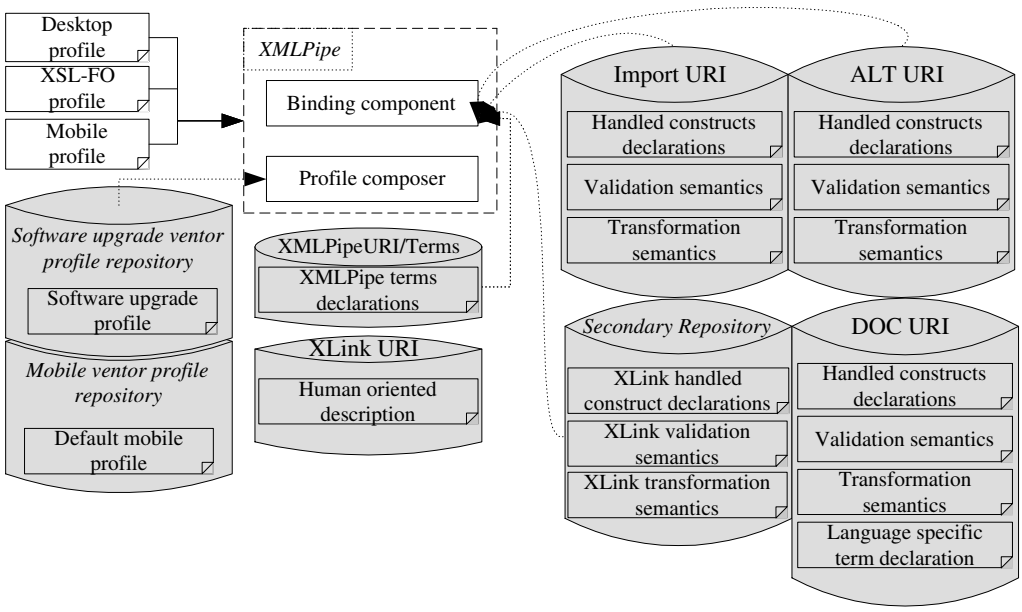

Figure 10.6: Proposed processing semantics distribution

partial transformers, such as the image converter. Its design and adequacy expressions are significantly different from the aforementioned generic WML transformation pipeline. The handled construct based associations enable its association with the `doc:img` construct, independently of the generic WML pipeline.

10.4.8 Semantics binding

An adequate distribution of the described processing semantics requires a combination of both primary and the secondary location mechanisms. The pilot XMLPipe implementation has limited binding support and uses a single processing semantics repository, as described in Section 10.3. Consequently, the document processing experiment uses a single local file that contains all described processing semantics. Nonetheless, this section will describe a theoretical adequate semantics distribution, in order to illustrate the benefits of the proposed binding model.

Figure 10.6 illustrates the proposed distribution of the case study processing semantics.

The location of the composite adaptation profiles does not relate to the binding model, because the preprocessing initiation entity must provide all adaptation requirements information. The *profile composer* is responsible for resolving any composite profile external references and retrieving the corresponding adaptation requirements. For instance, the proposed mobile profile contains two external references to the default device profile and to the software upgrade profile. The profile composer retrieves both and combines them with the mobile profile statements.

The L_{doc}, L_{imp} and L_{alt} languages have been introduced by this case study, and L_{xl} corresponds to the W3C XLink language. Therefore, the principal location mechanism can be used for the case study languages by adding RDDL links to the Web pages that

correspond to their URIs. The XMLPipe binding model can use the namespace URIs to locate these RDDL links and retrieve all necessary handled construct, validation and transformation semantics. W3C owns the XLink namespace URI and there are no XMLPipe specific RDDL links in the corresponding Web page. Consequently, a secondary location mechanism is necessary for locating the L_{xl} processing semantics. In addition to the core processing semantics, the interpretation of the adaptation profiles and the optimal pipeline selection process require the semantics of several adaptation terms. The case study uses a set of well defined XMLPipe adaptation terms and the language specific `doNotRecurse` term. All the adaptation term processing semantics, in a similar manner to the case study specific languages, can be located using the RDDL links in the corresponding Web pages. The RDDL link that corresponds to the `doNotRecurse` term must coexist with the L_{doc} processing semantics, because they both share the same namespace URI.

The proposed semantics distribution is one of many alternative distributions. For instance, any of the case study language semantics can also reside in secondary semantics repositories. In fact, the local processing semantics repository that is used by the XMLPipe pilot implementation is such a secondary repository. Moreover, the proposed binding model is guaranteed to use the remote semantics repositories only during its first invocation. Subsequent processing can use the local semantics cache to acquire the required processing semantics, without requiring any external communication.

10.4.9 Document processing

The described processing semantics enable the authoring validation and adaptive transformation of the example document, for all three adaptation profiles. A detailed description of all processing steps would be prohibitively lengthy, because of the recursive nature of the proposed processing. Therefore, this section will describe each processing step once. Subsequently, Section 10.4.10 will illustrate the collective output of all processing steps.

Both the document user and the document author provide the example document's URI *uri* to initiate its processing. For document transformation, the document user also provides a composite adaptation profile *cpr*, which can correspond to any of the three case study profiles: the XHTML desktop, the WML mobile or the XSL-FO printer profile. For instance, the pilot implementation invocation for transforming `document.xml` according to the mobile adaptation profile is

```
java XMLPipeCmd aMobile document.xml
```

The pilot implementation does not support validation, but the corresponding invocation could be

```
java XMLPipeCmd -V document.xml
```

The authoring validation invocation omits the adaptation profile name, because authoring validation is adaptation requirements independent.

For both validation and transformation, the first processing step is document parsing. The validation and transformation drivers use the XML parser to map *uri* to its corresponding DOM tree $d = parse(uri)$.

The next step, in the case of transformation, is to retrieve and process the composite profile. XMLPipe retrieves the specified composite profile *cpr* from a local profile repository. Subsequently, the transformation driver uses the profile composer to convert

it to an adaptation profile $pr = ccompose(cpr)$. For example, consider the invocation of *ccompose* for the mobile composite profile, which consists of the sequence of adaptation statements illustrated in Figures 6.3, 6.4 and 6.5 (page 98). Initially, *ccompose* adds the statements of Listing 6.3 to the resulting adaptation profile pr. Subsequently, it resolves the first introduced conflict, because of the statement in Listing 6.4, using the resolution expression of the `supported` term: a union of the two conflicting values. The resulting profile contains both WML and WBMP URIs. Finally, Listing 6.5 introduces two statements that conflict with the initial image size specifications. *ccompose* applies the XMLPipe default conflict resolution, because there are no term or statement specific resolution expressions. Therefore, the resulting profile restricts the maximum image width and height to 96 and 100 pixels, respectively.

The profile composer retrieves the required processing semantics by interoperating with the binding component. For instance, consider the `supported` adaptation term. When the profile composer encounters the `supported` adaptation statement, it retrieves the term's semantics by invoking the *locateTermSem* binding component function.

$$ts_1 = locateTermSem((\texttt{XMLPipeURI/Terms}, \texttt{supported})$$

The *locateTermSem* function delegates the above request to the generic *locateSemantics* function. *locateSemantics* returns the corresponding semantics information, if it is available in the processing semantics cache. Otherwise, it uses *cacheImport* to ensure that the semantics cache is up-to-date. The Web page that corresponds to the `XMLPipeURI/Terms` URL, contains an RDDL link to the `supported` term semantics. *cacheImport* uses this RDDL link to locate the corresponding term semantics. This case study does not include any conflicting semantics. However, if there where conflicting semantics, *cacheImport* would only cache the most trusted semantics, according to the set of predefined repository trust levels. The semantics retrieval process is identical for all other XMLPipe processing semantics: the validation semantics, transformation semantics and handled construct information.

After the document parsing and profile composition, the transformation driver passes the transformation request to the integration model transformation driver, which implements the *transformRev* function. *transformRev* performs a postorder document traversal and transforms the individual handled construct rooted subtrees. Figure 10.7 illustrates the simplified[1] postorder document traversal and subtree transformation substitution, for the example document. The postorder traversal begins at the contents of the first `alt:case` element and proceeds until it encounters the `alt:alt` element, which is a non-natively supported handled construct. Subsequently, it calls *bestHCTS* to choose the corresponding optimal pipeline transformation specification. `alt:alt` is associated with a single transformation pipeline (T') that has no associated adequacy or applicability expressions. Therefore, *bestHCTS* returns its corresponding transformation pipeline T. *transformRev* separates the subtree d_1 that is rooted at `alt:alt`, as Figure 10.7 illustrates, because `alt:alt` is an element *SMC* construct. The application of T' to d_1 results in d'_1: $d_1 \xrightarrow{T'} d'_1$. d'_1 is not null and $d_1 \neq d'_1$, because d'_1 contains one of the alternative subtrees, originally enclosed within the `alt:case` elements. Therefore, *transformRev* substitutes d_1 with d'_1 and resumes the document traversal from the innermost node of the newly added d'_1. In a similar manner, *transformRev* processes all the constructs of d, until the document solely consists of natively supported constructs.

[1] It is a simplified illustration, because it does not contain all the tree substitutions and their corresponding recursive traversals.

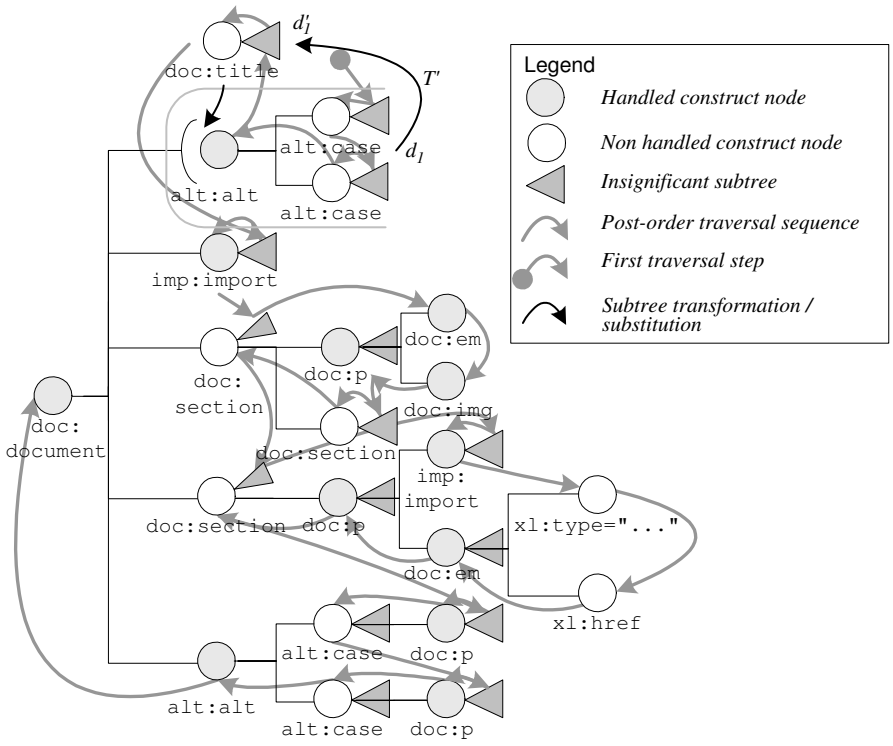

Figure 10.7: Transformation processing: document traversal and subtree transformation

The processing of handled constructs that are associated with multiple pipeline specifications requires interoperation with the adaptation measure evaluator, as opposed to the described `alt:alt` transformation. For instance, during the processing of `xl:type` and `xl:href`, *bestHCTS* must interoperate with the adaptation measure evaluator, in order to choose the optimal transformation pipeline. Consider that the binding specifications B_d, B_m and B_p correspond to the XHTML, WML and printer transformation pipelines, respectively. Additionally, consider the example mobile adaptation profile pr_m. The desktop pipeline is not adequate for the mobile profile, because the mobile adaptation profile does not state XHTML support: $measure(pr_m, B_d) = 0$. In a similar manner $measure(pr_m, B_m) = 2$ and $measure(pr_m, B_p) = 0$. Consequently, *bestHCTS* returns the WML pipeline for the mobile profile. Subsequently, *transformRev* uses it to transform the simple XLink links to their equivalent WML constructs.

The transformation of the `doc:img` construct deserves special mention, because it uses language specific adaptation statements and there are conflicting transformation pipelines. When *transformRev* reaches the `doc:img` construct, *bestHCTS* must choose the optimal transformation pipeline between four alternatives: the `doc:img` specific pipeline and the three generic L_{doc} pipelines.

When interpreting d for the mobile profile, only the generic WML and the `doc:img` specific image converter correspond to non-zero *measure* results, because the other pipelines require unsupported languages (XHTML and XSL-FO). Consider that the

binding information B_i and B_m correspond to the WBMP image converter and to the generic WML pipeline, respectively. During the first subtree invocation, the former results in a higher measure: $measure(pr_m, B_m) = 2$ and $measure(pr_m, B_i) = 3$. Therefore, *transformRev* will prioritise the image conversion pipeline. The image conversion pipeline substitutes the initial `doc:img` element with another `doc:img` element that refers to a temporary WBMP version of the initial image. Additionally, the `context` element, used in the pipeline definition, introduces the value *true* for the language specific term `doNotRecurse`.

Subsequently, *transformRev* revisits the newly introduced `doc:img` construct and calls *bestHCTS*. The introduced *true* `doNotRecurse` value applies to the `doc:img` rooted subtree and influences the adaptation measure evaluation. Specifically, the image converter applicability expression requires that `doNotRecurse` is false; therefore the image converted is no longer applicable: $measure(pr_m, B_i) = 0$. Consequently, *transformRev* uses the generic WML transformation pipeline, which maps `doc:img` to its corresponding WML image inclusion construct. Subsequently, the traversal continues to the remaining document constructs. The processing of other `doc:img` constructs is not affected, because the language specific term value modifications are local to each subtree.

The pipeline driver is responsible for applying all described subtree transformations by instantiating and invoking the individual atomic transformations, which are described by the pipeline specifications. For example, consider the `imp:import` transformation pipeline, which is a dynamic transformation. It contains two atomic specifications A_1 and A_2, which correspond to the non-dynamic and the dynamic XSL-T atomic transformations, respectively. The pipeline driver instantiates the corresponding pipeline by locating and instantiating T_1 and T_2, which are the transformations that correspond to the A_1 and A_2 specifications. Subsequently, it applies T' to d_1: $d_1 \xrightarrow{seq(\texttt{validateHC}, dyn(T^{\epsilon}, T_1, T_2))} d'_1$.

If the preprocessing initiation entity requests authoring validation, the authoring validation component, which implements the *validateAuth* function, performs a two step authoring validation process. Firstly, it calls the *transformAuth* function, which is implemented by the transformation driver, in order to remove as many *SMC* rooted subtrees as possible. In a similar manner to the described document transformation, *transformAuth* traverses the document in a postorder manner. However, it only transforms the *SMC* rooted subtrees, which are the `alt:alt` and `imp:import` rooted subtrees. Prior to their transformation, it also calls `validateHC`, in order to ensure the validity of the topmost processed constructs. All the *SMC* rooted subtrees of the example document are valid. Consequently, *transformAuth* succeeds and *validateAuth* proceeds to calling *validate*, in order to validate the *SMC* free document. The latter is valid, according to the XMLPipe integration model. Therefore, the authoring validation process succeeds.

Both `validateHC` and `validateSubtree` delegate all validation requests to *validate*. Figure 10.8 illustrates how *validate* traverses the document tree and separates the individual subtrees. As opposed to document transformation, document validation begins with the document root construct. For each subtree, *validate* firstly separates all foreign namespace subtrees. Subsequently, it locates and applies the optimal atomic validation. For instance, as Figure 10.8 illustrates, *validate* firstly separates all foreign namespace descendants of `doc:document` and validates the resulting single namespace subtree. After the validation of a subtree, *validate* recurses into its foreign namespace

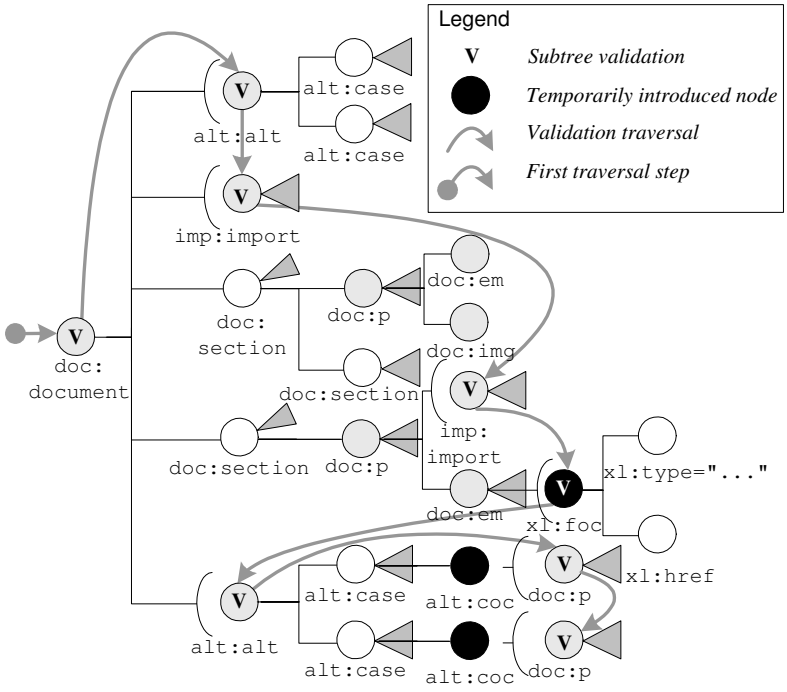

Figure 10.8: Validation processing: document traversal and subtree separation

subtrees.

During the separation of the document subtrees, *validate* introduces the necessary predefined `foc` and `coc` constructs. For example, as Figure 10.8 illustrates, *validate* adds both XLink attributes to a temporary `foc` element. The `foc` element provides a well defined context to the XLink attributes, and it enables the specification of schemas that restrict their syntax. After their validation, *validate* removes the `foc` element and adds them back to their original parent (the `doc:em` element). Additionally, during the separation of the `doc:p` constructs from the second `alt:alt` rooted subtree, *validate* temporarily replaces them with the predefined `coc` construct. The `coc` element denotes foreign namespace subtrees that are rooted at a *COC*. The validation specifications must allow the occurrence of `coc` all places where foreign namespace content is expected. Consequently, its introduction ensures that subtree validation fails, if there are invalidly nested *COC* rooted subtrees.

10.4.10 The transformed document

The previous section described all aspects of XMLPipe document processing. The processing of the example document results in a different output for each of the three example adaptation profiles. This section will describe the individual transformation outputs and illustrate their rendering by the corresponding browsers and devices.

For each adaptation profile, the produced output is adequate for the corresponding device and browser. For instance, Figure 10.9 illustrates the document interpretation for the desktop browser profile. The interpretation of L_{doc} constructs as a continuous

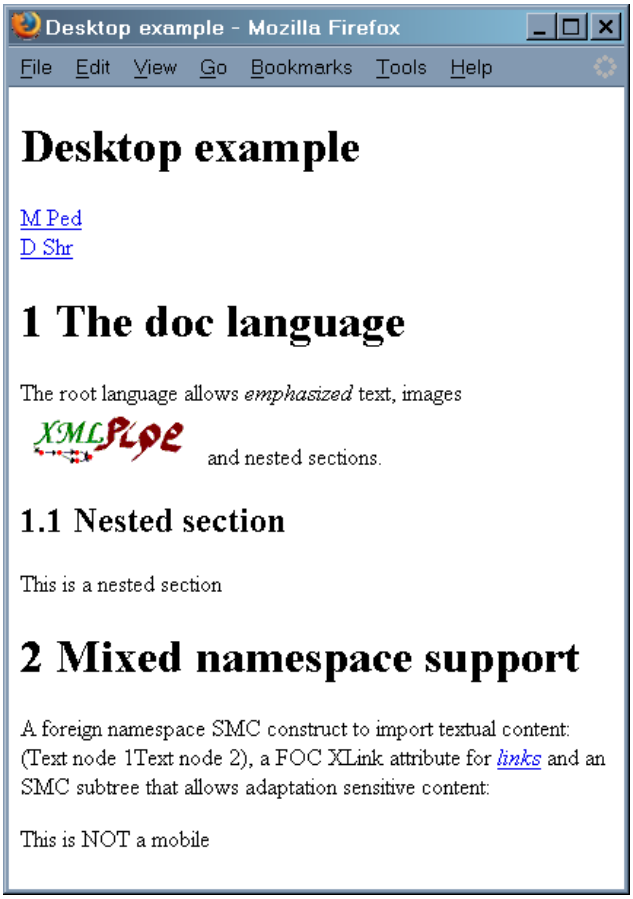

Figure 10.9: Document transformation result: Desktop profile

content flow is adequate for online document browsing.

In addition to the overall document layout, the interpretation of the individual constructs is also adequate for the desktop profile. Specifically, the sections are numbered according to their nesting, and the abstract author information has been transformed to a list of author names, which are linked to their e-mail addresses. The `doc:em` and `doc:img` constructs have been transformed to their corresponding XHTML constructs. The `alt:alt` *SMC* handled construct has also been interpreted correctly, as illustrated by the document title that is "Desktop example" instead of "Mobile example". The linked text "links", towards the end of the rendered document, illustrates that XMLPipe mapped the XLink attributes to the corresponding XHTML anchor element. Finally, the processing of the `imp:import` *SMC* rooted has imported the referenced author information.

In a similar manner to the desktop profile, the XSL-FO printer profile also allows rich content presentation. The corresponding interpretation only differs in the overall layout and the presentation of links. Figure 10.10 illustrates an XSL-FO previewer's rendering of XMLPipe's output, for the XSL-FO printer profile. As opposed to the desktop continuous content flow, the printer interpretation consists of a sequence of fixed size pages. The top level formating consists of centred headings and justified

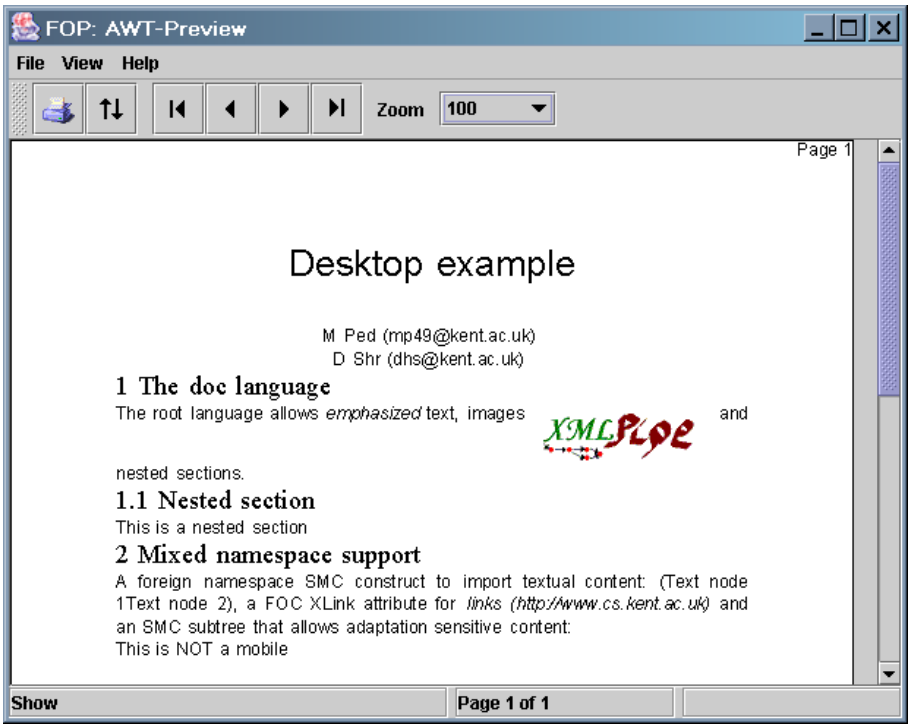

Figure 10.10: Document transformation result: XSL-FO printer profile

text, which are more adequate for printed media. The interactive interpretation of links is not adequate for printed media. Consequently, the source links have been interpreted as pairs of their name and URL target.

The document interpretation for the mobile profile, illustrated in Figure 10.11, differs substantially from the desktop and printer interpretations, because of the example mobile's limited presentation and interaction capabilities.

There is a table of contents for the whole document and for each document section, in order to assist the browsing of the document using a small low resolution display. The first screenshot illustrates the root contents table, which links to the two top level sections. The second screenshot (on the right) illustrates the contents table of the first section, which contains a paragraph ("text section") and a nested section. The document user does not have to scroll over large portions of text and can use the content table links to navigate through the document sections and paragraphs.

All document constructs have been mapped to their optimal interpretations for the mobile profile. The first screenshot illustrates the correctness of the `alt:alt` (lines 7–14) processing, because the document title is "Mobile example". The third screenshot shows the imported image and the mobile rendering of emphasised text, which is enclosed in asterisks. The processing of the `doc:img` construct uses both the the image converter, which maps the source JPEG representation to a WBMP file, and the generic WML transformation, which maps the `doc:img` to its corresponding WML construct. The fourth and fifth images illustrate the imported content and the WML interpretation of the XLink simple links. Finally, the last screenshot illustrates the

Figure 10.11: Document transformation result: Mobile profile

correct selection of the optimal `alt:alt` alternative paragraph: "This is a mobile".

10.4.11 Reusing the semantics of existing languages

The recursive nature of the proposed processing allows the reuse of existing languages to simplify the processing semantics of a new language. For instance, the transformation semantics of L_{doc} and L_{xl} consist of multiple transformation pipelines that fulfill several sets of adaptation requirements. If the transformation semantics of a new language L map its constructs to the constructs of L_{doc}, no multiple transformation pipelines will be necessary, because the processing of L_{doc} constructs is well defined for all case study profiles.

An illustration of the above language reuse concept requires a higher level presentation language than L_{doc} and L_{xl}. However, the case study languages have been purposely kept simple, and they cannot provide the foundation for usable higher level presentation languages. Nevertheless, they can be used for the presentation interpretation of a non-presentation language's constructs.

Listing 10.19 uses such a language to describe a compact disk (CD) collection. Each `cd:cd` entry of the non-presentation language L_{cd} defines the information of a CD: its URI identifier, cover image, title and list of authors and songs. Listing 10.18 uses the constructs of L_{doc}, to define the top level structure of a document, and the constructs of L_{imp}, to import the compact disk information from Listing 10.19.

The compact disk language is not a presentation language, because the primary interpretation of its constructs relates to the disk information and their presentation is context dependent. For instance, within the context of the above example, each `cd:cd`

```
<doc:document
 <doc:title>CD collection</doc:title>
  ...
 <doc:section>
  <doc:title>The CDs</doc:title>
  <doc:p>One subsection per CD</doc:p>
  <doc:section>
   <doc:title>
    <imp:import href="imp2.xml"
     select="//*/*[@uri='uri:cdInfo1 ']
            /*[1]/text()"/>
    </doc:title>
    <imp:import href="imp2.xml"
    select="//*/*[@uri='uri:cdInfo1 ']"/>
  </doc:section>
  <doc:section>
   <doc:title>
    <imp:import href="imp2.xml"
     select="//*/*[@uri='uri:cdInfo3 ']
            /*[1]/text()"/>
    </doc:title>
    <imp:import href="imp2.xml"
    select="//*/*[@uri='uri:cdInfo3 ']"/>
  </doc:section>
 </doc:section>
</doc:document>
```

Listing 10.18: Semantics reuse example: driving document

```
<media xmlns="CD URI"
       xmlns:xl="XLink URI">
 <cd coverImg="cdcover1.jpg"
     uri="uri:cdInfo1">
  <title>...</title>
  <artists>
   <artist name="..."/>
   <artist name="..."/>
  </artists>
  <songs>
   <song>...</song>
   <song>...</song>
  </songs>
 </cd>
 ...
 <cd coverImg="cdcover3.jpg"
     uri="uri:cdInfo3">
  <title>...</title>
  <artists>
   <artist name="..."/>
  </artists>
  <songs>
   <song>...</song>
   <song>...</song>
   <song>...</song>
   <song>...</song>
  </songs>
 </cd>
</media>
```

Listing 10.19: Semantics reuse example: imported document

rooted subtree can be presented as a sequence of paragraphs that contain all compact disk information. If the same information was presented as a result of a media database query, a more compact presentation would be more adequate.

For the purpose of this case study, we will associate the constructs of L_{cd} with their adequate transformation semantics for their illustrated use, in Listings 10.18 and 10.19. Specifically, each CD information entry can be considered as an independent information entity that has a well defined presentation. Consequently, the `cd` element construct is a *COC* construct, because it introduces a well defined piece of presentable information and its processing can be defined independently of its context.

A single atomic transformation that reuses the L_{doc} and L_{xl} constructs is sufficient for processing all L_{cd} constructs. Specifically, an XSL-T stylesheet can map each `cd` element to a series of `doc:p` paragraph elements. The first can contain the compact disk cover image and the compact disk title, which can also be a link the associated unique URI. The subsequent paragraphs can list the compact disk artists and songs. The associated stylesheet consists of two simple transformation templates. The corresponding transformation semantics require a minimal declaration and neither applicability nor adequacy expressions, because the atomic transformation is adaptation requirements independent.

The above minimal specification of transformation semantics is sufficient for adapting the illustrated documents, according to all case study profiles. Figure 10.12 illustrates the rendering of the XMLPipe pilot implementation output for all three adaptation profiles. The XHTML desktop interpretation is a continuous flow of the document information, which uses interactive links and full sized images. The interpretation for the XSL-FO printer profile also provides a rich presentation, but it uses a paged layout and a non interactive presentation of links. The mobile interpretation consists of a set or hierarchically organised cards. The corresponding CD cover images have been converted to low resolution WBMP images.

The illustrated rendering is not of a high standard, because the case study languages are simple and do not contain fine grained presentation functionality. However, it establishes that the proposed processing allows reusing existing languages to simplify the processing semantics of new languages.

10.5 Case study discussion

The purpose of the described case study is twofold. Firstly, it must bridge the described theory with its practical applications, by illustrating how to combine all introduced concepts to preprocess a presentation document. Secondly, it must illustrate the feasibility of generic document processing for the Web, which is essential for supporting our hypothesis. The subsequent XMLPipe discussion section will describe the adequacy of XMLPipe, but a case study provides clearer insight in the benefits and feasibility of generic document processing.

The case study has adequately illustrated all XMLPipe concepts. Specifically, it described the necessary semantics specifications for processing a mixed namespace presentation document, according to three separate profiles. The processing description covered both the authoring validation and transformation. Furthermore, it described how each processing component operates for both processes. The provided rendering of the output documents indicated their adequacy for the corresponding adaptation profiles. Therefore, the case study fulfilled its initial purpose, since it illustrated how XMLPipe enables the adaptation of presentation documents that combine independently developed languages, without requiring processing information by either the document user or the document author.

The following discussion focuses on investigating whether the case study has sufficiently illustrated the feasibility of generic document processing. Specifically, it will describe the adequacy of the chosen case study example, the illustrated semantics definitions and the illustrated processing. Section 10.5.1 focuses on the chosen processing scenario and the sufficiency of the case study document and adaptation profiles. Section 10.5.2 addresses the proposed specifications, their simplicity and their adequacy for open sets of languages. Finally, Section 10.5.3 describes how the case study illustrated the feasibility of document validation and transformation, for a variety of adaptation requirements.

10.5.1 Processing scenario discussion

An adequate processing scenario must cover the variety of Web information sources and consumers, within the scope of this thesis. A preprocessing scenario consists of a set of languages, a document that combines them and a set of target adaptation profiles. The unrestricted nature of the Web impedes the development of exhaustive

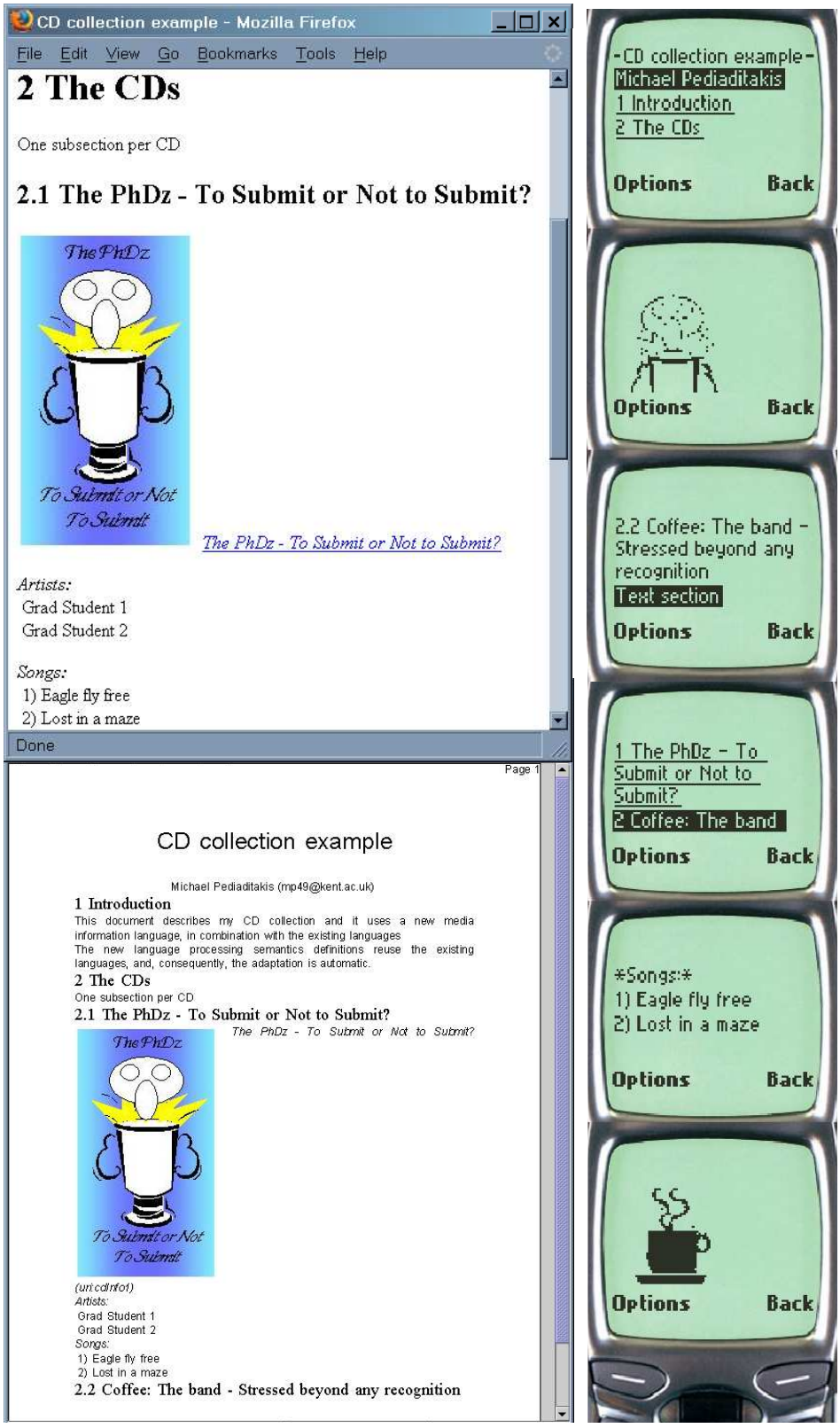

Figure 10.12: Semantics reuse example: document rendering for all case study profiles

case studies, because there are unlimited combinations of presentation documents and adaptation profiles. Nevertheless, an adequate processing scenario can use examples that span the necessary information and functionality spectrum, without introducing a prohibitive multitude of processing cases. Such examples can indicate the feasibility of generic processing, by illustrating the breadth of a processing model.

The minimal design of the driving example languages and document does not allow a sufficient coverage of the presentation document domain, but they illustrate a combination of common presentation abstractions that are defined by independently developed languages. Specifically, the four example languages are presentation languages and the resulting document is a presentation document. The example languages and document focus on reducing the complexity of the case study, and they do not expose a rich set of presentation functionality. However, they introduce document and text formating abstractions, interactive links, dynamic content and images, which are essential to many document presentation applications. Additionally, they follow the modular design principle, and each language covers a minimal application domain and is defined independently of the others, since there are no predefined inter-language relationships.

In a similar manner to the input languages, the example adaptation profiles cannot cover the whole spectrum of Web adaptation requirements. However, they allow sufficient illustration of the XMLPipe adaptation capabilities, because the three adaptation profiles describe significantly different devices that belong to opposing sides of the Web device capabilities spectrum. Specifically, the desktop profile corresponds to high-end devices and browsers, and it allows a rich presentation of all the driving example constructs. In contrast, the mobile adaptation profile corresponds to a minimal set of processing and presentation capabilities. The supported WML and WBMP representations do not allow a straightforward presentation of most driving example constructs, and the limited display and navigation capabilities require a highly customized document layout. The combination of the desktop and mobile profiles test the adaptation range that a processing model can provide. The XSL-FO printer adaptation profile lies in the middle of the capabilities spectrum, but it does not allow any interaction. The lack of interaction capabilities poses a significant adaptation challenge, because it requires different output layout and presentation of naturally interactive constructs (such as links).

10.5.2 Processing semantics discussion

A core case study aim was to illustrate that XMLPipe enables the straightforward specification of processing semantics that are adequate for an open set of independently developed languages. A semantics specification process is adequate for an open set of independently developed languages, if it neither relies on predefined language relationships nor on predefined integration profiles. Moreover, a semantics specification process is straightforward, if it does not require complex processing definitions and complex associations. This section will show that the described case study illustrated both the adequacy and straightforward nature of the XMLPipe semantics specifications.

The XMLPipe semantics for a language consist of its handled construct, validation and transformation specifications. The case study illustrated that they are adequate for an open set of independently developed languages, because no specification relied on inter-language relationships or integration profiles.

The handled construct specifications depended on the semantics and the syntax of the corresponding languages, and not on other languages. In a similar manner,

each language's validation semantics did not refer to other languages, and they only depended on its handled constructs and syntax. Furthermore, the predefined `foc` and `coc` constructs allowed the identification of the necessary integration points, such as the places where content is expected, without requiring references to foreign namespace constructs.

The XMLPipe transformation model allows the individual transformers to access foreign namespace constructs, but no illustrated transformation specification required such access. The transformation semantics consist of the transformation pipelines and their corresponding atomic transformations. Both may access foreign namespace constructs, because each transformed subtree is not separated from its descendant foreign namespace subtrees. However, no illustrated specification required references to other languages, and they only had to obey the XMLPipe transformation design guidelines (such as copying foreign namespace content).

The case study also illustrated that the lack of inter-language dependencies and the well defined design guidelines enable the straightforward definition of all necessary semantics. A semantics author can define the semantics of a language L, without referring to all other languages that can be potentially combined with L. Additionally, the well defined validation and transformation design guidelines provide all necessary information for the creation of adequate semantics specifications. The semantics authors must only be aware of the design guidelines, and they are not required to have a thorough understanding of the XMLPipe processing. Furthermore, the proposed design guidelines do not complicate the specification of schemas and transformations, since the illustrated specifications would not be significantly simpler, if they were designed for single namespace processing.

The transformation semantics specification is further simplified, because of the proposed handled construct based binding, transformation pipelines and recursive processing. The handled construct binding and the transformation pipelines assist the definition of modular transformations, which combine simple specifications to achieve complex transformation tasks. For instance, the illustrated image converter required a significantly different design and binding from the generic L_{doc} handler. The handled construct based associations enabled its definition as a separate transformer, without requiring its integration in the L_{doc} transformer. Additionally, the dynamic transformation pipelines enabled the processing of L_{imp} by a simple seven line long specification. Without the proposed transformation pipeline a significantly more complex XPath evaluator would be necessary. Furthermore, the semantics author can utilise the recursive XMLPipe processing to reuse existing languages, in order to simplify the semantics specifications. For instance, the semantics of L_{cd} would require multiple transformation specifications, in order to cover a variety of adaptation requirements. Nevertheless, the case study illustrated that a single minimal XSL-T stylesheet is sufficient, if it reuses the other case study languages.

Summarising, the case study illustrated that the XMLPipe processing semantics are adequate for independently developed languages and their definition is not overly complex.

10.5.3 Processing discussion

In addition to illustrating the adequacy of the XMLPipe semantics, the case study also aimed to illustrate the feasibility of the proposed processing for the Web. Such an aim can be accomplished by showing that a set of independently developed semantics can

be combined to validate and transform mixed namespace presentation documents. The transformation must be adequate for an unrestricted variety of adaptation requirement sets. Furthermore, it must not require explicit processing/adaptation information from either the document user or the document author, because such information restricts the document processing and impedes the processing of an open set of languages.

The case study illustrated that neither document processing nor document adaptation information is required by either the document user or the document author. The input case study document consists of only the information that the document author wishes to convey. The document user, who is the preprocessing initiation entity, only specifies the desirable type of processing (such as validation or transformation) and a set of adaptation requirements. The subsequent XMLPipe processing steps use the XMLPipe binding model to retrieve all the necessary processing semantics, for transforming and validating the document. The current version of the XMLPipe pilot implementation only supports a local semantics repository, and the document user might have to specify its location. However, a complete implementation would combine the proposed principal and secondary location mechanisms to retrieve all required information, without the intervention of the document user.

The case study processing descriptions illustrated the feasibility of combining independently developed processing semantics to correctly preprocess a presentation document. The illustrated authoring validation process combined the language validation semantics and the necessary transformation semantics (for processing the *SMC* rooted subtrees), and it successfully validated the example document. In a similar manner, the illustrated transformation processing used the independently developed transformation semantics, in order to recursively transform each document subtree into its optimal representation. The resulting documents provided an adequate interpretation of the original information, according to the three adaptation profiles.

The case study only used three predefined profiles. However, it illustrated that XMLPipe preprocessing is adequate for the versatility of the Web environment and does not depend on fixed adaptation profiles. If the transformation selection directly depended on the individual profiles, a separate binding would be required for each adaptation profile. Furthermore, the transformation semantics would have to be updated upon the modification of a profile, because it would express a different set of adaptation requirements. In contrast, the optimal transformation selection was solely based on the evaluation of the applicability and adequacy expressions, and not on the specific adaptation profiles. For instance, the desktop L_{doc} handler could be chosen for any XHTML supporting desktop, and not only for the example desktop profile.

Summarising, the case study illustrated the feasibility of validating and adapting mixed namespace documents for an open set of adaptation profiles, according to independently developed processing semantics. Within the context of the sound processing scenario and the illustrated adequacy of the semantics definition process, it has illustrated the feasibility of all XMLPipe processing model aspects for the Web.

10.6 XMLPipe model discussion

This chapter introduced the necessary concepts for composing the complete XMLPipe preprocessing model and illustrating the feasibility of the proposed processing. The transformation and validation driver components orchestrate the individual XMLPipe sub-models to serve a preprocessing entity's transformation and validation requests.

The discussion of the implementation issues and pilot XMLPipe implementation provided the core foundation for implementing the XMLPipe preprocessing model. The case study used both the proposed theory and the XMLPipe implementation to successfully illustrate the feasibility of the proposed processing.

The following discussion investigates the adequacy of XMLPipe for the Web and extrapolates the XMLPipe observations to the complete processing domain, in order to support our hypothesis.

10.6.1 Framework based evaluation

The XMLPipe preprocessing model is adequate for the Web, because it consists of a set of sub-models that cover all preprocessing framework functionality and fulfill their corresponding framework requirements. The driving components, introduced in this chapter, are necessary for the XMLPipe preprocessing model, but they do not affect its adherence to the preprocessing framework. The XMLPipe sub-models, which were introduced in the previous chapters, covered all necessary preprocessing functionality. They are also adequate for the Web, because they either fulfil all their corresponding framework requirements or provide well balanced tradeoffs, if strict adherence to the framework can harm the processing generality. Consequently, the XMLPipe preprocessing model is adequate for the Web. Nevertheless, a further investigation of its adherence to the Web design principles and of its efficiency is feasible within the context of complete XMLPipe proposal, its pilot implementation and the introduced case study.

Both the XMLPipe sub-models and XMLPipe as a whole adhere to the Web design principles. Specifically, XMLPipe is as simple as possible, because it does not introduce redundant functionality that is not directly related to the preprocessing framework requirements. Furthermore, XMLPipe is modular, because its individual components interoperate closely, but they are clearly separated and their interoperation is well defined. XMLPipe also enables modular processing, because the semantics authors can independently develop the processing semantics of each language. Moreover, the proposed document processing is decentralised, because the XMLPipe binding model enables the distribution of all the necessary processing semantics. The test of independent invention is fulfilled, because the well defined atomic validation and transformation interfaces allow the seamless integration of a multitude of validation and transformation technologies. Finally, XMLPipe enables the design of languages that conform to the principle of least power, because it defines the processing of mixed namespace documents that combine an open set of languages. Consequently, it allows the combination of multiple languages that can cover minimal application domains.

The preprocessing framework requires computationally efficient processing of presentation documents. The proposed validation algorithm has been shown to be slower than single namespace validation, but of the same computational complexity order (both are $O(n \log(n))$). Consequently, the proposed XMLPipe validation can be characterised as efficient, considering that it enables the significantly more generic processing of mixed namespace documents that combine an open set of languages. As opposed to the validation, the multitude of factors that influence a document's transformation impede the investigation of its computational complexity, as described in Section 5.4. The exact nesting of a a document, and more precisely its average depth, can significantly influence the efficiency of its transformation, because of the recursive nature

of the transformation algorithm. Therefore, the proposed validation processing is efficient, but the computational efficiency of the transformation processing, which is the core of the XMLPipe processing model, cannot be proven.

However, the average transformation complexity can be reliably inferred from an experimental investigation that utilises the structural characteristics of presentation documents. Specifically, consider the case study example document, which is adequate for illustrating the XMLPipe processing functionality, as described in Section 10.5.1. An n-fold increase of its size would result in a similar increase of its number of nodes. However, its depth would not vary significantly, because typical size gains result from additional information (e.g. more sections and paragraphs) and not from increased presentation complexity (e.g. deep nesting of the presentation constructs). Therefore, documents that contain different number of copies of the case study document constructs are adequate for a reliable experimental investigation of the relationship between a document's size and the duration of its transformation.

A reliable experimental investigation must use a multitude of document sizes and perform multiple processing measurements for each size, in order to eliminate irrelevant variations, such as operating system processing overheads. Our experiment consisted of measuring the pilot implementation transformation time for 21 automatically generated documents, which contained from one to 2500 copies of the example document constructs (corresponding to a range of approximately 70 to 170000 document nodes). Each document was processed 10 consecutive times, and the minority of measurements that significantly deviated from the average, due to irrelevant outside factors, were discarded. Our experimental investigation was based on the arithmetic average of the remaining values.

Assessing the efficiency of a transformation approach requires a well defined comparison benchmark, but no existing approach offers comparable transformation functionality to XMLPipe, as described in Section 7.9. However, comparing the XMLPipe execution time to a single XSL-T atomic transformation is beneficial. XMLPipe incorporates several XSL-T atomic transformations and its processing is expected to be significantly slower than their processing, because of its enhanced processing functionality. However, the execution time comparison can illustrate whetеher XMLPipe is of higher computational complexity than simple atomic transformation and, under the assumption that current transformation approaches are adequate for the Web, illustrate whether XMLPipe is sufficiently fast for the Web.

A separate document must be used for the atomic transformation processing, because a stand-alone atomic transformer cannot adequately process the case study example document. Our experiment used a modified version of the example document that only contains L_{doc} constructs. The constructs of the remaining languages have been replaced by an equivalent number of L_{doc} constructs, and the number of handled constructs has been preserved, in order to preserve the document complexity.

Figure 10.13 illustrates the collected data and their statistical analysis. The two sets of data correspond to the measurements of the pilot XMLPipe implementation and the Apache Xalan XSL-T transformer, which is the transformer used by the XMLPipe's XSL-T atomic transformation. Each mark illustrates the average, minimum and maximum duration measurements for the corresponding input document. The data regression analysis results in the illustrated polynomial functions, which have been drawn over the data marks. The squared correlation coefficients (R^2) provide a reliable statistical measure of how well the polynomial functions match the experimental data. Both R^2 values indicate a nearly excellent match, and they are significantly greater than

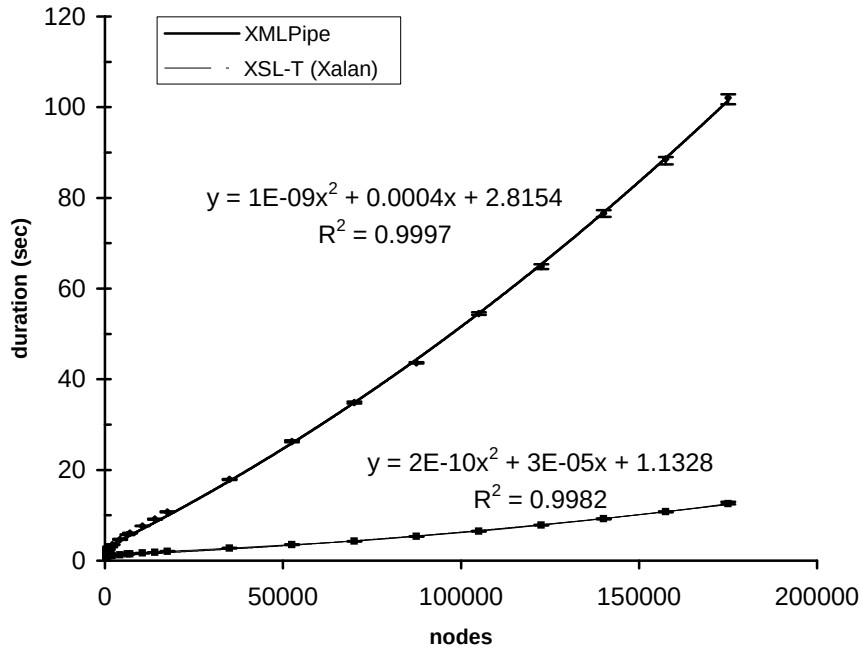

Figure 10.13: Transformation duration in relation to the document nodes

their counterparts for linear, logarithmic and exponential functions. Consequently, the described functions are a reliable approximation of the relationship between the number of document nodes and the processing time of both XMLPipe and Xalan XSL-T.

The first observation is that XMLPipe's computational complexity is $O(n^2)$, which could be considered as inadequate for the efficient processing of Web documents. However, the factor of n^2 $(1E-09)$ is significantly smaller than the factor of n $(4E-04)$. The illustrated experiment purposely used large documents[2] in order to enable the investigation of the processing tendencies, which are not apparent for typical document sizes. Nevertheless, for typical presentation documents, within the $[0, 20000]$ nodes range, XMLPipe behaves in a nearly linear fashion, because of the small factor of n^2.

Furthermore, the XMLPipe execution time is significantly slower that the single XSL-T transformation execution time, but they both belong to the same computational complexity class. Specifically, both regression functions are described by second degree polynomials, where the factor of n^2 is 5 orders less than the factor of n. Consequently, the two functions express the same class of computational complexity. The execution times relationship becomes more apparent by drawing the execution time ratio between XMLPipe and XSL-T. Figure 10.14 illustrates their execution duration ratio. Within the $[0, 60000]$ range the ratio is not fixed but it increases. Such an increasing ratio may result from memory allocation variations, since XMLPipe's memory footprint is significantly larger than the footprint of a single XSL-T transformation. However, for large documents, the ratio becomes fixed and is the same to the ratio between the factors of n^2.

[2]The largest document results in a 670 pages long presentation.

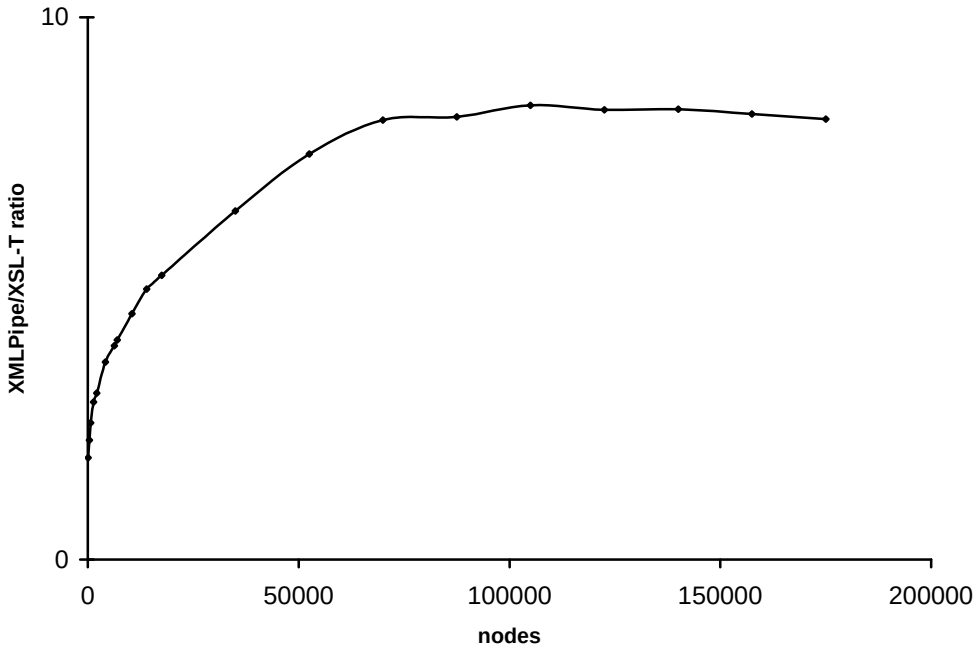

Figure 10.14: Transformation duration ratio between XMLPipe and XSL-T

Therefore, XMLPipe is slower than a single XSL-T transformation, but its computational complexity is of the same order. Considering its significantly more powerful processing, an approximately 10-fold increase of the transformation time can be considered as acceptable. Consequently, XMLPipe fulfils the preprocessing framework efficiency requirement, because our investigation does not illustrate any computational efficiency problems and it can efficiently process large presentation documents.

Both the complete XMLPipe preprocessing model and its individual components fulfill the preprocessing framework requirements. Therefore, the XMLPipe proposal allows the processing of presentation documents that combine an open set of presentation languages, in a way that is adequate for the Web.

10.6.2 Hypothesis support

Our hypothesis was that the development of generic processing models is feasible within the environment of the Web, if they utilise the constraints of the presentation domain and address the problem of document processing as a whole. Such a hypothesis can be supported by firstly developing a sufficiently generic processing model and subsequently illustrating that its sufficiency stems from following the principles expressed in our hypothesis. Consequently, the support of our hypothesis would benefit from a complete processing model that addressed all presentation document processing aspects: validation, transformation and rendering.

A preprocessing model, such as XMLPipe, does not include a rendering sub-model, but it is an adequate first step towards generic document processing and it is sufficient for supporting our hypothesis. A preprocessing model is simpler than a complete processing model, because it reuses existing presentation functionality and does not

require the time consuming development of a generic rendering component. Moreover, it assists the transition from existing processing technologies, because it both allows the document users to keep their chosen Web browsers and can be used on both the server and the client side. Current XML processing approaches are not sufficient for the generic processing of presentation XML documents. Therefore, a preprocessing model can sufficiently support our hypothesis, if it can reuse existing browsers to provide such generic processing. Additionally, more preprocessing model observations can be extrapolated to generic processing models, in order to further illustrate the soundness of our hypothesis. This section will use the proposed XMLPipe model to support our hypothesis.

XMLPipe fulfils the preprocessing framework requirements, as described in Section 10.6.1. Consequently, XMLPipe is an adequate preprocessing model for the Web, because the proposed framework provides a sound indication of a preprocessing model's adequacy, as described in Section 4.5

Addressing the processing problem as a whole has been essential for the adequacy of XMLPipe. The proposed XMLPipe model combines the integration, validation, transformation, adaptation and binding sub-models. The functionality of their composition is greater than the sum of their individual processing capabilities, because removing a sub-model would result in more unfulfilled framework requirements than the ones that correspond to it. For instance, consider that XMLPipe did not contain the proposed distributed binding sub-model. The resulting model, in addition to not fulfilling the binding requirements, would also impede the independent definition of all necessary semantics and the processing of an open set of languages. In a similar manner, the lack of an adaptation model would not only prohibit the document adaptation, but it would also harm most XMLPipe functionality, because there would be no method to choose the optimal transformation for each subtree. Consequently, addressing the problem of document processing as a whole is integral to the adequacy of XMLPipe.

Utilising the constraints of the presentation domain, has also been essential to XMLPipe's adequacy. The XMLPipe integration model uses the handled construct observations, which are specific to presentation documents, in order to define the validity of mixed namespace documents. The validation and transformation models, which are the core XMLPipe sub-models, use the integration model to enable the processing of mixed namespace documents. If XMLPipe was not based on such observations, it would either require predefined integration profiles or allow overly generic integration, such as the unconstrained integration of NRL and NVDL. Such approaches would result in a deficient preprocessing model, because the former is not adequate for the Web and the latter does not provide the a sufficient foundation for presentation document processing (it does not define the necessary relationships between the individual subtrees, as described in Section 5.4).

The adequacy of XMLPipe for the Web illustrates the soundness of our hypothesis, within the preprocessing domain. According to the above discussion, both principles expressed in our hypothesis have been integral to the adequacy and feasibility of XMLPipe. Consequently, the design of generic preprocessing models is feasible when they address the problem as a whole and utilise the constraints of the presentation document processing domain.

XMLPipe also illustrates the applicability of principles expressed in our hypothesis beyond the document preprocessing domain, because it enables generic document processing. Its combination with existing browsers allows rendering presentation documents that combine an open set of languages, according to an open set of adaptation

requirements. Such processing results from the XMLPipe preprocessing functionality and not from the browser's presentation engine, because no current browser provides sufficiently generic document processing. Consequently, the application of the principles expressed in our hypothesis enables the generic processing of presentation documents.

The application of the principles expressed in our hypothesis to an original rendering sub-model can further enhance the provided generic processing. A rendering sub-model must support a set of natively presentable languages and a corresponding presentation integration model. The former can benefit from utilising the presentation domain constraints, because it must cover a significant portion of the presentation functionality spectrum. The latter can either use or extend the XMLPipe integration model, which is based on the presentation domain constraints. Additionally, a rendering component is not sufficient in itself, and a processing model benefits from addressing the document processing problem as a whole. For instance, the rendering model can interoperate with a transformation model, in order to process non natively supported constructs and to allow presentation time transformations. A rendering model can also interoperate with a validation model, to ensure the validity of a document prior to its presentation. Furthermore, a generic binding model can be used to allow an extensible set of presentation languages. Consequently, our hypothesis also applies to the development of complete processing models.

Summarising, XMLPipe has illustrated our hypothesis for both the preprocessing and complete processing domains. Specifically, it enables the generic preprocessing of presentation documents, because it follows the principles expressed in our hypothesis. Additionally, its combination with existing rendering components enables the generic processing of presentation documents. Furthermore, the principles expressed in our hypothesis can be applied to a complete processing model, in order to provide richer processing functionality. Consequently, our hypothesis is correct.

> *The presentation domain is sufficiently constrained to allow the development of generic processing models. Such processing models can be developed by utilising its constraints and addressing the document processing problem as a whole.*

10.7 Summary

This chapter concluded our proposal by completing the XMLPipe model and illustrating the soundness of our hypothesis. Specifically, it introduced the validation and transformation drivers, addressed the core XMLPipe implementation issues and described our pilot XMLPipe implementation. An extensive case study illustrated the feasibility of XMLPipe and its adequacy for the Web, in combination with a preprocessing framework based evaluation. The discussion section extrapolated the XMLPipe preprocessing observations to the whole spectrum of presentation document processing and illustrated the correctness of our hypothesis.

The introduced validation and transformation drivers were necessary, because they provided a processing interface to the external entities. The validation driver receives the authoring validation request from the document author and uses the document parser and integration model validation driver to fulfill it. In a similar manner, the transformation driver receives the transformation requests by the preprocessing initiation entities, and uses the document parser, the adaptation component and the

integration model transformation driver to fulfill them.

The resulting XMLPipe model provides a detailed description of all processing aspects, but it does not cover all XMLPipe implementation issues. Existing presentation integration models are overly restrictive and they impede the generic document processing, because they only allow single namespace documents or profile based composition. The proposed short term solution consists of using minimal adaptation profiles and allowing a document's root transformation to generate DOCTYPE declarations. The former ensures that XMLPipe output documents use a minimum set of languages. The latter ensures that the resulting documents are adequately presented. The XMLPipe model did not define the representation of the processing semantics, because an adequate representation is not the main focus of this thesis. The pilot implementation establishes a representation that is adequate, but not necessarily optimal. Finally, the recursive processing of document subtrees according to a distributed set of specifications results in losing the majority of necessary context information for error reporting. The proposed solution is to associate the necessary location information with the document nodes and preserve it by serialising it prior to each subtree transformation.

The document processing case study used the pilot implementation to illustrate the feasibility and benefits of the proposed processing. It illustrated the processing of the transformation driving example document for three adaptation profiles, which cover a wide spectrum of adaptation capabilities. Initially, it described all the necessary processing semantics: the handled construct information, the validation semantics and the transformation semantics. Subsequently, the pilot implementation was used to transform the case study document, according to the three profiles. Moreover, the case study illustrated that the proposed recursive processing allows the straightforward introduction of additional languages.

The proposed XMLPipe model allows the generic preprocessing of presentation documents and enables the illustration of the correctness of our hypothesis. It only covers document preprocessing and does not address document rendering. However, its combination with existing browsers can provide generic document processing. Furthermore, the XMLPipe concepts can be applied for the development of a complete processing model that allows additional processing functionality. The adequacy and power of XMLPipe prove our hypothesis, because they are a direct consequence of following the two principles expressed in our hypothesis: addressing the problem as a whole and utilising the constraints of the presentation domain.

Chapter 11

Future research and concluding remarks

This thesis established the importance of presenting XML documents, described a generic XML preprocessing model and provided the foundation for the development of future processing proposals. Our investigation started with the literature review, which indicated that current processing approaches cover a wide spectrum of functionality, but they are not adequate for the generic processing of presentation documents within the Web. We hypothesised that generic document presentation can be achieved by utilising the constraints of the presentation processing domain and addressing the problem of document presentation as a whole. A preprocessing framework was the first step towards supporting our hypothesis, because it established the necessary functionality for adequately preprocessing documents within the Web. The XMLPipe preprocessing model proposal applied the principles expressed in our hypothesis within the preprocessing domain, and it enabled the validation and transformation of mixed namespace presentation documents that combine an open set of languages. Its evaluation using the preprocessing framework and a comprehensive case study illustrated that it is both feasible and adequate for the Web, because it follows the principles expressed in our hypothesis. Finally, the XMLPipe preprocessing observations where extrapolated to cover the complete spectrum of presentation document processing and prove the proposed hypothesis.

This chapter concludes this thesis by further investigating our hypothesis, summarising the contributions of this thesis and describing a set of future research proposals.

11.1 Expressing and supporting the hypothesis

Chapter 2 described a review of the existing XML processing approaches that inspired the formulation of our hypothesis. The literature review illustrated that existing approaches provide a wealth of XML processing functionality, but they only address the individual processing problems separately and outside the context of a generic processing model. No existing approach can combine the individual solutions to provide adequate generic document processing for the Web. Our hypothesis was based on the observation that there are no generic processing approaches, because the necessary functionality set is not well defined. This is a consequence of the lack of well established XML processing domain boundaries, which could allow the definition of the required

functionality.

The domain of XML document presentation can be overly generic as the foundation of a generic processing model. XML documents can represent any information, and their presentation can be arbitrarily complex and depend on the context of their use. In order to assist the development of a generic processing model, we restricted the scope of this thesis to the processing of presentation documents. Presentation documents can also require arbitrarily complex presentations, but their presentation semantics are well defined and do not depend on their context. Additionally, presentation document processing is the necessary first iteration towards the presentation of all XML documents, because the presentation of any XML document can be expressed by a presentation XML document.

The formulation of our hypothesis was based on the assumption that the domain of processing presentation documents is sufficiently constrained to facilitate the development of generic processing models. Presentation documents can express any type of information, but the common characteristics of their processing can enable the definition and development of all required processing functionality. Therefore, we hypothesised that processing approaches can utilise the constraints of the presentation document processing domain to addresses all processing issues, in a way that is adequate for the Web.

We supported our hypothesis by investigating the presentation document preprocessing domain, which further restricts the scope of this thesis to approaches that do not include a rendering component. Specifically, none of the preprocessing framework, XMLPipe and the case study addressed the rendering of presentation documents (apart from the example rendering by existing browsers). However, they established the soundness of our hypothesis in the preprocessing domain, in such a way that also allowed the extrapolation of our observations to cover the whole spectrum of presentation document processing. Additionally, preprocessing approaches are beneficial as a first step towards generic processing, because the lack of a rendering component simplifies their implementation and they can be adopted without requiring the users to change their browser of choice.

A preprocessing framework was the first step towards supporting our hypothesis. The necessary preprocessing functionality is not self-evident, because the preprocessing domain is insufficiently constrained and the Web processing requirements can be ambiguous. Supporting our hypothesis for the preprocessing domain required a well defined method to evaluate the adequacy of preprocessing approaches. The preprocessing framework provided such a method by applying the Web design principles, within the preprocessing domain, to identify the necessary preprocessing functionality and partition it into several interoperating components.

The XMLPipe preprocessing model provided the core support of our hypothesis, since it enabled generic document preprocessing by utilising the presentation domain constraints and addressing the problem as a whole. Specifically, the XMLPipe model addressed the preprocessing problem as a whole, because its individual sub-models covered all preprocessing aspects. Additionally, the XMLPipe integration model utilised the presentation domain constraints to define the interpretation of mixed namespace documents, because it was based on a set of presentation document structure observations. The preprocessing framework based evaluation of XMLPipe illustrated that the proposed sub-models and their composition is adequate for transforming and validating presentation documents for the Web. Additionally, the described case study illustrated

the feasibility of the proposed processing. XMLPipe supported our hypothesis, because its adequacy stems from following the principles expressed in the hypothesis, as described in the previous chapter.

The final step towards supporting our hypothesis was the extrapolation of the preprocessing observations to the domain of generic presentation document processing. Such an extrapolation was feasible, because the combination of XMLPipe with existing browsers allows the generic processing of presentation documents, which is not feasible by the browsers alone. Furthermore, the principles expressed in our hypothesis can be applied to further enhance the provided processing functionality by the introduction of a generic rendering component and its interoperation with the proposed preprocessing components.

11.2 Contributions

The core contribution of this thesis to the XML document processing domain is the XMLPipe preprocessing model and both the formulation and support of our hypothesis. Additionally, each XMLPipe sub-model can also be considered as a separate contribution, because the XMLPipe sub-models have been shown to significantly outperform existing approaches.

11.2.1 XMLPipe and the hypothesis

XMLPipe is a significant contribution, because it offers necessary document processing functionality that is not sufficiently covered by existing approaches. Specifically, it provides adaptive transformation and validation of mixed namespace documents that combine an open set of languages, according to a distributed set of processing semantics. XML allows the creation of new languages and language authors will continue to create new languages that cover a multitude of information domains. Therefore, the transformation of documents that combine an open set of languages is necessary for mapping them to their natively presentable interpretations. Document transformations must be adaptive, because they must accommodate for the multitude of users, devices, and browsers that can access Web documents. Document validation is not explicitly related to their presentation, but it assists the process of document authoring and the development of document transformations. The use of distributed semantics is essential for the development of languages and their processing definitions, since centralised approaches are not adequate for the distributed nature of the Web.

The support of our hypothesis is the second core contribution of this thesis, because it provides the foundation for developing generic XML processing models. Specifically, XML documents can cover a broad spectrum of information and may require arbitrary presentation functionality. The support of our hypothesis illustrated that the presentation domain is sufficiently constrained to allow the development of generic processing models for presenting XML documents. Furthermore, the hypothesis establishes the two core principles for developing such models: addressing the problem of presentation as a whole and utilising the presentation domain constraints. The XMLPipe preprocessing model has illustrated the value of our hypothesis, since it used these two principles to provide generic document preprocessing, which is significantly more powerful than existing approaches.

11.2.2 The preprocessing framework and the individual sub-models

The proposed preprocessing framework allows the evaluation and assists the development of preprocessing approaches. Such a framework is necessary, because the required preprocessing functionality is not self-evident and there are no existing specifications that define it. The proposed framework design was based on an application of the Web design principles to the presentation document preprocessing domain, which provided a sound set of functionality requirements. The core preprocessing framework purpose was to allow the evaluation of XMLPipe. Inversely, XMLPipe also illustrated the adequacy of the preprocessing model, because the adoption of the framework component separation resulted in the generic preprocessing of presentation documents. Consequently, the preprocessing framework can be considered as a separate contribution, because it can assist the development and evaluation of presentation document processing approaches.

The XMLPipe validation and transformation models provide the core XMLPipe functionality. However, the remaining XMLPipe sub-models also contribute separately to the XML processing domain.

This thesis introduced the concept of integration models that infer the interpretation of mixed namespace documents. Existing integration approaches are either based on predefined integration profiles or the arbitrary composition of valid document subtrees. The former are not adequate for the composition of an open set of languages. The latter do not provide the necessary foundation for document processing, because the interpretation of mixed namespace documents remains undefined. In contrast, integration models, such as the proposed XMLPipe integration model, define the necessary document interpretation for document authoring and processing, within a constrained application domain. For instance, both XMLPipe validation and transformation models require the foundation of the XMLPipe integration model, in order combine the processing semantics of the individual languages. Furthermore, document authors must follow the expressed integration guidelines to ensure the well defined interpretation and processing of a document. Consequently, the introduction of the integration model concept is beneficial for the generic processing of mixed namespace documents.

In addition to the general concept of integration models, the XMLPipe integration model in specific is also a separate contribution. Specifically, the handled construct observations, the valid documents definition and the subtree separation process provide the foundation for processing presentation documents by either the XMLPipe or other processing models. The proposed integration model is not the only adequate alternative for defining the interpretation of presentation documents, since other integration models might devise different integration rules. However, it is sufficient for allowing generic validation and transformation of mixed namespace presentation documents.

The concept of *semantic correctness*, as opposed to validity, is also original within this thesis. Existing processing approaches only consider document validity as an indication of a document's correctness. However, it is not adequate for evaluating the correctness of documents that contain structure modification constructs. The disambiguation between the validity and semantic correctness of a document allows to overcome the *SMC* construct processing issues, in order to evaluate whether a document has well defined interpretation.

The core contribution of the XMLPipe adaptation model is the well defined method to select an optimal specification over a set of independently developed alternatives. Such a selection is essential for enabling both adaptive processing and distributed development of processing semantics. Existing approaches, such as the CC/PP based

stylesheet selection[OH02], allow the application of alternative stylesheets, according to a set of adaptation requirements. However, they neither allow sufficiently expressive queries nor can select between independently developed specifications, because the choice depends on document-specific custom logic. In contrast, the XMLPipe binding adaptation specifications and the adaptation measure evaluator allow the optimal specification selection over a set of independently developed alternatives, according to declarative adequacy specifications. The proposed selection cannot be guaranteed to always provide the correct alternative. However, consistent design of the transformation binding specifications can allow the identification of the optimal or near-optimal alternative, in the majority of cases.

The XMLPipe composite profiles are also a significant adaptation model contribution. Specifically, existing approaches, such as CC/PP, allow the extensible representation of adaptation requirements, but they do not define their corresponding processing. In contrast, the XMLPipe adaptation model defines both the representation and the processing for an extensible set of adaptation requirements. The core composite profile contribution is the profile conflict resolution mechanism, which can resolve conflicting adaptation statements without requiring predefined relationships between the entities that specify the statements. Such a resolution mechanism is necessary for the Web, because there is not fixed set of entities that can influence the processing of a document.

The proposed binding model bridges the individual models with their corresponding semantics and allows their distribution, according to principal and secondary semantic repositories. Such a distributed binding model is essential for processing an open set of languages, without requiring inline document processing information. The described location mechanisms are not original in themselves. However, their application to locate all necessary XML processing semantics is innovative and enables the liberal introduction of XML languages and their use within mixed namespace documents.

11.3 Future research

This thesis has sufficiently covered the presentation domain to illustrate the feasibility of generic presentation document *preprocessing* and to support our hypothesis. The introduced concepts have illustrated an original document processing paradigm that can cover the processing requirements of unrestricted information domains, such as the Web. This section outlines the most significant extensions and future research ideas inspired by this thesis.

11.3.1 XMLPipe extensions and optimisations

The current pilot XMLPipe implementation is not complete and it only covers the necessary functionality for illustrating the feasibility of generic document preprocessing. Therefore, the most imminent extension of our work is to implement the remainder of the XMLPipe model. Specifically, the additional required components must cover the proposed validation, binding and profile composition functionality.

Additionally, XMLPipe could be integrated into a browser, in order to provide a more intuitive interface to document preprocessing. A document user would not have to separately request a document's preprocessing and rendering. In contrast, the browser would automatically invoke XMLPipe to retrieve the optimal document interpretation and, subsequently, present it to the user. Additionally, the browser interface can significantly simplify the specification of the adaptation requirements,

since the document user would no longer have to specify a composite profile and the browser could automatically compose it out of its browser-specific settings.

This thesis has not thoroughly investigated the representation of the processing semantics. As described in the previous chapter, a well defined, concise and easy to use processing semantics representation is necessary for the wide adoption of a pre-processing architecture. The proposed semantics representation covers all necessary information, but it is not necessarily optimal and it could be improved. For instance, a more compact adaptation expressions syntax would significantly assist the semantics authoring process, because the proposed tree-based syntax is overly elaborate. Furthermore, XMLPipe semantics use several named entities, such as atomic transformations, in addition to the core URI identified resources, such as handled constructs and languages. The proposed representation requires that named entities are identified by limited scope names and are declared prior to their use, within each processing semantics document. The reduced scope enables the use of non universally-unique short names, which are essential for their frequent use in pipeline specifications. However, such a method requires multiple declarations of the same entities, which can result to inconsistencies. An alternative method would be to use URIs for the unique identification of all entities. URIs would allow the location of all necessary definitions by the proposed binding model, in a similar manner to the location of all processing semantics. Additionally, URI-based entity references can be sufficiently short, if namespace prefixes are allowed.

XML processing approaches must be computationally efficient, in order to allow the timely processing of potentially large documents. The transformation and validation investigations concluded that the computational complexity of both processes is adequate. Specifically, the computational complexity of transforming presentation documents is the same as the complexity of simple XSL-T transformations. Moreover, the computational complexity of mixed namespace validation is of the same order as the validation of single namespace documents. However, both processes are significantly slower than existing approaches. Both the pilot implementation and proposed theory focused principally on the feasibility of generic document processing and not on its efficiency. Consequently, since the computational complexity investigation did not indicate fundamental complexity problems, significant efficiency improvements can be achieved by combining a set of algorithm optimisations and a more efficient implementation.

Firstly, the execution time of atomic transformation wrappers can be significantly reduced, if XMLPipe-specific implementations of the individual transformation technologies are used. Specifically, the incorporation of most existing transformation technologies requires time consuming subtree separation, because both their input and output are XML documents, as opposed to document subtrees. DOM implementations associate each document node with a corresponding *document node*. Consequently, detaching a subtree's parent from its context is not sufficient, and each subtree document node must be separately imported into a new document, prior to the transformation application, and subsequently re-imported to the original document. If atomic validations were directly transforming document subtrees, without requiring their separation and re-insertion, they could result to significantly reduced transformation time. A drawback of such an approach is that most atomic transformations would have to implement all necessary functionality, instead of using existing implementations. However, the development of such atomic transformations can be gradual. Furthermore, if XMLPipe is widely accepted, third party implementations could provide the necessary interfaces for efficient subtree processing.

Further efficiency improvements can be achieved by limiting the duplicate processing. For instance, the transformation driver must use the adaptation model to choose the optimal transformation, for each document subtree. This process requires retrieving all alternative semantics and evaluating their corresponding adaptation expressions. The selection result remains the same, if the adaptation requirements and the set of available specifications are not modified. In most document processing scenarios, these parameters remain fixed, during the execution of multiple consecutive transformations. Consequently, the time consuming process of optimal semantics selection can be avoided for most subtrees, if XMLPipe caches the optimal transformation selection for each handled construct.

Additional efficiency benefits can be achieved by reducing the recursive document traversals of *transformRev*. Specifically, *transformRev* locates the document handled constructs by traversing all document nodes and all transformation result nodes. A first level reduction can be achieved by reducing the number of separated subtrees. For instance, if a handled construct rooted subtree contains other handled constructs and they are all associated with the same optimal transformation pipeline, *transformRev* could transform all subtrees together. Subsequently, *transformRev* would only traverse the resulting nodes once and proceed to the remainder of the document. Separate subtree transformations would require traversing the resulting nodes more than once. Additionally, *transformRev* can avoid traversing a transformations' result when no further processing is necessary, if adequate transformation meta-information is available. For instance, if each transformation is associated with the list of handled constructs it produces, it can be a priori decided whether traversing its output is necessary.

11.3.2 Transformation model extensions

The proposed postorder subtree transformation is adequate for most integration cases, because it ensures that each subtree transformation can access all necessary content and that the relationships between *FOC* constructs and their context is not harmed. However, no predefined order can be adequate for all integration cases. For instance, consider that the semantics of an *SMC* construct is to substitute itself with a portion of its descendants, which is specified by an XPath expression. The processing of such a construct must be performed prior to any transformation of its descendants. However, if its descendants contain handled constructs, *transformRev* will process them first. The fixed postorder traversal, provided by *transformRev*, is not adequate for processing such constructs.

A transformation model extension can allow custom processing order, but it may complicate the design of the subtree transformations. For instance, *transformRev* could allow the individual transformers to specify whether the corresponding subtree must be processed before or after the processing of its descendants. Such information can be provided by either the transformation semantics associations or the transformation specifications.

The proposed transformation selection method could be extended to further support transitional language mappings. For instance, consider two transformations T_1 and T_2 that map the constructs of L_1 to L_2 and the constructs of L_2 to L_3, respectively. Additionally, consider a document d that contains constructs of L_1 and an adaptation profile pr that only declares support for L_3. The XMLPipe transformation model cannot adapt d for pr, because there is no direct transformation to L_3. However, *transformRev* could successfully process d by applying both T_1 and T_2. A

transformation model extension could allow such processing by incorporating intermediate transformations to the selection process. For instance, it could represent the transformation relationships using a graph and compute the optimal transformation route for each document subtree, which can consist of one or more transformation applications.

11.3.3 Adaptation model extensions

The introduced binding adaptation specifications enable the fine grained association of transformation specifications with their corresponding adaptation requirements. However, adaptation expressions can be extended to allow more precise transformation selection. Firstly, a wider set of term data types and operators is necessary for versatile adaptation expressions. Secondly, the adaptation expressions could be extended to access the transformed subtree. Such expressions can assist the transformation of a subset of handled constructs, such as the case study `doc:img` element. Specifically, for the mobile adaptation profile, the case study transformation semantics will always use the WBMP converter to process the `doc:img` element, independently of the referenced image. If document access was available, the transformation selection process could also take into account the referenced image and only invoke the WBMP converter for non-WBMP images. Such extended adaptation expressions can be straightforwardly introduced in the applicability expressions. However, special care is required for any adequacy expression extensions, in order to preserve their comparable nature: each adequacy expression must correspond to exactly one adaptation requirement.

The proposed adaptation model did not incorporate the existing CC/PP representation. CC/PP support would significantly increase the adaptation model's complexity, because of the multitude of necessary relevant technologies. Additionally, the unordered CC/PP statements are inadequate for the proposed conflict resolution mechanism. Nevertheless, a CC/PP extension to XMLPipe would assist the interoperation with several existing technologies, since CC/PP is the most commonly used adaptation requirements representation. Such an extension must specify the necessary relevant technologies and adapt the proposed conflict resolution to the unordered CC/PP statements.

11.3.4 Validation model extensions

The places where content is expected is the only XMLPipe processing information that is not explicitly specified by the XMLPipe semantics. The XMLPipe integration model chapter described two methods for identifying such places. The proposed validation model only used the explicit identification method, in order to avoid the inherent complexity of also supporting the heuristic identification method. However, a minor extension to the validation interface and the subtree separation process can allow the incorporation of several identification methods, without necessarily increasing the complexity of the atomic validations. Specifically, the atomic validation interface can be extended to provide information on the types of supported identification methods. *validate* can query the atomic validations and use the most precise available identification method. For instance, if an atomic validation supports the proposed explicit identification method, it is sufficient that *validate* introduces the predefined `coc` constructs. Otherwise, if only the heuristic method is supported, *validate* should use the atomic transformation interface to retrieve the set of constructs that accept arbitrary

content. *validate* could then use this information to test the valid nesting of the subtrees, without introducing the predefined `coc` constructs.

A more substantial extension of the validation model would be to omit the subtree separation altogether and use a composite super-schema to validate each document. Such an extension would require a common schema language that covers all necessary validation functionality. As described in Chapter 2, no existing validation technology covers all existing validation functionality. However the development of such a schema language is feasible, if it uses the foundation of tree automata, because they can cover the validation functionality of all existing validation approaches, as described by [MLM01]. If the validation model incorporates such an extension, the atomic validations must map the existing schema specifications to the common schema language. A top level validator can combine all the specifications into a schema for the mixed namespace document and, subsequently, validate the document as a whole, without separately processing its individual subtrees. Such a validation model can complicate the atomic validation development, but it can significantly improve the validation efficiency and also provide the foundation for a new generic language that covers all existing validation functionality.

11.3.5 Integration model extensions

The XMLPipe integration model defines the interpretation of mixed namespace presentation documents by incorporating presentation document specific observations to the NRL/NVDL subtree separation concept. Its adequacy has been indirectly illustrated by the feasibility and adequacy of XMLPipe for mixed namespace validation and transformation. A more explicit investigation could utilise the multitude of existing presentation languages. For instance, the existing standardised presentation languages cover a broad spectrum of presentation functionality. An investigation of how they fit within the XMLPipe integration model and how their resulting integration compares to existing integration profiles could provide further insight into the adequacy of the proposed integration model.

The XMLPipe integration model utilised the handled construct observations to define the valid integration of independently developed languages, without requiring inter-language interoperation. Additional observations can allow enhanced integration accuracy. For instance, at an adequate abstraction level, a predefined set of presentation data types and interfaces may sufficiently cover the presentation document domain. Such data types would allow more precise definition of which documents are valid. For instance, each construct could be associated with the data type it introduces and the set of descendant data types it can accept. Additionally, well defined interfaces can enable powerful interoperation between independently developed transformers. Such an approach could require significantly more complex semantics specifications. Furthermore, it can only be adequate for generic document processing, if all introduced data types and interfaces are derived from presentation domain constraints; otherwise, they would restrict the document processing and the set of supported languages.

In addition to extending the XMLPipe integration model, XMLPipe can benefit from supporting multiple integration models in each document. Specifically, the current XMLPipe model requires that all documents follow the same integration model, which is tightly coupled with the proposed processing. The support of multiple integration models can allow well defined intermediate processing states, more fine grained

integration and provide the foundation for more generic document processing. Specifically, during the proposed document transformation, the processed document is in an intermediate state that combines both the XMLPipe and the target integration model. A method to combine integration models would allow such states to be well defined, without requiring the introduced assumption that the target integration model is less generic than the XMLPipe integration model. Additionally, a single integration model cannot cover all integration cases. An integration model combination method can extend the applicability of a processing model by allowing the use of the optimal integration model, for each document portion. Furthermore, it can provide the foundation for extending XMLPipe outside the presentation processing domain. For instance, a document can combine both presentation and non-presentation constructs, where the former are integrated according to the XMLPipe integration model and the latter according to another data-oriented integration model.

However, support for multiple integration models requires addressing several issues: the association of integration models with the document portions, the description the associated processing in a device independent manner and the interoperation of the separate integration models. Only the first issue can be straightforwardly addressed by the combination of unique integration model identifiers and a predefined construct, such as an XML attribute, that can associate all document portions to such identifiers. The second issue can be addressed by either remotely retrievable processing drivers or requiring the atomic transformations to perform the necessary integration. The major obstacle is that most document processing aspects are tightly coupled with the underlying integration model. For instance, the proposed validation and transformation models are integration model specific, their interoperation is related to the integration model, and the transformation semantics binding is based on handled constructs, which are an integration model concept. Devising a method to dynamically incorporate an integration model into a processing model is a difficult problem. Finally, the a solution to the last issue can be based on investigating the constraints of the integration model domain, in a similar way as the XMLPipe integration model was based on investigating the constraints of the presentation processing domain. However, it is not clear if such constraints exist.

11.3.6 Binding model extensions

The XMLPipe binding model precisely defined the principal location mechanism, which is based on RDDL. However, it purposely left the secondary location mechanisms undefined, in order to avoid imposing any functionality restrictions. However, a further investigation of the secondary location mechanisms could allow more fine grained definition of the interoperation between the several repositories and the binding model. Additionally, it can provide the foundation for a more powerful semantics selection mechanism than the simplistic list of trust levels.

The proposed document processing raises several security and trust issues that must be addressed before its wide deployment. XMLPipe processes XML documents by combining distributed semantics specifications, atomic validation and atomic transformation implementations. A first level security extension would be to substitute the simplistic set of predefined trust levels with distributed trust networks, which are a similar approach to the accustomed concept of trusted certification entities. Specifically, each XMLPipe implementation instance can have a list of fully or partially trusted entities, which can in turn have further sets of trusted entities. Each implementation

can combine all known trust sets into a trust network, and use it to decide whether a specification can be trusted. Additionally, the resulting trust levels can be fed to the semantics selection process to enable a trust-level sensitive process of choosing the optimal transformations. Furthermore, the trust levels can be used to control the execution of the downloaded code, which can run in a applet-like sand-box, if it is not sufficiently trusted. This thesis only focused on illustrating the feasibility of generic document processing, because the time and space constraints did not allow the incorporation of a security mechanism. However, the above ideas can be used to support secure processing of presentation documents, according to a distributed set of semantics.

11.3.7 Towards a complete processing model

This thesis focused on document preprocessing, but a complete processing model that included a rendering sub-model would have provided more powerful processing of presentation documents. Additionally, it would have directly supported our hypothesis, without requiring to extrapolate the XMLPipe preprocessing observations to the complete processing domain. Nevertheless, the combination of the literature review (in Chapter 2), our hypothesis and the XMLPipe processing concepts can provide the foundation for extending XMLPipe to a complete processing model.

An adequate rendering sub-model must, at a minimum, address the issues of establishing a sufficient set of natively supported languages, incorporating imperative and declarative behaviour descriptions and interoperating with the preprocessing sub-models.

A sufficient and well defined set of natively supported languages ($\mathcal{L}_p$) is necessary, because it defines the rendering sub-model's interface. An adequate $\mathcal{L}_p$ can be composed out of either existing presentation languages or minimal feature-based languages. Specifically, existing standardised presentation languages are adequate $\mathcal{L}_p$ candidates, because they cover a wide spectrum of presentation functionality. Most existing languages are overly broad and may include redundant constructs, but their recent partitioning into loosely coupled modules allows their use within an adequate $\mathcal{L}_p$. Alternatively, an $\mathcal{L}_p$ definition can be based on the composition of new minimalistic languages that correspond to a set of presentation features. Table 2.3 (page 29) has summarised three proposals of the necessary features for presenting Web documents.

The definition of $\mathcal{L}_p$ also requires a corresponding integration model that defines the combination of all natively supported languages. The proposed XMLPipe integration model can be used, because the natively supported languages of a rendering sub-model are also presentation languages. However, a more specific integration model can utilise the well defined set of languages to provide more fine grained construct associations and allow inter-language interoperation.

Native support for dynamic functionality descriptions, such as scripting and numeric constraints, assist the development of precise presentation descriptions. Such descriptions are essential for rich presentation functionality and for adequately extending the $\mathcal{L}_p$. The former is true, because they can be used to create highly customised presentations by customising and combining the constructs in $\mathcal{L}_p$. As Section 2.4.2 described, plug-ins and applets can extend a browser's set of natively supported languages, but they do not allow the integration of the extensions within the common underlying presentation model. In contrast, the combination of functionality descriptions with behaviour binding approaches, such as XBL and RCC, enables the introduction of $\mathcal{L}_p$ extensions that are seamlessly integrated with the languages in $\mathcal{L}_p$.

Furthermore, supporting declarative behaviour descriptions, such as numerical constraints, enhances a rendering sub-model's integration and adaptation capabilities. They provide a high level description of the interoperation between separate presentation objects, without requiring the use of restrictive predefined interfaces. Additionally, they can enhance the adaptation capabilities of a rendering model, because as high level descriptions they increase its adaptation range. The rendering model's constraint solver can combine all document descriptions, in order to find an optimal solution for a given set of adaptation requirements. The core challenge for supporting such declarative behaviour descriptions is to identify the necessary types of descriptions and integrate their corresponding solvers within a single presentation model, in a way that is adequate for the Web.

Finally, a rendering sub-model must not be investigated independently, but in combination with all other sub-models. As described in our hypothesis, the problem of document processing must be addressed as a whole, because the functionality sum of the individual models is a subset of their combined functionality. Specifically, the interoperation between the rendering, transformation and validation models is necessary for preparing a document prior to its presentation, as described in Section 10.6.2. Additionally, the introduction of a rendering model changes the document presentation process from a one way transformation to a bidirectional process, where the transformation model can feed a document into the rendering model and vice-versa. For instance, the transformation model can provide a preprocessing document, in order to present it to the user. Inversely, a script, a constraint solver or the document user can introduce document modifications that require further transformation prior to their rendering.

11.3.8 Beyond presentation documents

The core presentation document processing ideas can be extended to non-presentation documents, if their processing context is well defined. The processing of a non-presentation document can either present it or perform some other form of processing. In the first case, its presentation depends on its context, because non-presentation documents can be associated with several presentation semantics. In the latter case, the desirable form of processing can depend on the application that accesses the document. If such context information is well defined and the introduced processing domains are sufficiently restrictive, generic processing models may be feasible.

Consider the presentation of documents that include constructs of non-presentation languages. The presentation of such constructs is not a priori defined, and they can be mapped to different presentations, according to their context. A simplistic method to present them would be to allow the explicit association of transformation specifications with the individual document subtrees. The transformation of each subtree can map all its non-presentation constructs into their adequate presentable interpretation for the given document context. However, such an approach is restrictive, because it relies on the document author to provide the necessary processing information. It can also inhibit the integration of multiple languages, because document specific stylesheets have the same drawbacks as predefined integration profiles. Further investigation of the non-presentation document domain is necessary to allow less restrictive methods to present non-presentation documents.

Further extensions outside the domain of document presentation can be achieved by introducing multiple semantics associations. XML language semantics do not necessarily span multiple processing domains. For instance, consider the case study media

language L_{cd} and a document that uses its constructs. The presentation of such a document requires presentation semantics, such as the illustrated case study semantics, that describe the presentation layout of the media information. In contrast, its processing for storing the media information into a database would require a different set of semantics that define the construct data types and relationships. Consequently, the introduction of multiple semantics associations can enable generic processing. Nevertheless, not all processing domains have similar well defined constraints as the presentation document processing domain, and generic document processing might not be always feasible.

Finally, the expressed ideas can inspire the development of processing models for any distributed freely evolving system, such as the Web. Exhaustive predefined sets of representations and processes are not adequate for such systems, because they can lead to restricted functionality and custom incompatible extensions, since an information system's evolution cannot be always predicted. In contrast, predefined evolution mechanisms that allow the individual representations and processes to follow the progress of an information system are more adequate. For instance, XMLPipe processing is not based on predefined processing interfaces or predefined set of languages. On the contrary, it allows the integration of independently developed and continuously evolving languages, which can describe any Web presentation. If XMLPipe used a predefined set of languages or a predefined interoperation interface its generality would be restricted and it would be eventually rendered obsolete. Similar mechanisms can be applied to other information systems and allow them to freely evolve, under a minimum set of constraints, according to their usage by the corresponding user community.

11.4 Concluding remarks

A well defined and powerful method to process presentation XML documents is necessary, because most user initiated interaction with XML documents results in some form of presentation. An adequate processing model must cover the parsing, validation, transformation and presentation of documents that combine an open set of languages. Additionally, document transformation and presentation must be adequate for a variety of adaptation requirements. Existing processing approaches cover a wide spectrum of functionality, but they do not provide sufficiently generic and powerful processing models. In contrast, they only cover a subset of the required functionality, which is also not always adequate for the Web.

The lack of adequate XML presentation models might originate from the lack of well defined boundaries for the presentation processing domain. Additionally, most approaches address the separate processing sub-problems individually. We hypothesised that generic document presentation can be achieved by utilising the constraints of the presentation processing domain and addressing the document processing problem as a whole.

We supported our hypothesis by using XMLPipe to cover the preprocessing domain and extrapolating the preprocessing observations to the generic document processing domain. Specifically, we firstly developed a preprocessing framework that established the necessary functionality of a preprocessing model that is adequate for the Web. By applying the principles expressed in our hypothesis, we developed the XMLPipe preprocessing model that enabled the validation and transformation of mixed namespace presentation documents that combine an open set of languages. The evaluation of XMLPipe using the preprocessing framework and case study established both its

adequacy and feasibility. The discussion of XMLPipe supported our hypothesis in the preprocessing domain by establishing that XMLPipe's adequacy is a direct consequence of following the two principles expressed in our hypothesis. Finally, in order to confirm the proposed hypothesis, the XMLPipe observations where extrapolated to cover the complete spectrum of presentation document processing.

The confirmed hypothesis combined with the XMLPipe architecture enable the generic processing of presentation documents and can be extended to cover additional processing domains. The support of our hypothesis inspires the development of generic processing models for presentation documents by utilising the constraints of the presentation domain and addressing the problem as a whole. The same paradigm can be used for the processing of non-presentation documents and for any other information in a freely evolving information system. Such information systems increasingly become the main means of information representation and retrieval. Adequate processing methods must not restrict them, but evolve with them. This thesis has illustrated that such an approach is feasible for the XML presentation domain and can inspire similar processing methods for all freely evolving systems.

Bibliography

[AAB+01] Murray Altheim, Murray Altheim, Frank Boumphrey, Sam Dooley, Shane McCarron, Sebastian Schnitzenbaumer, and Ted Wugofski. *Modularization of XHTML*. W3C, April 2001. Available in the Web: http://www.w3.org/TR/xhtml-modularization/.

[ABC+01a] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. *Extensible Stylesheet Language (XSL) Version 1.0, W3C Recommendation*. W3C, October 2001. Available in the Web: http://www.w3.org/TR/xsl/.

[ABC+01b] Jeff Ayars, Dick Bulterman, Aaron Cohen, Ken Day, Erik Hodge, Philipp Hoschka, Eric Hyche, Muriel Jourdan, Michelle Kim, Kenichi Kubota, Rob Lanphier, Nabil Layaida, Thierry Michel, Debbie Newman, Jacco van Ossenbruggen, Lloyd Rutledge, Bridie Saccocio, Patrick Schmitz, Warner ten Kate, and Thierry Michel. *Synchronized Multimedia Integration Language (SMIL 2.0), W3C Recommendation*. W3C, August 2001. Available in the Web: http://www.w3.org/TR/smil20/.

[All04] Sarah Allen. The Future of the Web is not the Past of Windows, Laszlo Systems. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[Bad98] Greg J. Badros. *Constraints in Interactive Graphical Applications, Ph.D. General Examination paper*. University of Washington, December 1998. Available in the Web: http://www.cs.washington.edu/homes/gjb/papers/constraints-iga.pdf.

[BB98] Greg J. Bardos and Alan Borning. *The Cassowary Linear Arithmetic Constraint Solving Algorithm*. Department of Computer Science and Engineering, University of Washington, June 1998. Available in the Web: http://bauhaus.cs.washington.edu/homes/gjb/papers/cassowary-tr.pdf.

[BB02] Jonathan Borden and Tim Bray. *Resource Directory Description Language (RDDL)*. W3C, February 2002. Available in the Web: http://www.rddl.org/.

[BBMS99] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint Cascading Style Sheets for the Web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, pages 73–82, November 1999.

[BCHL04] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. *Cascading Style Sheets, level 2 revision 1 CSS 2.1, W3C Candidate Recommendation.* W3C, February 2004. Available in the Web: http://www.w3.org/TR/CSS21/.

[BHL99] Tim Bray, Dave Hollander, and Andrew Layman. *Namespaces in XML.* W3C, January 1999. Available in the Web: http://www.w3.org/TR/REC-xml-names.

[Bir04] Mark Birbeck. A Standards-based Virtual Machine. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[BL98a] Tim Berners-Lee. *Realising the Full Potential of the Web.* W3C, 1998. Available in the Web: http://www.w3.org/1998/02/Potential.html.

[BL98b] Tim Berners-Lee. *Why RDF model is different from the XML model (W3C Design Issues notes).* W3C, October 1998. Available in the Web: http://www.w3.org/DesignIssues/RDF-XML.

[BL00] Tim Berners-Lee. *Weaving The Web.* TEXERE Publishing Limited, London, 2000.

[BL02a] Tim Berners-Lee. *Axioms of Web Architecture: the meaning of a document (W3C Design Issues notes).* W3C, December 2002. Available in the Web: http://www.w3.org/DesignIssues/Meaning.html.

[BL02b] Tim Berners-Lee. *Principles of Design (W3C Design Issues notes).* W3C, January 2002. Available in the Web: http://www.w3.org/DesignIssues/Principles.html.

[BL02c] Tim Berners-Lee. *The Interpretation of XML documents (W3C Design Issues notes).* W3C, March 2002. Available in the Web: http://www.w3.org/DesignIssues/XML.

[BL02d] Tim Berners-Lee. *Web Architecture from 50,000 feet (W3C Design Issues notes).* W3C, February 2002. Available in the Web: http://www.w3.org/DesignIssues/Architecture.html.

[BL03] Tim Berners-Lee. *The Stack of Specifications (W3C Design Issues notes).* W3C, June 2003. Available in the Web: http://www.w3.org/DesignIssues/Stack.

[BLFIM98] Tim Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax.* The Internet Engineering Task Force, Network Working Group, Standards Track, August 1998. Available in the Web: http://www.ietf.org/rfc/rfc2396.txt.

[BLM00] Alan Borning, Richard Kuang-Hsu Lin, and Kim Marriott. Constraint-based document layout for the Web. *Multimedia Systems, Springer-Verlag New York, Inc.*, pages 177–189, 2000.

[BM01] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes, W3C Recommendation*. W3C, May 2001. Available in the Web: http://www.w3.org/TR/xmlschema-2/.

[BM04] Dave Beckett and Brian McBride. *RDF/XML Syntax Specification (Revised), W3C Recommendation*. W3C, February 2004. Available in the Web: http://www.w3.org/TR/rdf-syntax-grammar/.

[BML+04] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient Incremental Validation of XML Documents. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society*, pages 671–682, 2004.

[BMSX97] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'97), ACM, New York NY USA*, pages 87–96, 1997.

[Bos04] Bert Bos. Setting the scope for light-weight Web-based applications. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C, October 2000. Available in the Web: http://www.w3.org/TR/REC-xml.

[BTM+01] Greg J. Badros, Jojada J. Tirtowidjojo, Kim Marriott, Bernd Meyer, Will Portnoy, and Alan Borning. Constraint extension to scalable vector graphics. In *Tenth International World Wide Web Conference, Hong Kong*, pages 489–498, May 2001.

[CBN+03] Petr Cimprich, Oliver Becker, Christian Nentwich, Honza Jirou, Michael Kay, Paul Brown, Manos Batsis, Tom Kaiser, Pavel Hlavnicka, Niko Matsakis, Cyrus Dolph, and Norman Wiechmann. *Streaming Transformations for XML (STX) Version 1.0*. Published in the Web, May 2003. Available in the Web: http://stx.sourceforge.net/documents/spec-stx-20030505.html.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C, November 1999. Available in the Web: http://www.w3.org/TR/xpath.

[CIMP03] David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. *Mathematical Markup Language (MathML) Version 2.0, W3C Recommendation*. W3C, October 2003. Available in the Web: http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[Cla99a] James Clark. *Associating Style Sheets with XML documents*. W3C, June 1999. Available in the Web: http://www.w3.org/TR/xml-stylesheet.

[Cla99b] James Clark. *XSL Transformations (XSLT) Version 1.0*. W3C, November 1999. Available in the Web: http://www.w3.org/TR/xslt.

[Cla03] James Clark. *Namespace Routing Language*. Thai Open Source Software Center Ltd., June 2003. Available in the Web: http://www.thaiopensource.com/relaxng/nrl.html.

[CLNL03] David W. Cheung, Eric Lo, Chi-Yuen Ng, and Thomas Lee. Web Services Oriented Data Processing and Integration. In *the twelfth international World Wide Web conference (WWW2003)*, pages 252–261, May 2003.

[CM01] James Clark and Makoto Murata. *RELAX NG Specification*. OASIS, December 2001. Available in the Web: http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[CMZ03] Yu Chen, Wei-Ying Ma, and Hong-Jiang Zhang. Detecting Web Page Structure for Adaptive Viewing on Small Form Factor Devices. In *the twelfth international World Wide Web conference (WWW2003)*, pages 225–233, May 2003.

[Cov98] Robin Cover. *XML and Semantic Transparency, XML Cover pages, Technology Report*. OASIS, November 1998. Available in the Web: http://www.oasis-open.org/cover/xmlAndSemantics.html.

[DKMR03] Micah Dubinko, Leigh L. Klotz, Roland Merrick, and T. V. Raman. *XForms 1.0, W3C Recommendation*. W3C, October 2003. Available in the Web: http://www.w3.org/TR/2003/REC-xforms-20031014/.

[DMO01] Steve DeRose, Eve Maler, and David Orchard. *XML Linking Language (XLink) Version 1.0*. W3C, June 2001. Available in the Web: http://www.w3.org/TR/xlink/.

[Dub04] Micah Dubinko. Position Paper on Compound Documents. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[ECM99] ECMA. *Standard ECMA-262: ECMAScript Language Specification, Third Edition*. ECMA International, December 1999. Available in the Web: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[ET01] Jerome Euzenat and Laurent Tardif. XML transformation flow processing. *Markup Languages: Theory and Practice*, 3:285–311, December 2001.

[Fal01] David C. Fallside. *XML Schema Part 0: Primer, W3C Recommendation*. W3C, May 2001. Available in the Web: http://www.w3.org/TR/xmlschema-0/.

[FGK02] Daniela Florescu, Andreas Grunhagen, and Donald Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proceedings of the eleventh International World Wide Web Conference, Hawaii, USA*, pages 65–75, May 2002.

[FIG+99] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.* The Internet Engineering Task Force, Network Working Group, Standards Track, June 1999. Available in the Web: http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[FJJ03] Jon Ferraiolo, FUJISAWA Jun, and Dean Jackson. *Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation.* W3C, January 2003. Available in the Web: http://www.w3.org/TR/SVG11/.

[FKS01] Wenfei Fan, Gabriel M. Kuper, and Jerome Simeon. A Unified Constraint Model for XML. In *Proceedings of the tenth international conference on World Wide Web*, pages 179–190, May 2001.

[Fou04] Mozilla Foundation. *Position Paper for the W3C Workshop on Web Applications and Compound Documents.* Mozilla Foundation and Opera Software, June 2004. Available in the Web: http://www.w3.org/2004/04/webapps-cdf-ws/papers/opera.html.

[GFMS03] Roger Gimson, Shlomit Ritz Finkelstein, Stephane Maes, and Lalitha Suryanarayana. *Device Independence Principles, W3C Note.* W3C, September 2003. Available in the Web: http://www.w3.org/TR/2003/NOTE-di-princ-20030901/.

[GHHW01] Ben Goodger, Ian Hickson, David Hyatt, and Chris Waterson. *XML User Interface Language (XUL) 1.0.* Mozilla Organization, 2001. Available in the Web: http://www.mozilla.org/projects/xul/xul.html.

[GNSP94] Yechezkal-Shimon Gutfreund, John Nicol, Russel Sansett, and Vincent Phuah. WWWInda: An Orchestration Service for WWW Browsers and Accessories. In *proceedings of the second international WWW Conference*, December 1994.

[Har04] Vincent Hardy. Web Applications and Compound Documents. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[Heg01] Philippe Le Hegaret. *The XML Processing model.* Postion paper for the XML processing model workshop, June 2001. Available in the Web: http://www.w3.org/2001/06/ProcessingModel-plh.html.

[HHW+04] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 3 Core Specification Version 1.0, W3C Proposed Recommendation.* W3C, February 2004. Available in the Web: http://www.w3.org/TR/DOM-Level-3-Core.

[HM01] Mary Holstege and R. Alexander Milowski. *Issues related to chaining XML Processes.* Postion paper for the XML processing model workshop, July 2001.

[HMM02] Nathan Hurst, Kim Marriott, and Peter Moulder. Dynamic Approximation of Complex Graphical Constraints by Linear Constraints. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 191–200, October 2002.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions in Internet Technology*, 2:117–148, 2003.

[Hya01] David Hyatt. *XBL - XML Binding Language (W3C NOTE).* W3C, February 2001. Available in the Web: http://www.w3.org/TR/2001/NOTE-xbl-20010223/.

[ISO86] ISO. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, August 1986.

[ISO04] ISO. *Document Schema Definition Languages (DSDL) project under ISO/IEC JTC 1/SC 34 WG 1.* ISO, May 2004. Available in the Web: http://dsdl.org/0525.pdf.

[J+03] Ian Jacobs et al. *Architecture of the World Wide Web (Working Draft).* W3C, October 2003. Available in the Web: http://www.w3.org/TR/webarch.

[Jel03] Rick Jelliffe. *The Schematron: An XML Structure Validation Language using Patterns in Trees.* Academia Sinica Computing Centre, 2003. Available in the Web: http://www.ascc.net/xml/resource/schematron/schematron.html.

[JLS+03] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive grid-based document layout. *ACM Transactions on Graphics, ACM Press*, 3:838–847, 2003.

[JW02] Ian Jacobs and Normal Walsh. *Architectural Principles of the World Wide Web, Working Draft.* W3C, August 2002. Available in the Web: http://www.w3.org/TR/2002/WD-webarch-20020830/.

[KCM04] Graham Klyne, Jeremy J. Carroll, and Brian McBride. *Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation.* W3C, February 2004. Available in the Web: http://www.w3.org/TR/rdf-concepts/.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 1:7–105, August 1998.

[KK03] Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation Language for XML. In *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA*, pages 256–272, January 2003.

[KL02] Martin Kempa and Volker Linnemann. On XML Objects. In *Plan-X: Programming Language Technologies for XML worskhop*, October 2002.

[KMS00] N. Klarlund, A. Moller, and M. I. Schwatzbach. DSD: A Schema Language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, pages 39–47, 2000.

[KSR02] Bintou Kane, Hong Su, and Elke A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proceedings of the fourth international workshop on Web information and data management*, pages 1–8, November 2002.

[KST03] Peter King, Patrick Schmitz, and Simon Thompson. Behavioural Reactivity and Real Time Programming in XML. In *Proceedings of the twelfth International World Wide Web Conference, Budapest*, May 2003.

[LC00] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 3:76–87, 2000.

[LFCH02] Simon Lok, Steven K. Feiner, William M. Chiong, and Yoav Hirsch. A Graphical User Interface Toolkit Approach to Thin-Client Computing. In *proceedings of the eleventh International World Wide Web Conference, Hawaii, USA*, pages 718–725, May 2002.

[LKSW04] Quanzhong Li, Michelle Y Kim, Edward So, and Steve Wood. XVM: A Bridge between XML Data and Its Behavior. In *Proceedings of the thirteenth International World Wide Web Conference, New York*, pages 155–163, May 2004.

[LMS99] R Lin, K Marriott, and P Stuckey. Flexible Font-Size Specification in Web Documents. In *Proceedings of the 22 Australasian Computer Science Conference, Auckland, New Zealand*, January 1999.

[Mar01] Jonathan Marsh. *XML Base*. W3C, June 2001. Available in the Web: http://www.w3.org/TR/xmlbase/.

[Mas02] Ishikawa Masayasu. *An XHTML + MathML + SVG Profile (working draft)*. W3C, August 2002. Available in the Web: http://www.w3.org/TR/REC-xml-names.

[Maz02] Stefano Mazzocchi. *Introducing Cocoon 2.0*. XML.com, February 2002. Available in the Web: http://www.xml.com/pub/a/2002/02/13/cocoon2.html.

[McG01] Sean McGrath. *Draft Essay on the XPipe Approach*. Propylon, December 2001. Available in the Web: http://xpipe.sourceforge.net/Articles/Documentation/fog0000000013.html.

[MCV04] Paolo Marinelli, Claudio S. Coen, and Fabio Vitali. SchemaPath, a Minimal Extension to XML Schema for conditional constraints. In *Proceedings of the thirteenth International World Wide Web Conference, New York*, pages 164–174, May 2004.

[MLM01] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages, Montreal, Canada*, 2001.

[MMM04] Cameron McCormack, Kim Marriott, and Bernd Meyer. Common rendering framework for compound Web documents. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

[MMSB01] Kim Marriott, Peter Moulder, Peter J. Stuckey, and Alan Borning. Solving Disjunctive Constraints for Interactive Graphical Applications. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 361–376, 2001.

[MO03] Jonathan Marsh and David Orchard. *XML Inclusions (XInclude) Version 1.0 (WD)*. W3C, November 2003. Available in the Web: http://www.w3.org/TR/xinclude/.

[Mol03] Anders Moller. *Document Structure Description 2.0*. BRICS, 2003. Available in the Web: http://www.brics.dk/DSD/dsd2.html.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML Transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22, 2000.

[NSL02] Markus L. Noga, Steffen Schott, and Welf Lowe. Lazy XML processing. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 88–94, 2002.

[OGC+01] Jacco van Ossenbruggen, Joost Geurts, Frank Cornelissen, Lynda Hardman, and Lloyd Rutledge. Towards Second and Third Generation Web-Based Multimedia. In *Tenth International World Wide Web Conference, Hong Kong*, pages 479–488, May 2001.

[OGHR03] Jacco van Ossenbruggen, Joost Geurts, Lynda Hardman, and Lloyd Rutledge. Towards a Multimedia Formatting Vocabulary. In *the twelfth international World Wide Web conference (WWW2003)*, pages 384–393, May 2003.

[OH02] Jacco van Ossenbruggen and Lynda Hardman. Smart Style on the Semantic Web. In *Semantic Web Workshop, the eleventh International World Wide Web Conference, Hawaii, USA*, May 2002.

[PAA+00] Steven Pemberton, Daniel Austin, Jonny Axelsson, Tantek Celik, Doug Dominiak, Herman Elenbaas, Beth Epperson, Masayasu Ishikawa, Shinichi Matsui, Shane McCarron, Ann Navarro, Subramanian Peruvemba, Rob Relyea, Sebastian Schnitzenbaumer, and Peter Stark. *XHTML 1.0 The Extensible HyperText Markup Language, Second edition, W3C Recommendation*. W3C, August 2000. Available in the Web: http://www.w3.org/TR/xhtml1/.

[PHV02] Kari Pihkala, Mikko Honkala, and Petri Vuorimaa. A browser framework for hybrid XML documents. In *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 164–169, August 2002.

[Por04] Portable Applications Standards Committee, IEEE Computer Society, USA. *Standard for information technology - portable operating system interface (POSIX). System interfaces 1003.1*, 2004. Available in the Web: http://www.unix.org/single_unix_specification/.

[PS03a] Michael Pediaditakis and David Shrimpton. Device neutral pipelined processing of XML documents". In *proceedings of the Twelfth International World Wide Web Conference*, page 2, 2003.

[PS03b] Michael Pediaditakis and David Shrimpton. Device-neutral pipelined processing of XML documents. *interChange*, 9(4):33–36, 2003.

[PS04] Michael Pediaditakis and David Shrimpton. Towards a generic XML content presentation model. In *the W3C Workshop on Web Applications and Compound Documents, San Jose, California, USA*, June 2004.

[PZB02] Thomas Phan, George Zorpas, and Rajive Bagrodia. An Extensible and Scalable Content Adaptation Pipeline Architecture to Support Heterogeneous Clients. In *Proceedings of The 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, July 2002.

[RHDS99] Franklin Reynolds, Johan Hjelm, Spencer Dawkins, and Sandeep Singhal. *Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation*. W3C, July 1999. Available in the Web: http://www.w3.org/TR/NOTE-CCPP/.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification W3C Recommendation*. W3C, December 1999. Available in the Web: http://www.w3.org/TR/html4/.

[Sch02] Patrick Schmitz. Multimedia meets computer graphics in SMIL2.0: A Time Model for the Web. In *Proceedings of the eleventh International World Wide Web Conference, Hawaii, USA*, pages 45–53, May 2002.

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures, W3C Recommendation*. W3C, May 2001. Available in the Web: http://www.w3.org/TR/xmlschema-1/.

[THHH01] Norio Touyama, Yasuyuki Hirakawa, Takashi Hattori, and Tatsuya Hagino. XEBRA: The Design and Implementation of Integrated Programming Environment for XML Processing and Browsing. In *Poster Proceedings of the The Tenth International World Wide Web Conference, Hong Kong*, May 2001.

[Via01] Victor Vianu. A Web Odyssey: From Codd to XML. In *Symposium on Principles of Database Systems*, 2001.

[Vli03] Eric van der Vlist. *Examplotron*. Dyomedea, February 2003. Available in the Web: http://examplotron.org/.

[web04] The W3C workshop on web applications and compound documents, June 2004. W3C, San Jose, California, USA
URL: http://www.w3.org/2004/04/webapps-cdf-ws/
Accessed at June 2006.

[Wei94] Pei Y. Wei. *A Brief Overview of the VIOLA Engine, and its Applications.* O'Reilly and Associates, Inc., August 1994. Available in the Web: http://www.xcf.berkeley.edu/ wei/viola/violaIntro.html.

[WM02] Norman Walsh and Eve Maler. *XML Pipeline Definition Language Version 1.0.* W3C, February 2002. Available in the Web: http://www.w3.org/TR/xml-pipeline/.

[WMF01] Norman Walsh, Eve Maler, and Christopher Ferris. *Sun position paper.* XML processing model workshop, June 2001.

[YW03] Christopher C. Yang and Fu Lee Wang. Fractal Summarization for Mobile Devices to Access Documents on the Web. In *the twelfth international World Wide Web conference (WWW2003)*, pages 215–224, May 2003.

[Zil04] Stephen Zilles. Web Applications/Compound Document Workshop Position Paper. In *the W3C Workshop on Web Applications and Compound Documents*, June 2004.

PRESENTING MULTI-LANGUAGE XML DOCUMENTS: AN ADAPTIVE TRANSFORMATION AND VALIDATION APPROACH.

APPENDICES

a thesis submitted to
The University of Kent at Canterbury
in the subject of computer science
for the degree
of doctor of philosophy.

By
Michael Pediaditakis
September 2006

Appendix A

Abbreviations

CC/PP: Composite Capabilities/Preferences Profile

CCSS: Constraint CSS

COC: Content Oriented Construct (there is also the$COC()$ binding model function)

CSS: Cascading Style Sheets

DOM: Document Object Model

DOM-2/DOM-3: DOM Level 2 and DOM Level 3, respectively

DSDL Document Schema Definition Languages

DTD: Document Type Definition

FOC: Functionality Oriented Construct (there is also the $FOC()$ binding model function)

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

LPS: Local Propagation Solver

MathML: The mathematical markup language

NRL: Namespace Routing Language

NVDL: Namespace-based Validation Dispatching Language

RCC: Rendering Custom Content

RDDL: Resource Directory Description Language

RDF: Resource Description Framework

SGML: Standard Generalized Markup Language

SMC: Structure Modification Construct (there is also the $SMC()$ binding model function)

SMIL: Synchronized Multimedia Integration Language

SVG: Scalable Vector Graphics language

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

W3C: World Wide Web Consortium (`http://www.w3.org/`)

WBMP: Wireless Bitmap

WML: Wireless Markup Language

WWW: The World Wide Web

XBL: XML Binding Language

XHTML: eXtensible HTML

XLink: XML Linking Language

XML: eXtensible Markup Language

XSL: eXtensible Stylesheet Language

XSL-FO: XSL Formating Objects

XSL-T: XSL Transformations

XUL: XML User Interface language

XVM: XML Virtual Machine

Appendix B

Terminology

This thesis introduced a new XML processing paradigm that combines several existing concepts and introduces a multitude of new terms. This thesis has defined all introduced concepts and terminology. Moreover, it has provided unambiguous interpretations of the necessary Web and XML terms, because the accustomed Web and XML processing terminology is not always well defined. This terminology reference appendix assists the reader by summarising all the terms used throughout this thesis.

Adaptation attribute is the CC/PP term for each URI identified resource that corresponds to an adaptation requirement. Within this thesis, we will use the term *adaptation term* instead, in order to avoid any ambiguity with the *XML attributes* term.

Adaptation component is the component that implements the XMLPipe adaptation model.

Adaptation expression is an expression that maps a tuple of term values to a value. Adaptation expressions are the basis of the XMLPipe adaptation requirements query mechanism. The applicability, adequacy and conflict resolution expressions are purpose specific instances of the abstract adaptation expressions. Each of them allows the necessary computations for their respective application domain. The set $\mathcal{F}$ contains all adaptation expressions. Specifically, $\mathcal{F}^{Type'}_{\{Type_1,\dots Type_n\}}$ is the set of all expressions that map an n-tuple of values, which correspond to the $Type_1, \dots Type_n$ term types, to a value of type $Type'$. Additionally, the convenience set $\mathcal{F}^{Type_2}_{(Type_1)^n}$ represents all expressions that map n values of a type $Type_1$ to a value of a type $Type_2$.

Adaptation expressions processing is the evaluation of one or more adaptation expressions, in order to either resolve a profile composition conflict or to evaluate the adequacy of a specification.

Adaptation factors are the document author, the document user and the target device. An XML presentation model must adapt a presentation according to the requirements of the adaptation factors, in order to be adequate for the target device, fulfill the document user presentation requirements and conveys the document author's message.

Adaptation measure is the combined applicability and adequacy measure of a transformation specification, and it is a comparable measure of how adequate is a

specification for an adaptation profile. Over a set of alternative specifications, XMLPipe chooses the one that has the maximum non-zero adaptation measure.

Adaptation measure function is function *measure* which maps a pair of an adaptation profile and a binding adaptation specification to the corresponding adequacy measure. If there is an applicability expression, the adaptation measure is the product of the applicability measure and the adequacy measure. If there is no applicability expression, the returned measure equals the adequacy measure, if all adequacy expressions evaluate to non-zero or to true. Otherwise, the resulting measure is 0.

Adaptation Model is the model that defines a representation of the adaptation requirements and a mechanism to use them for adapting a document. The XMLPipe adaptation model defines the adaptation profiles, which are a representation of the adaptation requirements. It also defines the necessary processing for composing adaptation profiles and for selecting the most adequate semantics definition over a set of independently developed alternatives.

Adaptation profile is a set of adaptation statements. The XMLPipe adaptation profiles consist of sets of XMLPipe adaptation statements. *Profiles* is the set of all XMLPipe adaptation profiles.

Adaptation profile composition is the process of composing multiple, possibly independently developed, adaptation profiles. The profile composition process must resolve any introduced conflicts. *ccompose* is the XMLPipe profile composition function, which resolves any introduced conflicts, using an ordering guideline and three distinct levels of conflict resolution specifications: the default conflict resolution, the term specific conflict resolution and the statement specific conflict resolution.

Adaptation Range is the freedom a component has for adapting its input document. Generally, the more abstract its input is, in relation to its output, the greater the adaptation range.

Adaptation requirements are the requirements of the adaptation factors.

Adaptation requirements representation is a representation for adaptation requirements, such as CC/PP. The XMLPipe adaptation profiles represent sets of adaptation requirements.

Adaptation statement is a statement that represents an individual adaptation requirement. The XMLPipe adaptation statements consist of an adaptation term and a value that corresponds to its type. The set *Statements* contain all XMLPipe adaptation statements.

Adaptation statement conflict resolution expression is the highest priority resolution expression and it overrides both the adaptation term specific conflict resolution expression and the XMLPipe default conflict resolution. It enables the authors of composite adaptation profiles to override the default conflict resolution mechanism, in order to customise the resulting adaptation profile.

Adaptation term is the equivalent of the CC/PP adaptation attributes: a URI qualified name that corresponds to an adaptation requirement. The set *Terms* is the set of all XMLPipe adaptation terms.

Adaptation term default value is the value that is used during expression evaluation, if an adaptation profile does not specify a value for the corresponding term. The default value of an adaptation term is specified in its semantics, and it is essential for using an open set of adaptation terms. Specifically, semantics authors can use applicability and adequacy expressions of a new term, even if a profile does not specify its value. The adaptation term default value will ensure a meaningful evaluation of the expression, until the profile authors introduce an adequate adaptation statement.

Adaptation term semantics is the necessary processing information for adaptation terms. *TermSemantics* is the set of all adaptation terms semantics and it consists of tuples of a default value, a data type and an optional conflict resolution expression.

Adaptation term semantics location function is the function *locateTermSem*. It is responsible for mapping a term to its semantics. The XMLPipe binding component implements *locateTermSem* by accessing the cached semantics entries for the term's namespace and returning the first matching definition.

Adaptation term specific conflict resolution expression is an optional part of the *adaptation term semantics*. The profile composer uses it to resolve two conflicting statements, if there is no statement specific conflict resolution expression.

Adaptation term type is the data type associated with a term, and it is specified by the term semantics. For instance, it can by a *Numeric* or *Boolean*.

Adequacy expression is a pair of a term and an unary adaptation expression that evaluates to either a *Numeric* or a *Boolean* value. According to the XMLPipe adaptation model, the evaluation of an adequacy expression provides a measure of the fulfilment of an adaptation requirement. The set *EAd* contains all XMLPipe adequacy expressions.

Adequacy expressions set is a set of adequacy expressions, and enables expressing the adequacy of a semantics specification for the various adaptation profiles. The adequacy measure function maps a pair of a set of adequacy expressions and an adaptation profile to a comparable adequacy measure.

Adequacy measure results from the evaluation of a set of adequacy expressions, for an adaptation profile. According to the XMLPipe adaptation model, the adequacy measure is an absolute measure of the adequacy of a semantics specification for an adaptation profile.

Adequacy measure function is the function *adequacy*, which calculates the adequacy measure of a set of adequacy expressions for an adaptation profile, by normalising and summing the evaluation of the individual adequacy expressions.

Alternative transformation algorithm : look at *revised transformation algorithm*

Applet is a pre-compiled Java program that enables rich document presentation by allowing full Java control over a rectangular document area. Applets, in a similar manner to plug-ins, introduce their own separate presentation model.

Applicability expression is an adaptation expression that maps multiple values to a boolean or numeric value, which must be in the $[0, 1]$ range. Applicability expressions are used by the applicability function to compute the applicability measure of a transformation specification, for an adaptation profile. EAp is the set of all applicability expressions.

Applicability function is the function *applicability* and it is responsible for mapping a pair of an applicability expression and an adaptation profile to the corresponding applicability measure.

Applicability measure is the measure of how applicable is a transformation specification alternative is for an adaptation profile. An applicability measure of 0 represents inadequacy, 1 represents full applicability and intermediate values intermediate applicability. The *applicability* function is responsible for calculating the applicability of an applicability expressions in EAp, according to an adaptation profile.

Application domain specific adaptation refers to methods that allow the adaptation of a document's presentation, which are specific to an individual application domain, such as the multimedia content domain.

Atomic transformation is a function that maps an adaptation profile to a transformation that can process a subset of a language's handled construct rooted subtrees. Atomic transformations are defined as functions, in order to incorporate the concept of a transformation, in a adaptation profile dependent way. Each XMLPipe transformation must implement the atomic transformation interface, which allows the seamless interoperation of a multitude of existing and future technologies. The set $\mathcal{A}$ contains all XMLPipe atomic transformations.

Atomic validation is process that validates a single namespace subtree, according to an optional external input. The common atomic validation interface allows the seamless integration of a multitude of validation technologies within the XMLPipe validation model. $\mathcal{A}$ represents the set of all atomic validations, and it contains all transformations that map a document subtree to its validated form.

Authoring validation is a part of the authoring process, because it provides document validity feedback to the document author. The XMLPipe authoring validation focuses on the semantic validity of a document, instead of its syntactic validity, because it is more adequate for evaluating if a processing model can successfully process a document. The XMLPipe authoring validation function is *validateAuth* that describes a two step transformation, which interoperates with both the transformation model and the processing validation interfaces. It firstly attempts to eliminate as many *SMC* rooted subtrees as possible, and it subsequently performs a deep subtree validation to the resulting document.

Authoring validation transformation is the authoring validation specific transformation algorithm that is responsible for preprocessing a document's *SMC* rooted

subtrees. *transformAuth* is the corresponding function. *transformAuth* is necessary, because there are no adequate alternative method to get the necessary *SMC* processing functionality form the generic *transformRev* function of the transformation model.

Autonomous validation is the usual validation process, where a document is validated against a schema specification. It contrasts with the *integrated validation processes*, which validate the processes that create or modify documents, instead of their output.

Binary adaptation expression is an adaptation expression with exactly two parameters. A typical example of binary adaptation expressions are the conflict resolution expressions.

Binding adaptation specification is the information that must be associated with the individual semantics specifications, in order to allow the selection of the optimal specification for an adaptation profile. $\mathcal{B}$ is the set of all XMLPipe binding adaptation specification. Each member of $\mathcal{B}$ is pair of an optional applicability expression and a set of adequacy expressions.

Binding component is the XMLPipe component that implements the proposed binding model. It is responsible for retrieving all necessary processing semantics, for processing a document.

Binding model is the XMLPipe sub-model that is responsible for bridging the other sub-models (adaptation, validation, transformation, integration) with the processing semantics. The proposed model uses a primary and a set of secondary location mechanisms. It orchestrates them using a local semantics cache, which contains a copy of all the necessary semantics, for the processing of a document. The binding model interface consists of the semantics location functions defined by the other preprocessing sub-models.

Browser refers to either an *XML browser* or a *Web browser*, according to its context.

Cache access is the XMLPipe binding model's component that is responsible for accessing the local semantics cache, in order to map a pair of a URI and a type of semantics to a set of semantics specifications. The cache access component implements the *locateSemantics* function.

Cache refresh is the XMLPipe binding model's component that is responsible for orchestrating both primary and secondary location mechanisms, in order to update the local semantics cache. The cache refresh process implements the *cacheImport* function, which updates all cache entries that corresponding to a URI and a type of processing semantics.

Cascading style sheets is a W3C recommendation that provides a syntax and a method to attach presentation style to structured documents. CSS is not an XML language, but it is related to XML, because document authors can use it to customise the presentation of XML documents.

CC/PP based stylesheet selection is the document adaptation method proposed in [OH02], which uses CC/PP profile queries to choose the optimal CSS or XSL-T

stylesheet for processing a document. It is the most prominent existing approach for generic document adaptation.

Cocoon is server based Web publishing approach that associates URIs and URI patterns to transformation pipelines.

Combination profiles Look at *integration profiles*.

Complete subtree validation is one of the two alternative XMLPipe subtree validation methods, and it tests the validity of all subtree nodes. The XMLPipe built-in pipelines include the predefined construct `validateSubtree`, which results to a complete subtree validation, prior to the subsequent pipeline processing steps.

Composite profile is the XMLPipe adaptation requirements representation that the preprocessing initiation entity provides. It consists of a *sequence* of composite adaptation statements, as opposed to the *adaptation profiles*, which are *sets* of statements. Composite profiles use statement sequences, because the XMLPipe entity-independent conflict resolution is based on an ordering guideline. *CProfiles* is the set of all composite adaptation profiles.

Composite adaptation statement is a pair of an adaptation statement and an optional conflict resolution expression. *CStatements* is the set of all composite adaptation statements.

Composite Capabilities/Preferences Profiles is a recommendation that specifies a representation for the adaptation factor requirements. It uses an RDF-based two level hierarchy of attribute-value pairs.

Conflict resolution is the process of resolving a conflict between two composite adaptation statements. *ccompose* is the function that performs all necessary conflict resolution, and it is a part of the XMLPipe adaptation model.

Conflict resolution expressions are binary adaptation expressions, which map two values of a data type into a single value of the same type. They are used in the semantics of the adaptation terms and in the composite profile statements, in order to define the resolution between two conflicting adaptation statements. The XMLPipe adaptation model replaces two conflicting statements with the result of evaluating a conflict resolution expression, over the conflicting values. The *ccompose* function, which is implemented by the profile composer component, is responsible for resolving all introduced conflicts.

Constraints refers to either numerical constraints or to the constraints of the presentation document processing domain, according on the context of its use.

Constraint CSS is a numerical constraints CSS extension that combines simultaneous linear constraints for positioning and one way constraints for font sizes. It allows significantly more generic presentation that the plain CSS, without significant performance costs.

Constraint problem is a conjunction of constraints of the form:

$$P = C_1 \wedge C2 \wedge \cdots \wedge C_m$$

Constraint problem solution A tuple $X = (x_1, x_2, \ldots, x_n)$ is a *solution* to a problem P iff X satisfies $C_i \forall i \in [1, m]$.

Constraint satisfaction A tuple $X = (x_1, x_2, \ldots, x_n)$ *satisfies* a constraint $C : f(X_1, X_2, \ldots, X_n)$ op c iff the expression $f(x_1, x_2, \ldots, x_n)$ op c is true. c is a constant and op is an operator where $op \in \{<, >, =, \leq, \geq\}$.

Constraint solver is a function $S : \mathcal{P}_{\mathcal{C}_n} \longrightarrow \wp(\mathbb{R}^n)$ that maps each problem in $\mathcal{P}_{\mathcal{C}_n}$ to a set of n-tuple solutions.

Content oriented constructs are the handled constructs that introduce a well defined piece of presentable information. According to the XMLPipe integration model, *COC* constructs can only occur at places where content is expected. The *COC* function maps a presentation language to its corresponding set of content oriented constructs.

CSS stylesheet selection is the CSS method of associating a document with multiple stylesheets, according to several device types, such as "mobile" and "desktop".

Declarative presentation specification/description is the use of declarative constructs to prepare and control the presentation. Such constructs define presentation relationships without defining how such relationships should be enforced. Examples include numerical constraints and the declarative binding in X-Forms.

Deep subtree validation look at *complete subtree validation*.

Default conflict resolution is the default XMLPipe mechanism for resolving adaptation statement conflicts in composite adaptation profiles. The default resolution always chooses the most recent value, and it is used when there are no term or statement conflict resolution expressions. The corresponding conflict resolution expression is $f(v_1, v_2) = v_2$.

Device independent authoring is the concept of authoring content independently of the several devices that it can be displayed. Device independent authoring approaches use purpose specific mechanisms to adapt the original content, according to a variety of devices and user preferences. Most approaches impose significant constraints on either the source syntax or the presentation component.

Document within the Web is a *resource description*. Within the context of this thesis and after Section 3.2, the term *document* refers to a *presentation document*.

Document author is entity that creates an XML document.

Document constructs are all the XML constructs of a document. The relationship between a document and its constructs is defined by functions $docConstructs()$, $docConstructs_e()$ and $docConstructs_a$. They map a document to the set of all its constructs, the set of all its element constructs and the set of all its attribute constructs, respectively.

Document languages is the set of languages used in a document uses. The relationship between a mixed namespace document d and its corresponding set of languages $\mathcal{L}_d$ is only well defined for namespace qualified document and namespace bound languages.

Document Object Model is the W3C recommendation that provides a generic and device independent interface for manipulating XML data.

Document presentation is the rendering of a document, in order to present it to a document user.

Document presentation processing is the necessary document processing for for document presentation. Document presentation processing, in addition to document presentation, also contains the document validation and transformation.

Document processing is the processing of a document by a processing model, which locates, combines and applies the processing model specific semantics that correspond to the document's languages.

Document Schema Definition Languages is an ISO work in progress that aims to generic XML validation. It is separated in several parts. The most relevant part to this thesis is the namespace based validation dispatching language (NVDL).

Document Type Declaration is the XML document declaration that can associate a document with its language and its syntax specifications. A document type declaration is introduced by the `DOCTYPE` XML construct.

Document Type Definition is the first XML schema language, and it is a part of the XML recommendation. It allows the specification of a language's elements and attributes and of their valid nesting.

Document User is the receiver of the information in an XML document. Typically, the document user has a set of preferences and uses a device and a browser to present an XML document, according to these preferences.

DOM parser is an XML parser that creates a DOM tree. XMLPipe uses DOM parsers, and the function *parse*() defines their common interface.

Domain constraints are the set of constraints, within an application domain, that can form the foundation of a generic processing model. The lack of such constraints results in an unbounded application domain and impedes proving the sufficiency of any related theory.

Driving example is the presentation document example introduced in Section 7.2 (page 115).

Dummy profile the adaptation profile used by the authoring validation process to drive the *SMC* subtrees elimination. Such a profile is necessary, because document transformation is adaptation requirements dependent, but authoring validation must be adaptation requirements independent.

ECMAScript is a widely used device neutral scripting language, and it is the standardised foundation of JavaScript.

Explicit identification method is a method of identifying the places where arbitrary *COC* constructs can occur. The explicit identification method requires that the individual schemas explicitly identify all such places. The XMLPipe subtree separation process introduces the predefined `coc` construct at all places where foreign namespace *COC* rooted subtrees occur. Schema specifications must identify

the places where content is expected by allowing occurrences of the predefined `coc` construct.

Extensible HTML is an XML representation of HTML.

Extensible Markup Language is an SGML-derived meta language and the cornerstone of the W3C's effort to establish a common data representation for the Web.

Functionality oriented constructs are the handled constructs that amend the presentation of their ancestors. According to the XMLPipe integration model, *FOC* constructs can occur at any place in a document. The *FOC* function maps a presentation language to its corresponding set of functionality oriented constructs.

Fundamental concepts of the Web consist of the protocols, concepts, addressing schemes and markup languages that where originally used for the majority of Web-based communication. Namely, the Hypertext Transfer Protocol (HTTP), the concept of links, the Uniform Resource Identifiers (URI) and the Hyper-Text Markup Language (HTML).

Generic adaptation refers to document adaptation methods that do not restrict either the document languages or the target presentation models. Generic adaptation methods are typically more generic but less powerful than application domain specific methods.

Generic document is a document which is not necessarily a presentation document.

Grammar–based schema is a schema which defines an XML language by defining the exact grammar to which documents must conform.

Grounded document is a document that contains exclusively constructs that have a pre-defined processing, within an application domain. In some cases, it is possible to define the meaning of non-grounded documents by combining constructs of grounded documents.

Handled constructs are a subset of presentation language constructs, where their processing can be defined independently of their context. $langConstructs^{HC}$ is a function that maps a language to its handled constructs.

Handled construct information access component is the binding model component that implements the integration model semantics access functions: *COC*, *SMC*, *FOC*, $langConstructs^{HC}$, $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$.

Handled construct subtree validation is one of the two XMLPipe subtree validation methods. It only test the validity of the top level constructs that belong to the same namespace as the subtree's root construct. Handled construct subtree validation assists the processing of semantically correct but invalid documents, because it allows the validation of only the constructs that are necessary for processing a subtree. The XMLPipe pipelines include the predefined construct `validateHC`, which introduces a handled construct subtree validation step prior to the subsequent pipeline processing steps.

Heuristic identification method is one of the methods to identify the places where arbitrary *COC* constructs can occur. According to this method the validation

model must attempt to infer the places where content is expected by the syntax of the language. The proposed heuristic method considers all places where *COC* construct of a language occur as places where any *COC* construct can occur. Such heuristic identification support complicates the design of the validation wrappers, because they must implement XMLPipe specific schema interpreters. Consequently, the XMLPipe validation model does not require such functionality, and it relies on the explicit identification method.

Hidden content is the individual subtrees that are associated to document elements by XBL. The hidden content is not part of the data part of the DOM tree, but it exists for controlling the presentation of the individual elements.

Hypertext Markup Language is the common representation for describing Web resources.

Hypertext Transfer Protocol is networking protocol used for the majority of Web communications.

Imperative presentation specification/description is the use of imperative programming techniques to prepare and control a presentation. Examples include JavaScript within XHTML documents, Java applets and plug-ins.

Integrated validation is the pre-runtime customisation of various processing components that ensures the validity of their output, against a predefined schema. This essentially consists process validation, as opposed to data validation.

Integration model is a model that defines the interpretation of a mixed namespace document according to the interpretation of its individual constructs.

Integration model semantics is the necessary language information for integration model specific processing. Within the context of this thesis, the term integration model semantics refers to the XMLPipe integration model semantics that consists of the handled constructs information. The set *ISemantics* contains all XMLPipe integration model semantics. Functions *COC*, *SMC*, *FOC*, $langConstructs^{HC}$, $langConstructs_e^{HC}$ and $langConstructs_a^{HC}$ map a language to its corresponding handled constructs.

Integration model transformation driver is the component that is responsible for combining the independent language transformation specifications to transform a mixed namespace document, which uses the corresponding integration model. The XMLPipe integration model transformation driver implements function *transformRev*, which uses independently developed transformation semantics to create the optimal interpretation of a document, according to a set of adaptation requirements.

Integration model validation driver is the component that is responsible for combining combining the individual language validation semantics to validate a mixed namespace document or document subtree. The XMLPipe integration model validation driver implements the function *validate*, which is responsible for separating a document in its single namespace subtrees and separately validating them.

Integration profile is a specification of the combined syntax and processing of a fixed set of languages. For instance, the XHTML+SVG+MathML W3C profile defines the syntax of valid documents that combine all three languages. Integration profiles are problematic, because their enumeration for an increasing number of XML languages becomes exponentially complex.

JavaScript See ECMAScript

Language author is the entity that creates an XML language.

Linear constraint is a numerical constraint of the form:

$$a_1X_1 + a_2X_2 + \cdots + a_nX_n \ op \ \ c$$

where $a_1, \ldots, a_n$ are constants, c is also a constant and op is an operator where $op \in \{<, >, =, \leq, \geq\}$.

Language constructs are all XML constructs defined by a language. Functions *langConstructs*(), $langConstructs_e()$ and $langConstructs_a$ define the relationship between a language and its constructs.

Language semantics is the intended usage and interpretation of a language by its author.

Language specific term is an adaptation term σ that shares the same namespace URI with a language L. XMLPipe uses language specific terms to allow the interoperation between separate instances of a language's transformers. Such interoperation is necessary for identifying and terminating infinite transformation loops, which can occur because of cyclic transformation dependencies.

Local propagation problem is a numerical constraint problem that can be solved by a local propagation solver.

Local propagation solver is a numerical constraint problem solver that only considers a single constraint at a time.

Mathematical markup language is an XML representation of both the structure and the content of mathematical notation.

Measure function See *adaptation measure function*.

Merger is transformation pipeline process that accepts multiple document inputs.

Minimal adaptation profile is an adaptation profile that, instead of specifying all supported languages, only specifies a languages subset that can co-exist in a single document. For instance, a browser that supports the XTHML+SVG+MathML profile and SMIL should have two separate minimal profiles, because they cannot be both used within a single document.

Mixed namespace document is an XML document that combines XML constructs from multiple namespaces.

Mixed namespace transformations are transformations that map a mixed namespace document into another mixed namespace document. More information is provided in the definition of *transformation*.

Multi-language document is a mixed namespace document. This term has been used the abstract and introduction chapter, as more intuitive, before the introduction of XML namespaces.

Namespace is a set of local names that can be used by the elements and attributes of an XML document. A namespace is identified by a URI, and it provides a separation between the constructs of different XML languages.

Namespace assimilation is the category of proposals, for the presentation of mixed namespace documents, where a language assimilates all related constructs of other languages within its namespace.

Namespace-based Validation Dispatching Language is a part of the Document Schema Definition Languages (DSDL) standard. It is very similar to the Namespace Routing Language, and it provides a method to associate namespaces with schema specifications and a model to validate mixed namespace documents. It is described in page 18.

Namespace qualified documents are all documents that only contain element constructs with an associated namespace URI.

Namespace bound languages are all languages that only introduced element constructs that have an associated namespace URI.

Namespace Routing Language is a syntax to associate namespaces with schemas and a model to validate mixed namespace documents. It is described in page 18 and in [Cla03].

Native presentation languages set is the set of languages that a a presentation component natively supports. It is represented by the set $\mathcal{L}_p$.

Node context is used by XMLPipe to keep track of the necessary node contextual information, for rich error reporting. Its main contents are information on the original node location and all applied transformations. XMLPipe also uses it to keep track of language specific adaptation statements and to resolve relative URL references.

Non-grounded document is a document that contains constructs that do not have predefined meaning, within an application domain. In some cases, it is possible to define the meaning of such constructs by combining constructs of grounded documents.

Numerical constraints refers to the sub-area of mathematical programming that focuses on finding tuples that simultaneously satisfy one or more numerical expressions.

One way problem denotes a sub-class of local propagation problems, where each constraint is an assignment (it only constraints the value of a single variable)

Optimal pipeline selection function is function *bestHCTS*, and it is used by the transformation driver to retrieve the optimal transformation pipeline, for each document subtree. *bestHCTS* maps a pair of a handled construct and an adaptation profile to the corresponding optimal pipeline specification. It uses the

transformation semantics location function *locateHCTS* to retrieve the necessary semantics and the adaptation model *measure* function to choose the most adequate one, according to an adaptation profile.

Parsing is the process that maps the textual representation of an XML document to a representation that is more accessible by applications, such as a DOM tree or a sequence of SAX events. The parsing of a document is successful if and only if the document is well formed. Parsing is well defined for all XML documents. Within XMLPipe, *parse* represents the interface of a DOM parser.

Parsing component is the part of an XML processing model that is responsible for parsing an XML document.

Plug-in is a commonly used extension mechanism that introduces a separate presentation model, in order to present a new language or media type.

Point of execution the point within a processing model that a piece of imperative code is executed. For instance, imperative code can be executed immediately after parsing the input document or during the document presentation.

Pre-presentation validation is the validation of a document prior to any presentation processing.

Preprocessing architecture is a fine grained description of a preprocessing model that can be directly mapped to an implementation.

Preprocessing model is an XML processing model that addresses all presentation document processing issues apart from its rendering: validation, transformation, adaptation, semantics binding, and languages integration.

Preprocessor is the implementation of a preprocessing architecture.

Presentation component is the minimal component of an XML browser that natively supports a finite set of XML languages and can present them to a document user by rendering their constructs.

Presentation documents are the namespace qualified documents that only contain constructs of presentation languages. D^P is the set of all presentation documents.

Presentation module is a part of a presentation model that is responsible for the presentation of a specific XML language. The X-Smiles browser presentation model is based on interoperating presentation modules.

Presentation model is the rendering sub-model of an XML processing model, which is responsible for presenting XML constructs to the user.

Presentation processing model is is used before Section 3.2 equivalently to the processing model.

Presentation languages are the XML languages that are associated with presentation semantics. $\mathcal{L}^P$ is the set of all presentation languages.

Presentation semantics are the language semantics that describe the primary interpretation of language constructs, according to their presentation. In contrast, non-presentation semantics *may be associated* with a presentation, but such presentation is not the primary interpretation of the corresponding language constructs.

Presentation validation is a validation process that occurs during the presentation of a document, and it ensures the validity of either inter-process communication or newly introduced data. Presentation validation examples are the validation of a user's input in a form and the validation of a piece of information, prior to its submission to a server.

Principal location mechanism is the core location mechanism of the XMLPipe binding model, and its primary focus is to locate the authoritative interpretation of resources (such as XML languages). It uses processing semantics RDDL links within the Web pages that corresponds to resource URIs.

Processing instruction is the XML construct for introducing inline document processing information. For instance, an `xml-stylsheet` processing instruction can be inserted at the beginning of a document to associate it with an XSLT stylesheet.

Processing model is a definition of how to locate, combine and apply processing semantics for processing a mixed namespace document, according to set of adaptation requirements (within the context of this thesis). $\mathcal{P}$ is the set of all processing models.

Processing model semantics is the processing model specific interpretation of language semantics. $\mathcal{I}_(P)$ represents the set of processing model semantics for a processing model P.

Processing semantics refers to the XMLPipe processing semantics, within the context of this thesis.

Processing validation is the process of validating a document prior to its processing, and it is usually initiated by a process and not a person. Within presentation processing models, presentation and pre-presentation validation are the two main subclasses of processing validation.

RDFXML is an XML representation of RDF.

RDF Schema is the W3C recommendation that allows the association of data types to RDF associations. RDF Schema does not include a collection of data types, but it can use other specifications, such as the XML Schema data types.

Relax NG is an alternative to W3C schemas that is based on an XML formal model. Relax NG uses element, attribute and text patterns to define XML grammars.

Rendering Custom Content is the part of the SVG 1.2 specification that, in a similar way to XBL, allows the presentations of unknown content by associating foreign namespace elements with SVG "shadow trees".

Resource is anything that can be identified by a URI, within the context of the Web.

Resource description is a piece of information that describes a resource. For instance, an HTML document is the description for the resource that is identified by the URI used to retrieve the document.

Resource Description Framework is the foundation of the Semantic Web. It uses labelled graphs, which use URIs for vertices, to represent any type of information.

Resource Directory Description Language is an XML syntax for incorporating machine processible resources links, within human readable XHTML descriptions. Each link has a "nature" and a "purpose". A link's *nature* specifies the type of the linked resource (e.g. a language schema). A link's *purpose* refines the usage of the linked resource.

Revised transformation algorithm is a revision of the initial XMLPipe transformation algorithm that widens the XMLPipe transformation model's applicability, by using a looser set of assumptions. *transformRev* is the function that implements the revised transformation algorithm.

Rule–based schema is a definition of an XML language, according to a set of rules and assertions that must hold for valid documents.

Scalable Vector Graphics is an XML representation of vector graphics, which covers their presentation, interaction and animation.

Schema is a syntax specification for a class of documents. The specification is represented using a schema language.

Schema binding is any form of explicit or implicit association between a document's constructs and their corresponding schemas.

Schema document See *schema*.

Schema integrator is a component that combines schemas of separate languages into a schema for mixed namespace documents that combine these languages.

Schema language is a language for specifying the syntax of a document.

Schema validator is a process that validates a document according to a schema specification.

Secondary location mechanisms is an open set of location mechanisms, which is combined with the principal location mechanism by XMLPipe, in order to retrieve the required processing semantics. Secondary location mechanisms enable independent development of semantics, avoid central points of failure and allow the easier adoption of XMLPipe. The XMLPipe binding model does not explicitly define the details of any secondary location mechanisms. Nevertheless, it defines the corresponding semantics organisation.

Selection pipeline is the XMLPipe pipeline that applies the optimal transformation, over a set of alternatives, according to the adaptation requirements. Each alternative is associated with a binding adaptation specification, which is evaluated in the same way as in the selection of the optimal transformation semantics. *sel* is the function that maps a sequence of pairs of transformations and binding adaptation specifications to a selection pipeline.

Semantically correct is a document with well defined interpretation, according to the semantics of its languages. A semantically correct document is not necessarily valid. The proposed XMLPipe authoring validation evaluates the semantic correctness of a document, because the proposed transformation model can process semantically correct but invalid documents. The proposed validation model evaluates a document's semantics correctness by a recursive elimination of the *SMC* rooted subtrees that converts most invalid semantically correct documents to valid documents.

Semantics has multiple interpretations depending on its context. Within this thesis semantics is equivalent to the *language semantics*. After Section 3.2, unless stated otherwise, semantics refers to the even more specific notion of *presentation semantics*.

Semantics cache is the local XMLPipe cache that caches all necessary processing semantics, for processing a document. The semantics cache is accessed by the *cache access* component, and it is updated by the *cache import process*.

Semantics location is the process of locating the necessary semantics specifications for processing a document.

Semantics organisation is the organisation of the processing semantics within a medium.

Sequence pipeline is the XMLPipe transformation pipeline that applies two transformations in sequence. The sequential composition of multiple transformations can be described as nested sequence pipelines. *seq* is the function that maps two transformations to the corresponding sequence pipeline.

Shadow tree is the RCC equivalent of the XBL hidden content.

Simultaneous problem is a numerical constraint problem that cannot be solved by an LPS, because it requires considering multiple constraints at a time.

Sink is a process that outputs a document after its processing, within the context of transformation pipelines.

Source is a process that retrieves a source document prior to its processing, within the context of transformation pipelines.

Statement See *adaptation statements*.

Standard Generalized Markup Language is a generic meta-language that is, however, overly complex for the Web. XML is a simpler alternative.

Structure modification constructs are the handled constructs where the principal purpose of their semantics is to introduce document modifications. According to the XMLPipe integration model, *SMC* constructs can occur at any place in a document. *SMC* constructs require separate treatment by the validation process, because the introduced document modifications must be taken into account. The *SMC* function maps a presentation language to its corresponding set of structure modification constructs.

Synchronized Multimedia Integration Language (SMIL) is an XML representation of interactive multimedia applications. SMIL consists of separate modules that can be used in other languages. For instance, the SVG animation is based on the SMIL timing component.

Target device is the device used by the document user to interacting with a document's presentation.

Term See *adaptation term*.

The Web The World Wide Web

Transformation is a mapping of a document to another document. Prior to the XMLPipe transformation model chapter, $\mathcal{T}_{L_1}^{L_2}$ represents all transformations that map the constructs of a language L_1 to the constructs of a language L_2, optionally according to an external input. The transformation model chapter used the introduced concept of integration models to allow more fine grained definition of mixed namespace transformations. $\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2}$ is the set of all transformations that process the constructs of languages in $\mathcal{L}_2$ for mapping an input document, which combines the constructs of the languages in $\mathcal{L}_1$ using the integration model Im_1, to an output document, which combines the constructs of the languages in $\mathcal{L}_3$, using the integration model Im_2.

Transformation driver is the component that controls the transformation process of a processing model. The transformation driver typically interoperates or includes several integration model transformation drivers, to which it delegates all transformation requests. The XMLPipe transformation driver is represented by function *XMLPipeTrans* and it is responsible for driving the XMLPipe transformation process by interoperating with the parser, the adaptation model and the integration model transformation driver.

Transformation model is the sub-model of an XML processing model that is responsible for transforming an XML document. Document transformation is useful for content selection, content customisation and mapping non natively supported language constructs to natively supported ones. Within XMLPipe, the transformation component implements the XMLPipe transformation model.

Transformation pipeline is a transformation that combines multiple simpler transformations. The XMLPipe built-in transformation pipelines are recursive compositions of atomic transformations. The supported composition methods are the *sequence pipeline*, the *selection pipeline* and the *dynamic pipeline*. Additionally, an XMLPipe pipeline can contain calls to either handled construct or deep subtree validation. *Pipelines* represents the set of all XMLPipe pipelines.

Transformation selection look at *selection pipeline*.

Transformation semantics is the necessary processing information for transforming XML documents. The XMLPipe transformation semantics consist of the necessary information for transforming valid document subtrees, which are rooted at handled constructs. The set *HCTSemantics* represents the set of all XMLPipe transformation semantics. Each member of *HCTSemantics* is a pair of a transformation pipeline specification and an optional binding adaptation specification.

Transformation semantics access component is the binding model component that implements the transformation semantics location function *locateTermSem*.

Transformation semantics location function is *locateHCTS*, which is the binding model interface for locating the transformation semantics for a handled construct. *locateHCTS* maps a qualified handled construct to its corresponding set of transformation semantics.

Transformation sequence look at *sequence pipeline*

Transformation specification design guidelines are necessary guidelines for designing transformation semantics that are adequate for the XMLPipe subtree-based processing.

Transformer is a process that performs a transformation. Within the context of the transformation pipelines, it is the only pipeline process that transforms the a document. Within the context of the XMLPipe pipelines, it is represented by the atomic transformations.

User preferences is a set of statements that express a user's document presentation preferences.

Uniform Resource Identifier is a uniform representation that can identify any Web resource.

Uniform Resource Locator is a URI that identifies a resource, via a representation of its primary access mechanism (e.g. the network location).

Valid document is an XML document that is both well formed and consistent with a syntax specification. An integration model allows the validation of mixed namespace documents, according to schemas of their individual languages.

Validation is the process that tests the validity of a document, against a set of rules (typically described by a schema).

Validation assumptions are the necessary assumptions for establishing the equivalence of the proposed validation algorithm to the XMLPipe integration model definition of valid documents.

Validation design guidelines are the necessary schema design guidelines for ensuring the adequacy of schemas for the XMLPipe subtree validation model. An XMLPipe atomic transformation must adhere to these guidelines. Generic validation technology wrappers, which implement the XMLPipe atomic validation interface, must map the top level design guidelines to the corresponding technology specific ones.

Validation driver is that drivers the document validation process. It usually delegates the validation requests to the corresponding integration model validation driver. The XMLPipe validation driver implements function *XMLPipeVal*, which interoperates with the document parser and the authoring validation driver, in order to parse the input document and to perform its authoring validation, respectively.

Validation model is the part of an XML processing model that is responsible for the validation of XML documents.

Validation semantics is the necessary information for validating the usage of a language's constructs. The XMLPipe validation model semantics consist of a schema specification and a reference to the an atomic validation implementation. The set *VSemantics* contains all XMLPipe validation semantics. Function *locateVS* represents the corresponding part of the binding component interface that maps a language URI to a set of one or more validation semantics specifications.

Transformation semantics access component is the binding model component that implements the validation semantics location function *locateVS*.

Validation semantics location function is function *locateVS*, which maps a language URI to a set of one or more validation semantics specifications. The XMLPipe binding component implementation of *locateVS* combines all available specifications that are associated with a language's namespace.

Web browser is an application that allows users to interact with the presentation of all the - World Wide Web information.

Web design principles are the core design principles of the Web:

- Simplicity
- Modular Design
- Tolerance
- Decentralisation
- Test of independent invention
- Principle of least power

They have been described in Section 1.3.

Well formed document is an XML document that is consistent with the core XML syntax rules. The XML recommendation only requires that a document is well formed.

X-Smiles is XML browser that focuses on supporting several XML languages. It uses a simplistic integration model to enable their generic integration.

XForms is a device neutral XML language for the "online interaction of a person and another, usually remote, agent"[DKMR03]. XForms is an substantial extension of the HTML forms, which are a core component of interactive Web applications.

XML Binding Language is a representation for attaching processing behaviour to XML elements, in a similar manner to the way that CSS attaches style information. XBL uses device neutral scripting languages, such as ECMAScript[ECM99], to define an element's behaviour.

XML browser is an interactive application that provides a partial or a complete implementation of a processing model and of its user/machine interfaces. An XML browser should, at a minimum, be able to present documents that use a predefined set of languages.

XML constructs is the set of both qualified and unqualified XML elements and attributes.

XML document is a textual tree representation of data, which conforms with the XML syntax (well formed). After Section 3.2, unless stated otherwise, the term XML document is used interchangeably to the term *presentation document*.

XML language is a set of XML documents that is usually specified by a schema. After Section 3.2, unless stated otherwise, the term XML language is used interchangeably to the term *presentation language*.

XML linking language is a W3C recommendation that provides generic linking semantics, by defining a set of attributes that can be attached to any XML element.

XML schema is a W3C validation proposal that addresses most of the DTD shortcomings. XML Schemas support namespaces, data types and inheritance.

XML semantics is used interchangeably with the term *language semantics*

XML user interface language is a model and a language for building graphical user interfaces. The user interface of the Firefox browser is an XUL application.

XML Virtual Machine is a generic purpose XML processing approach that attaches behaviour to XML constructs and uses a distributed binding mechanism.

XMLPipe is the proposed preprocessing model which combines all proposed submodels: the integration model, the adaptation model, the transformation model, the validation model and the binding model. Their combination allows the generic preprocessing of presentation documents, in a way that is adequate for the Web.

XMLPipe valid documents are the presentation documents that are valid according to the XMLPipe integration model. They consist of either one or a valid combination of single namespace subtrees rooted at a handled construct. *FOC* and *SMC* subtrees can occur at any place, but *COC* rooted subtrees must only occur at places where content is expected.

XMLPipe validation driver See *validation driver*.

XMLPipe transformation driver See *transformation driver*.

XPath is an path-like syntax for addressing the constructs of an XML document. It is used in XSL to refer to source document constructs, define functional data computations and provide a library of predefined functions.

XSL Formating Objects is a vocabulary for document formating semantics, and it is mainly focused towards printed media.

XSL Transformations is a language that describes transformations of XML documents to other, XML or non-XML, documents. The principal application of XSL-T is to transform XML documents that use arbitrary XML languages to documents that only use natively supported languages, by a target browser.

Appendix C

Formalisms

This thesis has extensively used a formal notation to assist the precise communication of the XML processing concepts. The introduced notation adopted many concepts of functional analysis and set theory. Additionally, it introduced a multitude of new symbols and formal concepts, because most XML and Web concepts are underdefined and there is no formal model that covers all aspects of XML processing. This appendix aims to assist the reader by providing a reference to all used notation, organised according to the individual thematic areas.

C.1 Core notation

This section summarises the subset of set theory and function analysis notation used by the introduced definitions.

C.1.1 Symbol conventions

The introduced symbols can span a wide complexity spectrum and can be sets, members of that sets and sets of sets. The used notation uses small, capital and calligraphic letters to differentiate between the relative complexity of each concept. Table C.1.1 summarises the used conventions. Small letters represent relatively simple set members, such as local names (S) and documents (d). A capital letter, can either represent a set of such constructs or a composite set member. For instance, D represents the set of all documents. Additionally, L represents a language. Languages are represented by capital letters, because an XML language can be considered as a composite construct that is defined by the set of its valid documents. Caligraphic letters generally express sets of such composite constructs. $\mathcal{L}$ represents the set of all languages. The used notation is based on the relative (and possibly subjective) complexity of each construct, in order to stress the relationships between the introduced terms.

Convention	Example	Description
Small letters	d	simple set members
Capital letter	D, L	sets or composite set members
Calligraphic letters	$\mathcal{L}$	sets of composite entities

Table C.1: Symbol conventions

C.1.2 Sets notation

For all set members ($\forall$)

$\forall$ must can read as "for all members". For instance, $\forall d \in D$ can be read as "for all documents of the set of all documents".

There is a set member ($\exists$)

$\exists$ can be read as "There is a". For instance, $\exists d \in D$ can be read as "There is a document in the set of all documents.

Empty set ($\emptyset$)

$\emptyset$ represents a set that has no members: $\emptyset = \{\}$.

Null set member (ϵ)

ϵ represents a null set member that belongs to all sets and its addition to a set does not modify the set: $\forall$ sets$S, \epsilon \in S$, $\forall$ sets$S, \{\epsilon\} \cup S = S$.

Power set ($\wp$)

For a set A, $\wp(A)$ represents the power set (set of all subsets) of A. For instance, if $A = \{1, 2, 3\}$, then

$$\wp(A) = \{\emptyset, \{1\}\, \{2\}\, \{3\}\, \{1, 2\}\, \{1, 3\}\, \{2, 3\}\, \{1, 2, 3\}\}$$

C.1.3 Functions notation

Function definition ($f : A \to B$)

A function $f : A \to B$ is a function that maps values that belong to set A to values that belong to a set B. For instance, if $a \in A$, then $f(a) = b \in B$. Functions can have multiple input and output values. For instance $f : A \times B \to C \times D \times E$ maps pairs of values $(a, b) \in A \times B$ to triples of values $(c, d, e) \in C \times D \times E$.

The pipeline definitions combine set unions with the function definition notation. For instance,

$$f : \bigcup_{i \in [1,2]} A_i \to B_i$$

represents the union of functions $f : A_1 \to B_1$ and $f : A_2 \to B_2$.

C.2 Core XML notation

C.2.1 Documents

Documents are represented within this thesis using a small d: d, $d_1, d_2, \ldots, d_n$

XML documents (D) page 52

D represents the set of all well-formed XML documents.

XML constructs (Σ) page 52

The set of all *XML constructs* Σ includes all qualified or unqualified names of XML elements and attributes:

$$\Sigma = (URI \cup \epsilon) \times S$$

where URI is the set of URIs, S is the set of all non-qualified XML names and ϵ is an null URI

Document constructs functions ($docConstructs$) page 52

$docConstructs : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs(d)$ is the set of all the XML constructs in d.
$docConstructs_e : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs_e(d)$ is the set of all the XML constructs that appear as elements in d.
$docConstructs_a : D \rightarrow \wp(\Sigma)$ is a function where, $\forall d \in D, docConstructs_a(d)$ is the set of all the XML constructs that appear as attributes in d.

Namespace qualified XML documents (D_Q) page 53

The set of all *namespace qualified XML documents* $D_Q \subset D$ is the subset of XML documents where $\forall d \in D_Q, \forall \sigma = (uri, s) \in docConstructs_e(d), uri \neq \epsilon$

Presentation documents (D^P) page 54

The presentation documents subset of namespace qualified XML documents is the set $D^P \subset D_Q$, where $\forall d \in D^P$ the set of document languages $\mathcal{L}_d$ contains only presentation languages: $\mathcal{L}_d \subset \mathcal{L}^P$.

C.2.2 XML languages

Languages are represented within this thesis using a capital L: L, $L_1, L_2, \ldots, L_n$

XML languages ($\mathcal{L}$) page 52

$\mathcal{L}$ represents the set of all XML languages

Language constructs functions ($langConstructs$) page 53

$langConstructs : \mathcal{L} \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs(L)$ is the set of all the XML constructs that are defined by L.
$langConstructs_e : D \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs_e(L)$ is the set of all the XML constructs that are defined as elements by L.
$langConstructs_a : D \rightarrow \wp(\Sigma)$ is a function where, $\forall L \in \mathcal{L}, langConstructs_a(L)$ is the set of all the XML constructs that are defined as attributes language L.

Namespace bound XML languages ($\mathcal{L}_Q$) page 53

The set of all *namespace bound XML languages* $\mathcal{L}_Q \subset \mathcal{L}$ is the subset of XML languages where $\forall L \in \mathcal{L}_Q \ \forall \sigma = (uri, s) \in langConstructs_e(L), uri \neq \epsilon$

Presentation languages ($\mathcal{L}^P$) page 54

The set of all *presentation languages* is the subset of XML languages $\mathcal{L}^P \subset \mathcal{L}$, where each $L \in \mathcal{L}^P$ is associated with presentation semantics $I \in \mathcal{I}^P$.

C.2.3 Language sets

Document's languages ($\mathcal{L}_d$) page 53

For a namespace qualified document $d \in D_Q$, the set of its languages is the smallest subset of the XML bound languages $\mathcal{L}_d \subset \mathcal{L}_Q$, where
$\forall \sigma \in docConstructs_e(d), \exists L \in \mathcal{L}_d$ where $\sigma \in langConstructs_e(L)$
and
$\forall \sigma = (uri, s) \in docConstructs_a(d), uri \neq \epsilon, \exists L \in \mathcal{L}_d$ where $\sigma \in langConstructs_a(L)$.

Set of natively supported languages ($\mathcal{L}_p$)

The set of natively supported languages $\mathcal{L}_p$ represents the set of languages that a presentation component/rendering sub-model/target browser can represent.

C.3 Processing models

Processing models ($\mathcal{P}$) page 55

$\forall d \in D_Q$, where for all the languages in $\mathcal{L}_d = \{L_1, \ldots, L_n\}$ there are the respective processing model's P semantics $\{I_{P1}, \ldots, I_{Pn}\}$, P defines how to locate, combine and apply these semantics to interpret d, according to the processing model specific interpretation of the languages in $\mathcal{L}_d$. $\mathcal{P}$ denotes the set of all processing models.

C.4 Semantics

C.4.1 Core definitions

XML semantics ($\mathcal{I}$) page 54

For each XML language $L \in \mathcal{L}$, there is a single precisely defined semantics $I \in \mathcal{I}$, which represents the language intended usage and interpretation by its author. The set $\mathcal{I}$ is the set of all XML language semantics.

Presentation semantics ($\mathcal{I}^P$) page 54

The set of all *presentation semantics* is the subset of XML semantics $\mathcal{I}^P \subset \mathcal{I}$, that

define a language's interpretation according to the presentation of its constructs to the document user.

Processing model semantics ($\mathcal{I}_P$) page 55

$\forall I \in \mathcal{I}$, iff there is a semantics implementation I_P of the semantics I for the processing model P, then $I_P \in \mathcal{I}_P$

C.4.2 XMLPipe processing semantics

XMLPipe integration model semantics (*ISemantics*) page 87

ISemantics represents the set of all XMLPipe integration model semantics. Each member of *ISemantics* is a 5-tuple of sets of qualified names that correspond to the *COC*, *SMC*, *FOC*, element and attribute handled constructs of a language, respectively:

$$ISemantics = \wp(\Sigma)^5$$

If $is \in ISemantics$ and $is = (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5)$, then

$$\forall \sigma \in \Sigma_i : \begin{cases} \sigma \text{ is a } COC \text{ construct} & \text{, if } i = 1 \\ \sigma \text{ is a } SMC \text{ construct} & \text{, if } i = 2 \\ \sigma \text{ is a } FOC \text{ construct} & \text{, if } i = 3 \\ \sigma \text{ is an element construct} & \text{, if } i = 4 \\ \sigma \text{ is an attribute construct} & \text{, if } i = 5 \end{cases}$$

XMLPipe validation semantics (*VSemantics*) page 158

VSemantics contains all XMLPipe validation semantics. Each member of *VSemantics* is a pair of references to a schema specification and to the corresponding atomic validation implementation.

Transformation semantics (*HCTSemantics*) page 130

The set *HCTSemantics* contains all the transformation semantics, which consist of an optional binding adaptation specification and a pipeline specification:

$$HCTSemantics = ((\mathcal{B} \cup \{\epsilon\}) \times PipeSpec)$$

Term semantics (*TermSemantics*) page 95

The set of all term semantics *TermSemantics* contains all valid triplets of a term type, a default value and a conflict resolution expression:

$$TermSemantics = \bigcup_{\forall Type \in TermTypes} \left(\{ Type \} \times Type \times \left(\{\epsilon\} \cup \mathcal{F}^{Type}_{(Type)^2} \right) \right)$$

Term semantics functions (*termType*,*termDefault*,*termResolve*) page 96

For every term $\tau \in Terms$, where $locateTermSem(\tau) = (Type, v, f)$, the utility functions are defined as follows

$$\begin{array}{ll} termType : Terms \rightarrow TermTypes, & termType(\tau) = Type \\ termDefault : Terms \rightarrow \bigcup_{\forall Type \in TermTypes} (Type), & termDefault(\tau) = v \\ termResolve : Terms \rightarrow \mathcal{F}, & termResolve(\tau) = f \end{array}$$

Binding adaptation specification ($\mathcal{B}$) page 107

The set of all binding adaptation specifications $\mathcal{B}$ consists of pairs of an optional applicability expression and a set of adequacy expressions.

$$\mathcal{B} = (\{\epsilon\} \cup EAp) \times BA$$

C.4.3 XMLPipe semantics location functions

Handled construct information access functions page 179

$\forall uri \in URI$, where $locateSemantics(uri,$ `XMLPipeURI : intModelSemantics`$) =$
$= (\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5)$:
$langConstructs^{HC}(uri) = \Sigma_4 \cup \Sigma_5$,
$langConstructs_e^{HC}(uri) = \Sigma_4$,
$langConstructs_a^{HC}(uri) = \Sigma_5$,
$COC(uri) = \Sigma_1$, $SMC(uri) = \Sigma_2$, $FOC(uri) = \Sigma_3$

Adaptation term semantics location function page 179

$\forall uri \in URI, s \in S$,
$locateTermSem((uri, s)) = ts_{ij}$
iff $locateSemantics(uri,$ `XMLPipeURI : termSemantics`$) =$
$\{\{(s_{11}, ts_{11}), \ldots, (s_{1n_1}, ts_{1n_1})\} \ldots, \{(s_{k1}, ts_{k1}), \ldots, \{s_{kn_k}, ts_{kn_k}\}\}$ and $s_{ij} = s$.
Otherwise, $locateTermSem((uri, s)) = \epsilon$

Transformation semantics location function page 179

$\forall \sigma = (uri, s) \in \Sigma$, where
$locateSemantics(uri,$ `XMLPipeURI : transSemantics`$) =$
$\{QH_1, \ldots, QH_n\} \in \wp(\wp((\wp(\Sigma) \times HCTSemantics)))$,

$$locateHCTS(\sigma) = \bigcup_{i \in [1,n]} \{hcts : (\sigma, hcts) \in QH_i\}$$

Otherwise, if $locateSemantics(uri,$ `XMLPipeURI : transSemantics`$) = \emptyset$,
then $locateHCTS(\sigma) = \emptyset$

Validation semantics location function page 179

$\forall uri \in URI$,
if $locateSemantics(uri, \texttt{XMLPipeURI} : \texttt{valSemantics}) = \{VS_1, \ldots, VS_n\} \neq \emptyset$,

$$locateVS(uri) = \bigcup_{i \in [1,n]} VS_i$$

otherwise, $locateVS(uri) = \emptyset$

C.4.4 XMLPipe semantics binding

Semantics location function (locateSemantics) page 177

The *locateSemantics* function is defined by the following algorithm:

function *locateSemantics*(*URI uri*, *URI nature*) $\rightarrow$
$ISemantics \cup (\Sigma \times TermSemantics) \cup$
$\wp(\wp(VSemantics)) \cup \wp(\wp((\wp(\Sigma) \times HCTSemantics)))$
let *cache* be the semantics cache
if there is no $c \in cache$ that corresponds to *uri* and *nature*
cacheImport(*uri*, *nature*)
end if
let $ret = \emptyset$
for each $c \in cache$ that corresponds to *uri* and *nature*
Add the processing semantics described by c to *ret*
end for
return *ret*
end function

Cache import (cacheImport) page 178

The *cacheImport* function updates the semantics cache information, using both principal and secondary location mechanisms.

function *cacheImport*(*URI uri*, *URI nature*)
let *cache* be the semantics cache
let *alt* be the URI list of the alternative retrieval mechanisms
let $trust \in URI \times \mathbb{R}$ be the predefined URI trust associations
let *trustMin* be the minimum trust threshold
let *foundSemantics* = *false*
let $semList = \emptyset$ be an empty set of pairs of semantics and trust levels
//attempt to use the principal location mechanism
$sem = RDDLParse(uri, nature)$
if $sem \neq \emptyset$; add $(sem, 1)$ to *semList*; **end if**
//Use the alternative location mechanisms
for each $uri' \in alt$
if $\exists(uri', x) \in trust$ AND $x \geq trustMin$
$sem = altSem(uri', uri, nature)$

```
            if sem ≠ ∅
                if nature == intModelSemantics
                    if semList == ∅ OR ((x′, sem′) ∈ semList AND x > x′)
                        semList = ∅
                        add (x, sem) to semList
                    end if
                else if nature == termSemantics
                    add (x, sem′) to semList, where sem′ includes all term
                            semantics not declared in semList for higher priorities.
                    remove old duplicate term declarations from semList
                else if nature == valSemantics
                    add (x, sem′) to semList, where sem′ includes all atomic
                            validations, which do not have the same wrapper
                            implementation to a higher priority semList entry.
                    remove old duplicate (same wrapper implementation) atomic
                            validations from semList
                else if nature == transSemantics
                    add (x, sem) to semList
    end if; end if; end if; end for
    if semList ≠ ∅
        replace cache entry for uri and nature with semList
    end if
end function
```

C.5 Integration model

Integration models are represented within this thesis using the symbol Im: Im, Im_1, Im_2, ..., Im_n. Im_X represents the XMLPipe integration model and Im_P represents the presentation integration model of a presentation component/rendering submodel/target browser

Valid tree composition notation ($\overset{+}{\underset{Im}{\leftarrow}}$) page 85

For all $d_1, d_2 \in D_Q$, $d_1 \overset{+}{\underset{Im}{\leftarrow}} d_2$ is the set of all documents that can be produced by placing d_2 at a valid place within d_1, according to the integration model Im.

C.5.1 XMLPipe valid documents

Valid XMLPipe integration model composition ($\underset{Im_X}{\overset{+}{\leftarrow}}$) page 85

For all $d, d' \in D_Q$, where d' is rooted at a construct σ',

$$d \underset{Im_X}{\overset{+}{\leftarrow}} d' = \begin{cases} \emptyset & , \text{ if } \sigma' \notin \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} langConstructs^{HC}(L) \\ \{d_1, \ldots, d_n\} & , \begin{array}{l} d_1, \ldots, d_n \text{ result from placing } d' \text{ at any place in } d \\ \text{and } \sigma' \in \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} (FOC(L) \cup SMC(L)) \end{array} \\ \{d_1, \ldots, d_n\} & , \begin{array}{l} d_1, \ldots, d_n \text{ result from placing } d' \text{ at a place in } d \text{ where} \\ \textit{content is expected} \text{ and } \sigma' \in \bigcup\limits_{\forall L \in \mathcal{L}_{d'}} COC(L) \end{array} \end{cases}$$

XMLPipe valid documents ($\mathcal{V}^{Im_X}$) page 85

For a set of languages $\mathcal{L}_1$, the set of the XMLPipe integration model valid documents $\mathcal{V}^{Im_X}_{\mathcal{L}_1}$ is defined as follows:

$$d \in \mathcal{V}^{Im_X}_{\mathcal{L}_1} \text{ iff } \begin{cases} \mathcal{L}_d = \{L\},\ d \text{ is rooted at } \sigma \in (COC(L) \cup SMC(L)) \cap langConstructs^{HC}_e(L) \text{ and } d \text{ is a valid tree of } L \\ OR \\ \exists d_1, d_2, \text{ where } d_1 \in \mathcal{V}^{Im_X}_{\mathcal{L}_1},\ \mathcal{L}_{d_2} = \{L_2\},\ d_2 \text{ is rooted at } \sigma_2 \in langConstructs^{HC}(L) \text{ and } d_2 \text{ is a valid tree of } L_2, \text{ so that } d \in d_1 \underset{Im_X}{\overset{+}{\leftarrow}} d_2 \end{cases}$$

C.5.2 Further valid document definitions

Corollary 1 provides a simplified valid documents definition. It is useful when individually addressing a document's subtrees, because it defines valid mixed namespace documents as compositions of valid single namespace documents (as opposed to the previous recursive definition)

Corollary 2 page 123

$d \in \mathcal{V}^{Im_X}_{\mathcal{L}_d}$ if and only if there is a sequence of $n \geq 1$ single namespace valid documents $d_1, \ldots, d_n$, so that $d \in (d_n \cdots \underset{Im_X}{\overset{+}{\leftarrow}} (d_3 \underset{Im_X}{\overset{+}{\leftarrow}} (d_2 \underset{Im_X}{\overset{+}{\leftarrow}} d_1)) \cdots)$, where $\forall i \in [1, n]$, d_i is a valid single namespace document of L_i ($d_i \in L_i$), it is rooted at a handled construct $\sigma_i \in langConstructs^{HC}(L_i)$ and σ_n can only be an element COC or SMC handled construct.

Additionally, Proposition 1 is an adaptation of the valid documents definition to document processing. It provides the foundation for proving that valid documents have well defined processing.

Proposition 1 page 124

All valid XMLPipe documents have well defined processing if

- d has well defined processing $\forall d \in \mathcal{V}_{\mathcal{L}_d}^{Im_X}$, where $\mathcal{L}_d = \{L\}$ and d is rooted at $\sigma \in langConstructs_e^{HC}(L) \cap (COC(L) \cup SMC(L))$.
- All documents in $d_1 \xleftarrow[Im_X]{+} d_2$ have a well defined processing, if $d_1 \in \mathcal{V}_{\mathcal{L}_{d_1}}^{Im_X}$, $\mathcal{L}_{d_2} = \{L_2\}$, d_2 is rooted at $\sigma_2 \in langConstructs^{HC}(L_2)$ and both d_1 and d_2 have well defined processing.

C.6 Validation

A validation process is defined by a validation specification. Separate symbols are used to represent validation processes and validation specifications. *Validation processes* are represented by V: V, $V_1, V_2, \ldots, V_n$. *Validation specifications* are represented by the symbol vs: vs, $vs_1, vs_2, \ldots, vs_n$.

Document validation can be considered as a form of transformation. The validation notation reflects its relationship with transformation. If the validation of a document d by a validation process V results in d', we will write $d \xrightarrow{V} d'$. The validation process can either succeed or fail, if the document is valid or invalid, respectively.

C.6.1 Validation of XMLPipe documents

XMLPipe atomic validation processes ($\mathcal{A}^{\mathcal{V}}$) page 157

The set of all *XMLPipe atomic validation processes* $\mathcal{A}^{\mathcal{V}}$ is the subset of all transformations that map an XMLPipe document to its validated form. For each $V \in \mathcal{A}^{\mathcal{V}}$ the validation process $d \xrightarrow[I]{V} d'$ is successful if and only if d is a valid document, according to the external input I. d' is the validated result.

Validation algorithm (*validate*) page 160

validate validates the subtree of d that is rooted at the node n and results in the validated document.

function *validate*(D d, *Node* n, *Boolean deep*) $\rightarrow D$
  let $\sigma = (uri, s)$ be the construct that corresponds to n
  let L be the language that corresponds to σ
  //Instantiate the atomic validation for the identified subtree
  let $VS = locateVS(uri)$
  if ($\sigma \notin langConstructs^{HC}(L)$ OR $VS == \emptyset$) the validation **fails**; **end if**
  let *vs* be the most appropriate member of VS (implementation specific)
  let $V \in \mathcal{A}^{\mathcal{V}}$ be the validation process that corresponds to *vs*
  //Separate the subtrees
  for each subtree d_i of n rooted at a construct σ_i of a language $L_i \neq L$
    separate d_i from d
    if ($\sigma_i \in COC(L_i)$) AND ($deep == true$)

add a $(uri, \texttt{coc})$ element at its place
end if
end for
//apply the validation
let d' be the subtree of d, which is rooted at n
if $\sigma \in langConstructs_a(L)$ OR $\sigma \in FOC(L)$
if n has an ancestor n''
let $d' = (uri, \texttt{foc}) \xleftarrow{+} d'$ *//add the predefined* `foc` *construct*
else
the validation **fails**
end if
end if
apply V to d': $d' \xrightarrow{V} d''$
if the application of V was unsuccessful
the validation **fails**
end if
//re-compose the subtree and recurse
remove all $(uri, \texttt{foc})$ and $(uri, \texttt{coc})$ elements from d''
replace d' with d'' within d
for each separated subtree rooted at n_i
add the subtree at its corresponding place in d
if *deep* == *true*
call $validate(d, n_i, deep)$
end if
end for
the validation is **successful**; **return** d
end function

C.6.2 Processing validation

Processing validation wrappers (`validateSubtree` and `validateHC`)

The processing validation interface consists of two separate wrappers that correspond to the transformation pipeline validation constructs: `validateSubtree` and `validateHC`. They both call $validate(d, n, deep)$, where d is their input document subtree, n is the topmost subtree handled construct and *deep* is either true, for `validateSubtree`, or false, for `validateHC`.

C.6.3 Authoring validation

Authoring validation (*validateAuth*) page 166

The authoring validation is a transformation that maps a document d to its validated output d'', and it is composed out of a pre-validation transformation step and a subsequent subtree validation step: $d \xrightarrow{validateAuth} d''$ iff $d \xrightarrow{transformAuth} d'$ and $d' \xrightarrow{validate} d''$

The XMLPipe validation driver implements the function *XMLPipeVal* that represents XMLPipe's validation interface to the document author.

XMLPipe validation driver (*XMLPipeVal*) page 184

The function $XMLPipeVal : URI \rightarrow D$ represents the authoring validation XMLPipe interface. For a $uri \in URI$, $XMLPipeVal(uri) = d'$ where $parse(uri) = d$ and $d \xrightarrow{validateAuth} d'$

C.7 Transformations

We have introduced a generic and a more specific notation for expressing sets of transformations. The former is used prior to the introduction of the concept of integration models, and it does not separate between the languages of mixed namespace documents. The latter explicitly defines the input/output languages and integration models.

Simple set of transformations ($\mathcal{T}_{L_1}^{L_2}$)

$\mathcal{T}_{L_1}^{L_2}$ will denote the set of all transformations that map documents of L_1 to documents of L_2.

Set of mixed namespace transformations ($\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2}$) page 93

Consider that $\mathcal{L}_2 \subset \mathcal{L}_1 \subset \wp(\mathcal{L})$, $\mathcal{L}_3 \subset \wp(\mathcal{L})$, and that Im_1 and Im_2 are two integration models. The set of mixed namespace transformations $\mathcal{T}_{\mathcal{L}_1:Im_1(\mathcal{L}_2)}^{\mathcal{L}_3:Im_2} \subset \mathcal{T}$ includes all transformations that process the constructs of languages in $\mathcal{L}_2$ for mapping an input document, which combines the constructs of the languages in $\mathcal{L}_1$ using the integration model Im_1, to an output document, which combines the constructs of the languages in $\mathcal{L}_3$, using the integration model Im_2.

Document transformation ($\xrightarrow{T}$)

$d_1 \xrightarrow[I]{T} d_2$, or $d_1 \xrightarrow{T} d_2$ for insignificant external transformation input, denotes the mapping of d_1 to d_2 according to a transformation T and an external input I.

C.7.1 XMLPipe document transformation

The XMLPipe transformation model defines two transformation algorithms: *transform* and *transformRev*. The former is based on the initial overly restrictive set of assumptions, summarised in Table 7.2, that were necessary for proving the feasibility of transforming presentation documents. *transformRev* is based on *transform*, but it uses a significantly less restrictive set of assumptions, summarised in Table 7.3.

Proposition 2 page 124

Under the assumptions of Table 7.2, all valid XMLPipe documents d can be transformed by a finite iterative transformation $T \in \mathcal{T}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}^{\mathcal{L}_p:Im_X}$ to their most adequate representation d', according to an adaptation profile pr.

XMLPipe mixed namespace transformation page 128

The function $transform : D \times Profiles \rightarrow D$ represents the transformation $T \in \mathcal{T}^{\mathcal{L}_p:Im_X}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}$, which maps a valid XMLPipe document d to its most appropriate representation d', according to a profile pr.

function $transform(D\ doc, Profiles\ pr) \rightarrow D$
  let $d' = d$
  let n be the first node of d', according to a postorder tree traversal
  while (true)
    let σ be the XML construct that corresponds to n
    let L be the language that corresponds to σ
    let n' be the the next postorder tree traversal node after n
    if $L \notin \mathcal{L}_p$ AND $\sigma \in langConstructs^{HC}(L)$
      if T' is the optimal transformation for L, according to profile pr
        if $\sigma \in (COC(L) \cup SMC(L)) \cap langConstructs^{HC}_e(L)$
          Separate d_1 from d', where d_1 is rooted at n.
        else
          Separate d_1 from d', where d_1 is rooted at the parent of n.
        end if
        apply T' to d_1: $d_1 \xrightarrow{T'} d'_1$
        if d'_1 is not an empty tree
          replace d_1 with d'_1, within d'
          let n be the first node of d'_1 according to a postorder traversal
        else
          let $n = n'$
        end if
      else //*there is no appropriate transformation*
        the transformation **fails**; **exit**
      end if
    else //*the n rooted subtree does not require separate processing*
      if n is the root of d'
        if all d' nodes belong to languages in $\mathcal{L}_p$
          the transformation is **successful**; **exit**
        else
          the transformation **fails**; **exit**
        end if
      else
        let $n = n'$
      end if
    end if
  end while
end function

Revised XMLPipe mixed namespace transformation page 137

The function $transformRev : D \times Profiles \rightarrow D$ represents the transformation $T \in \mathcal{T}^{\mathcal{L}_p:Im_X}_{\mathcal{L}_d:Im_X(\mathcal{L}_d)}$, which maps a valid XMLPipe document d to its most appropriate representation d', according to a profile pr.

function *transformRev*$(D\ d, Profiles\ pr) \rightarrow D$
  let $d' = d$
  let n be the first node of d', according to a postorder tree traversal
  while (true)
    let σ be the XML construct that corresponds to n
    let L be the language that corresponds to σ
    let n' be the next postorder tree traversal node after n
    if $\sigma \in langConstructs^{HC}(L)$
      if $bestHCTS(\sigma, pr) \neq \epsilon$
        if $\sigma \in (COC(L) \cup SMC(L)) \cap langConstructs_e^{HC}(L)$
          Separate d_1 from d', where d_1 is rooted at n.
        else
          Separate d_1 from d', where d_1 is rooted at the parent of n.
        end if
        let pr' contain all adaptation statements associated with L
        apply T' to d_1: $d_1 \xrightarrow{T'} d_1'$, using both pr and pr'
        let pr'' contain all L specific statements introduced by T'
        if $d_1' \neq d_1$ AND d_1' is not an empty tree
          associate pr'' with L and with the parent of n
          replace d_1 with d_1', within d'
          let n be the first node of d_1', according to a postorder traversal
        else
          if there is a language specific pr_1 associated to n' parent
            discard pr_1
          end if
          let $n = n'$
        end if
      else if $L \neq L'$ *//No appropriate transformation*
        the transformation **fails**; **exit**
      end if
    else *//the n rooted subtree does not require separate processing*
      if n the root of d'
        if all d' nodes belong to languages in $\mathcal{L}_p$
          the transformation is **successful**; **exit**
        else the transformation **fails**; **exit**
        end if
      else
        let $n = n'$
        if there is a language specific pr_1 associated with the parent of n'
          discard pr_1
  end if; **end if**; **end if**; **end while**
end function

The XMLPipe transformation driver implements the function *XMLPipeTrans* that represents XMLPipe's transformation interface to the document user.

XMLPipe transformation (*XMLPipeTrans*) page 184

The function $XMLPipeTrans : URI \times CProfiles \rightarrow D$ represents the XMLPipe interface

for document transformation. For a URI $uri \in URI$ and a composite profile $cpr \in CProfiles$, $XMLPipeTrans(uri, cpr) = d'$ where $parse(uri) = d$ and $ccompose(cpr) = pr$ and $transformRev(d, cpr) = d'$.

C.7.2 Authoring validation specific transformation

An authoring validation specific transformation process is necessary for preprocessing a document's *SMC* rooted subtrees. As opposed to *transform* and *transformRev*, the authoring validation specific *transformAuth* has only document parameter, because the authoring validation is adatation requirement independent.

Authoring validation transformation (transformAuth) page 165

transformAuth is a transformation that eliminates as many *SMC* rooted subtrees as possible.

function *transformAuth*(D d) $\rightarrow D$
  let pr be the dummy profile that states support for all languages in $\mathcal{L}_d$
  let $d' = d$
  let n be the first node of d', according to a postorder tree traversal
  while (true)
    let σ be the XML construct that corresponds to n
    let L be the language that corresponds to σ
    let n' be the the next postorder tree traversal node of n
    if $L \in SMC(L)$ AND $bestHCTS(\sigma, pr) \neq \epsilon$
⟸ **if** $\sigma \in langConstructs_e^{HC}(L)$
        Separate d_1 from d', where d_1 is rooted at n.
      else
        Separate d_1 from d', where d_1 is rooted at the parent of n.
      end if
      let pr' be all adaptation statements associated with L
      apply a handled construct validation to d_1
      if the validation was successful
        apply T' to d_1: $d_1 \xrightarrow{T'} d_1'$ using both the initial pr and pr'
      else $d_1' = d_1$
      end if
      let pr'' be the set of all L-specific statements introduced by T'
      if $d_1' \neq d_1$ AND d_1' is not an empty tree
        Associate pr'' with L and with the parent of n
        Replace d_1 with d_1' within d'
        let n be the first node of d_1', according to a postorder traversal
      else
        if there is a language specific pr_1 for the parent of n'
          discard pr_1
        end if
        let $n = n'$
      end if
⟸ **else** *//the n rooted subtree does not require separate processing*
      if n the root of d'

the transformation is **successful**; **exit**

$\Longleftarrow$ **else**

let $n = n'$

if there is a language specific pr_1 associated with the parent of n'

discard pr_1

end if; **end if**; **end if**; **end while**

end function

C.7.3 Transformation specification selection

Optimal pipeline selection function (*bestHCTS*) page 131

The optimal pipeline selection function $bestHCTS : \Sigma \times Profiles \rightarrow PipeSpec$ maps a pair of a handled construct and an adaptation profile to their corresponding optimal pipeline specification. If $locateHCTS(\sigma) = \{(B_1, ps_1), \ldots, (B_n, ps_n)\}$ then $bestHCTS(\sigma, pr) = ps_k$, where

$$measure(pr, B_k) = \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) \neq 0$$

$$\text{OR}$$

$$B_k = \epsilon, \text{ if } \max_{\forall i, B_i \neq \epsilon} (measure(pr, B_i)) = 0$$

C.7.4 Transformation pipelines

Atomic transformations ($\mathcal{A}$) page 140

The set of *atomic transformations* $\mathcal{A}$ contains all functions $A : Profiles \rightarrow \mathcal{T}^{\mathcal{L}:Im_X}_{\mathcal{L}:Im_X(L)}$.

Sequence pipeline (*seq*) page 142

The sequence *seq* pipeline is a function

$$seq : \bigcup_{\forall \atop \mathcal{L}_1, \mathcal{L}_1', \mathcal{L}_2, \mathcal{L}_2', \mathcal{L}_3, Im_1, Im_2} \mathcal{T}^{\mathcal{L}_2:Im_2}_{\mathcal{L}_1:Im_1(\mathcal{L}_1')} \times \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_2:Im_2(\mathcal{L}_2')} \rightarrow \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_1:Im_1(\mathcal{L}_1' \cup \mathcal{L}_2')}$$

where $d \xrightarrow{seq(T_1,T_2)} d'$ iff $d \xrightarrow{T_1} d_1$ and $d_1 \xrightarrow{T_2} d'$.

Selection pipeline (*seq*) page 143

The selection *sel* pipeline is a function

$$seq : \bigcup_{\forall n, \forall i \in [1,n], \atop \forall \mathcal{L}_{i_1}, \mathcal{L}_{i_1}', \mathcal{L}_{i_2}, Im_1, Im_2} \left(\begin{array}{c} (\mathcal{T}^{\mathcal{L}_{1_2}:Im_{1_2}}_{\mathcal{L}_{1_1}:Im_{1_1}(\mathcal{L}_{1_1}')} \times (\mathcal{B} \cup \{\epsilon\})) \times \cdots \times \\ \times (\mathcal{T}^{\mathcal{L}_{n_2}:Im_{n_2}}_{\mathcal{L}_{n_1}:Im_{n_1}(\mathcal{L}_{n_1}')} \times (\mathcal{B} \cup \{\epsilon\})) \rightarrow \\ \rightarrow \mathcal{T}^{\mathcal{L}_{1_2} \cup \cdots \cup \mathcal{L}_{n_2}:Im_2}_{\mathcal{L}_{1_1} \cap \cdots \cap \mathcal{L}_{n_1}:Im_1(\mathcal{L}_1' \cup \cdots \cup \mathcal{L}_{n_1}')} \end{array} \right)$$

where for an adaptation profile pr, $sel((T_1, B_1), \ldots, (T_n, B_n)) = T_k$ where

$$measure(pr, B_k) = \max_{\forall i, B_i \neq \epsilon}(measure(pr, B_i)) \neq 0$$
$$\text{OR}$$
$$B_k = \epsilon, \text{ if } \max_{\forall i, B_i \neq \epsilon}(measure(pr, B_i)) = 0$$

Dynamic pipeline (*dyn*) page 144

The dynamic pipeline *dyn* pipeline is a function

$$dyn : \bigcup_{\substack{\forall \mathcal{L}_1, \mathcal{L}_1', \mathcal{L}_2, \mathcal{L}_2', \\ \mathcal{L}_3, \mathcal{L}_4, \mathcal{L}_4', \mathcal{L}_5, Im_1, \\ Im_2, Im_3, Im_4, Im_5}} \mathcal{T}^{\mathcal{L}_2:Im_2}_{\mathcal{L}_1:Im_1(\mathcal{L}_1')} \times \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_2:Im_2(\mathcal{L}_2')} \times \mathcal{T}^{\mathcal{L}_5:Im_5}_{\mathcal{L}_4:Im_4(\mathcal{L}_4')} \rightarrow \mathcal{T}^{\mathcal{L}_3:Im_3}_{\mathcal{L}_1:Im_1(\mathcal{L}_1' \cup \mathcal{L}_2')}$$

where $d \xrightarrow{dyn(T_1,T_2,T_3)} d'$ iff $d \xrightarrow{T_1} d_1$ and $d \xrightarrow{T_3} d_2$ and $d_1 \xrightarrow[d_2]{T_2} d'$.

XMLPipe pipelines (*Pipelines*) page 145

The set of all XMLPipe pipelines *Pipelines* contains all transformations in $\mathcal{T}^{\mathcal{L}:Im_X}_{\mathcal{L}:L(Im_X)}$, that correspond to all atomic transformations, subtree validation constructs and pipeline compositions of pipelines. For a profile pr,

$$p \in Pipelines \text{ iff } \begin{cases} p = A(pr) & , A \in \mathcal{A} \\ p = \texttt{validateSubtree} & \\ p = \texttt{validateHC} & \\ p = seq(p_1, p_2) & , p_1, p_2 \in Pipelines \\ p = dyn(p_1, p_2, p_3) & , p_1, p_2, p_3 \in Pipelines \\ p = sel(p_1, B_1, \cdots, p_n, B_n) & , \forall i \in [1, n] p_i \in Pipelines, B_i \in \mathcal{B} \end{cases}$$

C.8 XMLPipe adaptation model

C.8.1 Core concepts

Adaptation terms set (*Terms*) page 93

The set of all adaptation terms *Terms* is the set of all pairs of a URI and an XML local name:

$$Terms = URI \times S$$

where *URI* is the set of all URIs and S is the set of all XML local names.

Adaption term types (*TermTypes*) page 94

The set of all *adaptation term types* *TermTypes* contains all adaptation term data types

Numeric data type (*Numeric*) page 94

The *numeric type Numeric* ∈ *TermTypes* is the XMLPipe numeric data type, and its acceptable values are real numbers:
if $v \in Numeric$ then $v \in \mathbb{R}$

Boolean data type (*Boolean*) page 94

The *boolean type Boolean* ∈ *TermTypes* is the XMLPipe boolean data type, and it contains the values `true` and `false`:
$Boolean = \{\texttt{true}, \texttt{false}\}$

C.8.2 Adaptation profiles

Adaptation statements (*Statements*) page 96

The set of all adaptation statements *Statements* includes all pairs of a term and a value of its corresponding type.

$$Statements = \bigcup_{\forall \tau \in Terms} (\{\tau\} \times termType(\tau))$$

Adaptation profiles (*Profiles*) page 96

The set of all adaptation profiles *Profiles* includes all sets of adaptation statements:
$Profiles = \wp(Statements)$

Composite statements (*CStatements*) page 100

CStatements is the set of all composite statements and it contains all pairs of an adaptation statement and an optional binary adaptation expression, which maps two values of the corresponding term type to a value of the same type.

$$CStatements = \bigcup_{\forall \tau \in Terms} \left((\{\tau\} \times termType\,(\tau)) \times \left(\mathcal{F}^{termType(\tau)}_{(termType(\tau))^2} \cup \{\epsilon\} \right) \right)$$

Composite profiles (*CProfiles*) page 100

The set of all *composite profiles CProfiles* contains all composite statement *sequences*.

$$CProfiles = \bigcup_{\forall n \in \mathbb{N}} (CStatements)^n$$

C.8.3 Adaptation expressions

Adaptation expressions from a value tuple to a value ($\mathcal{F}^{Type'}_{\{Type_1,\ldots Type_n\}}$) page 94

The set $\mathcal{F}^{Type'}_{\{Type_1,\ldots Type_n\}}$ contains all XMLPipe expressions of the form $f : Type_1 \times Type_2 \times \cdots \times Type_n \rightarrow Type'$

Adaptation expressions from a single type value tuple ($\mathcal{F}^{Type_2}_{(Type_1)^n}$) page 94

$$\mathcal{F}^{Type_2}_{(Type_1)^n} = \mathcal{F}^{Type_2}_{\{\underbrace{Type_1,\ldots,Type_1}_{n}\}}$$

All adaptation expressions ($\mathcal{F}$) page 94

The set $\mathcal{F}$ includes all XMLPipe expressions:

$$\mathcal{F} = \bigcup_{\forall n\in\mathbb{N}, Type', \forall Type_1 \ldots Type_n} \mathcal{F}^{Type'}_{\{Type_1,\ldots,Type_n\}}$$

where $\mathbb{N}$ is the set of natural numbers {1, 2, ...}

Adequacy expressions (EAd) page 103

The set EAd of all adequacy expressions contains all pairs of terms and unary expressions that evaluate to a *Numeric* or a *Boolean* value.

$$EAd = \bigcup_{\forall\tau} \left(\left(\{\tau\} \times \mathcal{F}^{Numeric}_{(termType(\tau))^1} \right) \cup \left(\{\tau\} \times \mathcal{F}^{Boolean}_{(termType(\tau))^1} \right) \right)$$

Set of adequacy expression sets (BA) page 105

BA is the set of all binding expression sets: $BA = \wp(EAd)$

Applicability expressions (EAp) page 106

The set EAp contains all applicability expressions. Each EAp member is a pair of an n-tuple of terms and of an adaptation expression, which has a corresponding n-tuple of arguments. The expression must evaluate to either a *Numeric* or a *Boolean* value.

$$EAp = \bigcup_{\forall n\in\mathbb{N}} \bigcup_{\forall Type\in\{Numeric,Boolean\}} \bigcup_{\forall\tau_1,\ldots\tau_n\in Terms} \left(\{(\tau_1,\ldots,\tau_n)\} \times \mathcal{F}^{Type}_{\{termType(\tau_1),\ldots,termType(\tau_n)\}} \right)$$

C.8.4 Profile composition

The profile composition component of the XMLPipe adaptation model is responsible for mapping a composite adaptation profile to an adaptation profile. It implements function *ccompose*.

Profile composition (*ccompose*) page 101

The profile composition function $ccompose : CProfiles \rightarrow Profiles$, maps a composite profile $cpr \in CProfiles$ to its corresponding adaptation profile pr:

function $ccompose(cpr) \rightarrow pr$
  Let $cpr = (((\tau_1, v_1), f_1), ((\tau_2, v_2), f_2), \ldots, ((\tau_n, v_n), f_n))$
  Let $pr = \emptyset$
  for $(i = 1 \ldots n)$
    if $\nexists(\tau, v) \in pr$ where $\tau = \tau_i$ then
      $pr = pr \cup (\tau_i, v_i)$
    else
      if $f_i \neq \epsilon$ then
        $pr = (pr - (\tau, v)) \cup \{(\tau, f_i(v, v_i))\}$
      else if $termResolve(\tau) \neq \epsilon$ **then**
        $f = termResolve(\tau)$
        $pr = (pr - (\tau, v)) \cup \{(\tau, f(v, v_i))\}$
      else
        $pr = (pr - (\tau, v)) \cup \{(\tau, v_i)\}$
      end if
    end if
  end for
end function

C.8.5 Transformation selection

Adequacy measure function (*adequacy*) page 105

The adequacy measure function $adequacy : Profiles \times BA \rightarrow Numeric$ maps a profile and a set of adequacy expressions to the corresponding adequacy measure.

$$adequacy(pr, ba) = \sum_{\forall ead \in ba} mt(pr, ead)$$

where the function $mt : Profiles \times EAd \rightarrow Numeric$ is defined as:

$$mt(pr, (\tau, f)) = \begin{cases} 1 & , m \in (1, \infty) \cup \{true\} \\ m & , 0 \leq m \leq 1 \\ 0 & , m \in (-\infty, 0) \cup \{false\} \end{cases}, \text{ where}$$

$$m = \begin{cases} f(v) & , \exists(\tau, v) \in pr \\ f(termDefault(\tau)) & , \nexists(\tau, v) \in pr \end{cases}$$

Applicability measure function expression (*applicability*) page 106

The applicability measure function $applicability : Profiles \times EAp \rightarrow Numeric$ maps a pair of a profile and an applicability expression to the corresponding applicability measure:

$$applicability(pr, ((\tau_1, \ldots \tau_n), f)) = \begin{cases} 1 & , \gamma \in [1, \infty) \cup \{true\} \\ \gamma & , 0 \leq \gamma \leq 1 \\ 0 & , \gamma \in (-\infty, -1] \cup \{false\} \end{cases}$$

where $\gamma = f(v_1, \ldots v_n)$ and $\forall i \in [1, n]$

$$v_i = \begin{cases} v & , \exists(\tau_i, v) \in pr \\ v' & , \nexists(\tau_i, v) \in pr, termDefault(\tau_i) == v' \end{cases}$$

Adaptation measure function (measure) page 107

The adaptation measure function $measure : Profiles \times \mathcal{B} \rightarrow Numeric$ provides the absolute adaptation measure that corresponds to a binding specification, according to an adaptation profile.

$$measure(pr, (eap, ba)) = \begin{cases} 0 & , \text{if } eap = \epsilon, \ \exists ead \in ba \text{ so that} \\ & mt(pr, ead) = 0 \\ ad & , \text{if } eap = \epsilon, \text{ and} \\ & \forall ead \in ba, mt(pr, ead) > 0 \\ ap \cdot ad & , \text{ otherwise} \end{cases}$$

where $ad = adequacy(pr, ead)$, and $ap = applicability(pr, eap)$, and mt is the function defined in the adequacy measure function definition, in page 105.

C.9 Document parsing

DOM parser (parse) page 183

Function $parse : URI \rightarrow D$ represents the interface of a standards compliant DOM parsers, and it maps a URI *uri* to its corresponding DOM representation $d \in D$.

Appendix D

RDF integration example

This appendix includes the RDF/XML integration listings, described in Section 2.2.2.

Both Listing D.1 and Listing D.2 encode the same information, but the former combines two XML languages and the latter uses RDF. The encoded information is a structured document that contains author information, two sections, a nested subsection and formated text. In both cases there are two separate information areas: one covering the document structure and one covering the author information.

The first observation is that the first document can be intuitively interpreted by a person, but not necessarily by a process. A person could infer that the `authors` element, in line 6, describes the authors of the document. However, this association is not explicit and cannot be inferred by an XML processor that does not contain specific information on the integration of the two languages. For instance, a predefined integration specification can state that the `authors` element introduces a document's author, when it occurs before the document sections. However, a priori integration information is similar to integration profiles, which are not adequate for processing an open set of XML languages, as described Section 2.2.2.

The RDF example does not have such integration problems, because RDF enforces well defined associations between the separate information pieces. The only way to associate an author with a document is by a well defined association, which attaches the author of a resource to that resource. Lines 22 and 23 (Listing D.2) represent such an association. An RDF processor can use this association to explicitly infer the authors of the document. Consequently, the RDF explicit associations allow well defined integration, between separate information entities.

However, such explicit associations are closely tied to the RDF authoring model and are not compatible with the easy to author and ordered XML authoring model. Specifically, the explicit RDF associations and the rich processing information result in a document that is double the size of the XML document and significantly more difficult to author and comprehend. Consequently, as stated in 2.2.2, RDF based solutions are outside the scope of this thesis. Nevertheless, the main concept of well defined associations can provide the foundation for addressing the XML integration problems.

D.1 The mixed namespace XML document

```
<?xml version="1.0"?>
<doc xmlns="..."
     xmlns:a="...">
  <title>An example document</title>
  <date>18 June 2003</date>
  <a:authors>
    <a:author>
      <a:name>Mike</a:name>
      <a:surname>Pediaditakis</a:surname>
    </a:author>
    <a:author>
      <a:name>David</a:name>
      <a:surname>Shrimpton</a:surname>
    </a:author>
  </a:authors>

  <section><title>Introduction</title>
    Section tag can be used at <em>different</em> levels
    <section><title>Preliminaries</title>
      Such as in <strong>this</strong> section which
      is nested in the introduction.
    </section>
  </section>

  <section><title>Conclusion</title>
    The number of sections is unbounded.
  </section>
</doc>
```

Listing D.1: Integration Example: XML

D.2 The RDF document

```
<rdf:RDF
   xmlns:rdf='...'
   xmlns:docNs='...'
   xmlns:authNs='...'>
   <rdf:Description rdf:about='http://www.cs.kent.ac.uk/~mp49/MP'>
     <rdf:type rdf:resource='http://www.cs.kent.ac.uk/projects/XMLhandling
         /experiments/integration/refSet/rdf/authors#author'/>
     <authNs:first>Michael</authNs:first>
     <authNs:last>Pediaditakis</authNs:last>
     <authNs:homePage>http://www.cs.kent.ac.uk/~mp49</authNs:homePage>
   </rdf:Description>
   <rdf:Description rdf:about='http://www.cs.kent.ac.uk/~dhs/DHS'>
     <rdf:type rdf:resource='http://www.cs.kent.ac.uk/projects/XMLhandling
         /experiments/integration/refSet/rdf/authors#author'/>
     <authNs:first>David</authNs:first>
     <authNs:last>Shrimpton</authNs:last>
     <authNs:homePage>http://www.cs.kent.ac.uk/~dhs</authNs:homePage>
   </rdf:Description>

   <rdf:Description rdf:about=''>
      <rdf:type rdf:resource='http://www.cs.kent.ac.uk/projects/
          XMLhandling/experiments/integration/refSet/rdf/doc#document'/>
     <docNs:title>An RDF document</docNs:title>

```

```
22        <docNs:author rdf:resource="http://www.cs.kent.ac.uk/~mp49/MP"/>
23        <docNs:author rdf:resource="http://www.cs.kent.ac.uk/~dhs/DHS"/>
24
25        <!-- And the content -->
26        <docNs:content>
27          <rdf:Seq>
28             <rdf:li rdf:parseType="Literal">
29              A paragraph
30             </rdf:li>
31             <rdf:li>
32                <docNs:section>
33                   <docNs:content>
34                     <rdf:Seq>
35                       <rdf:li rdf:parseType="Literal">
36                          A paragraph of a section
37                       </rdf:li>
38                       <rdf:li>
39                         <docNs:formatedText>
40                           <docNs:content>
41                              <rdf:Seq>
42                                 <rdf:li rdf:parseType="Literal">
43                                   And another paragraph that has some
44                                 </rdf:li>
45                                 <rdf:li>
46                                   <docNs:emphasis>
47                                      <docNs:content>emphasized</
                                            docNs:content>
48                                   </docNs:emphasis>
49                                 </rdf:li>
50                                 <rdf:li rdf:parseType="Literal">
51                                   text
52                                 </rdf:li>
53                              </rdf:Seq>
54                           </docNs:content>
55                         </docNs:formatedText>
56                       </rdf:li>
57                     </rdf:Seq>
58                   </docNs:content>
59                </docNs:section>
60             </rdf:li>
61          </rdf:Seq>
62        </docNs:content>
63     </rdf:Description>
64  </rdf:RDF>
```

Listing D.2: Integration Example: RDF-XML

Appendix E

XMLPipe processing semantics representation

This appendix overviews the processing semantics representation that is used by the XMLPipe pilot implementation. Sections E.1 to E.2 introduce the top level representation concepts. The subsequent sections describe the syntax of the individual semantics information entities.

E.1 The top level structure

The processing semantics representation is in XML, because XML is adequate and its processing can reuse a preprocessing implementation's components. Specifically, XML is adequate, because XML documents can represent any information. Additionally, an XMLPipe implementation must also include, at a minimum, an XML parser that can be reused to process the processing semantics.

A well defined XML representation requires a well defined namespace URI. The namespace URI that corresponds to the XMLPipe processing semantics language is
`http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/XMLPipe`
The same URI is also used as a prefix for all other XMLPipe specific resources. Additionally, the `XMLPipeURI` abbreviation will refer to the XMLPipe URI, when necessary.

The top level processing semantics structure allows the binding model to locate the necessary processing semantics. We will only define the top level structure for the primary semantics repositories, because secondary repositories can use repository-specific semantics representations. Nevertheless, the returned semantics specifications must always follow the common processing semantics defined in the subsequent sections.

The principal semantics repositories contain RDDL links to the XML documents that contain the corresponding processing semantics. Listing E.1 illustrates four RDDL links to separate types of processing semantics. Specifically, an RDDL link is introduced by the `resource` element of the RDDL namespace. The `xlink:type` attribute must always be "`simple`", because all RDDL links are simple XLink links. The `xlink:href` attribute specifies the URL of the document that contains the corresponding semantics information. The `xlink:role` attribute specifies the role of an RDDL link. The XMLPipe model defined four URIs that correspond to the four types of processing semantics: the term declarations, the handled construct information, the validation semantics and the transformation semantics. The `xlink:role` attribute of each RDDL link can use one of these URIs to specify the type of XMLPipe semantics it links to.

The four separate `xlink:role` values, illustrated in lines 4, 7, 10 and 13, correspond to the handled construct information, the adaptation term declarations, the validation semantics and the transformation semantics, respectively.

```
1  <xhtml:html>
2  ...
3    <rddl:resource xlink:type="simple"
4          xlink:role="XMLPipeURI#intModelSemantics"
5          xlink:href="HCInfo.xml"></rddl:resource>
6    <rddl:resource xlink:type="simple"
7          xlink:role="xmlPipeURI#termSemantics"
8          xlink:href="TermInfo.xml"></rddl:resource>
9    <rddl:resource xlink:type="simple"
10         xlink:role="xmlPipeURI#valSemantics"
11         xlink:href="ValInfo.xml"></rddl:resource>
12   <rddl:resource xlink:type="simple"
13         xlink:role="xmlPipeURI#transSemantics"
14         xlink:href="TransInfo.xml"></rddl:resource>
15 ...
16 </xhtml:html>
```

Listing E.1: RDDL processing semantics links

The root element of all XMLPipe processing semantics documents, such as the RDDL link targets, is the `XMLPipeURI:config` element that is the parent element of all types of processing definitions. Its `name` attribute specifies the name of the corresponding collection of processing semantics.

```
<xmlPipe:config name="XMLPipeSemanticsExample">
...
</xmlPipe:config>
```

E.2 Data types and expressions

Each XMLPipe adaptation term is associated with a well defined data type, which is necessary for validating the adaptation statements and evaluating the adaptation expressions. This section describes the predefined XMLPipe data types, their associated operators and the individual types of adaptation expressions.

E.2.1 The predefined XMLPipe data types

Only a minimal set of data types is necessary, because most necessary adaptation expressions describe simple numerical and string operations. The XMLPipe pilot implementation defines four data types: *Numeric*, *Boolean*, *String* and *SetOfStrings*. Table E.1 illustrates the XMLPipe data types. The *Numeric*, *Boolean* and *String* data types represent the common notions of a number, a boolean value and a string. The *SetOfStrings* is necessary for representing sets of URIs, which are used to describe URI lists, such as the set of natively supported languages.

The adaptation expressions use a set of binary and unary operators/functions. Table E.2 illustrates three XMLPipe unary operators: `not`, `abs` and `round`. `not` is the boolean NOT operator and applies to boolean values. `abs` and `round` correspond to the

Data Type	Example value	Description
Numeric	-1, 2.5, 100	Integers and decimals
Boolean	*true*, *false*, 1, 0	Boolean values
String	"Any string value"	Arbitrary length strings
SetOfStrings	`<item>String1</item>` `<item>String2</item>`	A set of strings, where each string is denoted by an `<item>` element

Table E.1: Adaptation terms data types

Operator	Data types
not	*Boolean*
abs	*Numeric*
round	*Numeric*

Table E.2: Unary operators

absolute and rounded value of a decimal and apply to *Numeric* values. Table E.3 illustrates the XMLPipe binary operators. `add`, `sub`, `div` and `mul` apply the corresponding mathematical operations, between their two numeric arguments. In a similar manner `and` and `or` apply their corresponding boolean operations. The operator `equals` applies to all data types and returns *true*, when its two operands are equal. Finally, `contains` and `union` test set membership and perform set union, respectively.

E.2.2 Adaptation expressions

The XMLPipe adaptation expressions consist of a hierarchical structure of operators, functions, literal data values and references to term values. The XML representation of each operator is an element with the same name as the operator. Unary operator elements can only contain one operant child element, and binary operator elements can contain two child elements. Such a hierarchical representation can lead to lengthy expression definitions. However, it allows straightforward processing and, since most XMLPipe adequacy expressions do not involve more than two operators, it can be considered as an acceptable simplification.

The `val` element introduces literal data values, within an adaptation expression. For instance: `<val>1.4</val>` and `<val>String</val>` introduce a *Numeric* and a

Datatype (l/r)	*Numeric*	*Boolean*	*String*	*SetOfStrings*
Numeric	add, div, mul, sub, equals			
Boolean		and, or, equals		
String			equals	
SetOfStrings			contains	equals,union

Table E.3: Binary operators

String value, respectively. The XMLPipe expression evaluator does not automatically recognise the value types. According to the type of the adaptation expression, there can be a default data type. In such cases, the expression evaluator assumes that every unqualified literal is of the default data type. Literal values that do not belong to the default data type must use the `type` attribute. For instance the following value will always be interpreted as a string, independently of its context:

```
<val type="String">2.45</val>
```

A reference to the value of an adaptation term depends on the type of the adaptation expression, because each type relates to a different set of terms. The following sections will describe the separate methods of referencing term values. This section's example will use the most precise available notation, where both the name and the namespace of the referenced term are provided.

The `termVal` element introduces the value of the term that corresponds to its `name` and `ns` attribute values. For instance, Listing E.2 illustrates an example adaptation expression. If `uri:name1` and `uri:name2` are adaptation terms and their corresponding values are x and y, the illustrated example represents the expression

$$(x * 30) + (80 - y) == 20$$

```
1  <not>
2   <equals>
3    <add>
4     <mul>
5      <termVal ns="uri" name="name1"/>
6      <val type="Numeric">30</val>
7     </mul>
8     <sub>
9      <val type="Numeric">80</val>
10     <termVal ns="uri" name="name2"/>
11    </sub>
12   </add>
13   <val type="Numeric">20</val>
14  </equals>
15 </not>
```

Listing E.2: Adaptation expression example

E.2.3 Applicability expressions

The XMLPipe applicability expressions are the most comprehensive adaptation expressions, because they can arbitrarily combine the values of several terms. The only restriction is that they must result to either a numeric or a boolean value.

Within an applicability expression there is no default literal value type. Consequently, all literals must use the `type` attribute. Additionally, the value of any adaptation term can be used, and both its namespace and local name must be explicitly specified.

For instance, the representation of the applicability expression

$$NOT(X)\ AND\ (Y\ CONTAINS\ \text{“S1”})\ AND\ NOT(Y\ CONTAINS\ \text{“S2”})$$

```
1  <applicability>
2    <and>
3      <and>
4        <not>
5          <termVal ns="..."
6                   name="x"/>
7        </not>
8        <contains>
9          <termVal ns="..."
10                  name="Y"/>
11          <val type="String">S1</val>
12       </contains>
13     </and>
14     <not>
15       <contains>
16         <termVal ns="..."
17                  name="Y"/>
18         <val type="String">S2</val>
19       </contains>
20     </not>
21   </and>
22 </applicability>
```

Listing E.3: Applicability expression example

is illustrated in Figure E.3. All `termVal` elements (lines 5, 9, 16) use both the `ns` and `name` attributes to specify the corresponding adaptation term URI. Additionally, both string literals (lines 11 and 18) explicitly specify their type.

E.2.4 Adequacy expression sets

An adequacy expression set consists of one or more adequacy expressions. Each adequacy expression is restricted to using the value of a single term, as opposed to the applicability expressions. The proposed syntax enforces this constraint by only allowing the top level expression element to declare a term's URI. The actual references to the term value are represented by a `termVal` element, that has no `ns` and `name` attributes.

Listing E.4 illustrates the adequacy expression set {X *CONTAINS* "S1", Y = "S2"}. Each `expr` element introduces a set of adequacy expressions, and its `ns` and `name` attributes specify the corresponding adaptation term.

The type of each expression's adaptation term specifies the default expression data type. Consider that the data type of X is *SetOfStrings* and the data type of Y is *String*. The first occurrence of `val` explicitly specifies its data type, because it differs from the data type of X, which is the default expression data type. In contrast, the second occurrence of `val` (line 13) does not require such a declaration, because its *String* data type is the same as the default expression data type.

E.2.5 Conflict resolution expressions

Conflict resolution expressions can only access the values of a single term, in a similar manner to the adequacy expressions. However, they can refer to two separate values of the same term: a previous value and a newly introduced conflicting value. Their result

```
<adequacy>
  <expr ns="..."
        name="X">
    <contains>
      <termVal/>
      <val type="String">S1</val>
    </contains>
  </expr>
  <expr  ns="..."
         name="Y">
    <equals>
       <termVal/>
       <val>S2</val>
    </equals>
  </expr>
</adequacy>
```

Listing E.4: Adequacy expressions example

value must be of the same type as the corresponding adaptation term, because their result is used as the new term's value.

Conflict resolution expressions do not have to explicitly specify their corresponding term, because it can always be inferred from their context. Specifically, conflict resolution expressions can only appear within adaptation term semantics and composite profile statements. In the first case the resolution expression term is the term described by the processing semantics. In the latter case, the resolution expression term is the one referred by the composite adaptation statement.

In order to differentiate between the two term values, in a well defined manner, two value reference elements are used: `termVal` and `prevTermVal`. The former refers to the newly introduced conflicting value and the latter to the previous term value.

```
<resolution>
  <termVal/>
</resolution>

<resolution>
  <add>
    <mul>
       <prevTermVal/>
       <val>10</val>
    </mul>
    <termVal/>
  </add>
</resolution>
```

Listing E.5: Conflict resolution expressions example

For instance, consider a term T that has been associated with a value v_1 and that a new conflicting value v_2 is introduced. Listing E.5 illustrates two valid conflict resolution expressions. The former is equivalent to v_2, and it is the same as the default

XMLPipe resolution mechanism. The latter is equivalent to $(v_1 \times 10) + v_2$.

E.3 Adaptation processing semantics

The adaptation processing semantics consist of the adaptation term semantics, the composite adaptation profiles and the binding adaptation specifications. The adaptation term semantics provide the necessary information for processing adaptation statements and adaptation expressions. The composite profiles represent the set of adaptation requirements that a document must be adapted for. The binding adaptation specifications allow the optimal transformation selection. They will not be described in this section, because their syntax has been covered in the aforementioned adequacy/applicability expression syntax and their use will be illustrated in the following transformation semantics discussion.

E.3.1 Adaptation term semantics

```
1  <terms ns="URI1">
2
3    <term name="termLocalName" type="Boolean">
4      <default>true</default>
5    </term>
6
7    <term name="term2LocalName" type="SetOfStrings">
8      <default/>
9      <resolution>
10       <union>
11         <prevTermVal/>
12         <termVal/>
13       </union>
14     </resolution>
15   </term>
16
17 </terms>
```

Listing E.6: Term semantics example

The semantics of an adaptation term associate its qualified name with its data type, its default value and an optional conflict resolution expression. Listing E.6 illustrates an example declaration of two adaptation terms.

All term declarations are enclosed within the `terms` element, which specifies their URI. If a semantics repository uses a single document to describe multiple-URI terms, multiple `terms` elements must be introduced.

The first minimal semantics declaration specifies that the acceptable data values of the `URI:termLocalName` term are boolean values, and that its default value is "true". There is no explicit conflict resolution expression, and in the case of profile composition conflicts the default XMLPipe resolution mechanism will be used.

The second declaration introduces a *SetOfStrings* term, which has an empty-set default value and an explicit conflict resolution expression (the `resolution` element). The resolution expression returns the union of the two conflicting values. Such a

resolution could be adequate for an adaptation term that specifies the set of natively supported technologies.

E.3.2 Composite profiles

The preprocessing initiation entity must provide a composite adaptation profile to specify a set of adaptation requirements. A composite adaptation profile is a *sequence* of composite adaptation statements. Each statement can either specify a term's value or include a sequence of statements from an external composite profile. Additionally, a composite adaptation statement can contain a conflict resolution expression, in order to override the default term or XMLPipe conflict resolution mechanisms.

```
1  <profiles>
2    <profile name="aMobile">
3      <include
4        ref="compositeProfile1URL"
5        name="DefaultMobileProfile"/>
6      <include
7        ref="compositeProfile2URL"
8        name="SoftwareUpdateMobileProfile"/>
9      <statm ns="XMLPipeURI/Terms"
10       name="maxImageX">96</statm>
11     <statm ns="XMLPipeURI"
12       name="maxImageY">
13       <val>100</val>
14       <resolution>
15         <prevTermVal/>
16       </resolution>
17     </statm>
18   </profile>
19 </profiles>
```

Listing E.7: Composite adaptation profile example

Listing E.7 illustrates the declaration of a composite adaptation profile. In a similar manner to most processing semantics constructs, all profile declarations are enclosed within a `profiles` element. The `profile` element introduces a composite profile, and its `name` attribute provides the profile name and identifier. The first two child elements include the statements of composite profiles from other processing semantics documents. Each `include` statement specifies the URL of the processing semantics file and the name of the included composite profile (since a processing semantics document can contain several profile declarations). The first `statm` element assigns the value "96" to the (`XMLPipeURI/Terms`, `maxImageX`) term. The last statement describes a similar association and also contains a statement specific conflict resolution expression. When conflict resolution expressions are specified, the `val` element is used to enclose the term value, in order to separate it from the expression. In this case, the statement value is "100" and the conflict resolution expression always chooses the oldest between two conflict values (the opposite of the default XMLPipe resolution mechanism).

Local name	Data type	Description
supported	*SetOfStrings*	The namespace URIs of all natively supported languages and representations.
deviceType	*String*	The type of the target device. Typical values are: desktop, mobile, printer, text, etc.
baseURI	*String*	The URI of a node's primary source.
line	*Numeric*	The node's line within its primary source.
column	*Numeric*	The node's column within its primary source.
transformations	*SetOfStrings*	The URLs of the applied transformation specifications.
outputURI	*String*	The base URI for output documents.

Table E.4: Predefined adaptation terms

E.3.3 Predefined adaptation terms

XMLPipe allows the use of an open set of adaptation terms, in order to allow the expression of all necessary adaptation requirements. Nevertheless, there is a set of predefined adaptation terms that allow the well defined interoperation between XMLPipe and the individual transformers. Table E.4 summarises the core predefined adaptation terms. It lists only their local names, because they all share the common XMLPipe term URI: `XMLPipeURI/Terms`.

The most commonly used term is `supported`, and it specifies the list of natively supported technologies. Most binding adaptation specifications use its value and each adaptation profile must provide the namespace URI of at least a supported technology. `deviceType` is also commonly used for the primary classification of the target device and shares the same values with the CSS device types.

`baseURI`, `line`, `column` and `transformations` are used for the node context information and provide information on the source of a node and the applied processing. Specifically, `baseURI`, `line` and `column` provide its source document and its location within it. `transformations` contains a list of all the transformations that have been applied to it. The node context information can be used explicitly by error reporting processes or implicitly by the individual transformations. An error reporting process can use a node's location information to produce an error message. Subtree transformations may use this information implicitly, because, before processing a document subtree, XMLPipe temporarily adds the node context information of the subtree's root node to the adaptation profile.

Finally, the combination of `outputURI` and `baseURI` provide the necessary information for resolving relative URLs. `baseURI` defines a node's source and is a good candidate for resolving any relative input URLs. `outputURI` can be set by the preprocessing initiation entity, in order to specify the base location of any generated content.

E.4 Handled construct declarations

The processing of a presentation document according to the XMLPipe preprocessing model requires information on the handled constructs of its languages. A subset of each presentation language's constructs are handled constructs. A handled construct must belong to exactly one of the three predefine categories: *COC*, *FOC* or *SMC*. Additionally, a handled construct can be either an element or an attribute.

```
1  <constructs ns="DOC URI">
2    <hc name="document" class="coc" node="el"/>
3    <hc name="em" class="coc" node="el"/>
4    <hc name="img" class="coc" node="el"/>
5    <hc name="p" class="coc" node="el"/>
6  </constructs>
7
8  <constructs ns="IMPORT URI">
9    <hc name="import" class="smc" node="el"/>
10 </constructs>
11
12 <constructs ns="XLink URI">
13   <hc name="href" class="foc" node="at"/>
14   <hc name="type" class="foc" node="at"/>
15 </constructs>
```

Listing E.8: Handled construct information example

Listing E.8 illustrates the declaration of three sets of handled constructs. Each `consructs` element corresponds to an XML language and encloses handled construct declarations that share the same namespace URI. A `hc` element declares a handled construct, and it must specify its local name, its classification and whether it is an element or an attribute. For instance, the language that correspond to `DOC URI` has four handled constructs, which are all element *COC* constructs. In contrast, the language that correspond to `IMPORT URI` has a single *SMC* element handled construct. Finally, the language that corresponds to the `XLink URI` has two *attribute FOC* constructs.

E.5 Validation processing semantics

Each language's validation semantics consist of a set of validators and a set of atomic validations, which are associated with the language's URI. Listing E.9 illustrates the declaration of a validator and of an atomic validation that uses the declared validator.

All validator declarations must occur within the `validators` element. A validator is introduced by the `validator` element, which specifies its name, its Java implementation class and its Java implementation URL. A validator's name must be unique, within each processing semantics document, because the atomic validation declarations reference the necessary validators using their name. The Java implementation class is the main validator Java class, which must implement the XMLPipe atomic validation interface. The implementation URL defines the location of a `.class` or `.jar` file that contains the validator class.

The `validation` element introduces the validation semantics of the language that corresponds to the URI specified by its `ns` attribute. `validation` can contain several

```
<validators>
  <validator
      name="validatorName"
      implClass="javaPackage.JavaClass"
      implSource="ImplementationURL"/>
</validators>

<validation ns="DOC_URI">
  <atomVal ref="validatorName"
           src="SchemaURL"/>
</validation>
```

Listing E.9: Validation semantics example

`atomVal` elements, which introduce the alternative atomic validations that are adequate for the specified language. Each `atomVal` element uses its `ref` attribute to refer to a validator implementation. All referenced validators must be declared within the same processing semantics document. Additionally, `atomVal` specifies the location of a schema specification, if necessary. For instance, if `validatorName` is an XML Schema validator, then the `atomVal src` attribute can refer to the XML Schema specification that defines the syntax of `DOC URI`.

E.6 Transformation processing semantics

The transformation processing semantics are similar to the validation semantics, but require significantly more information. Specifically, they consist of atomic transformers, their declarations, their composition into transformation pipelines and the pipelines association with handled constructs, according to binding adaptation specifications.

E.6.1 Transformer declarations

```
<transformers>
  <transformer name="XMLPipe_DXSLT"
               implClass="javaPackage.JavaClass"
               implSrc="ImplementationURL"
               dynamic="true"/>

  <transformer name="XMLPipe_XSLT"
               implClass="javaPackage.JavaClass"
               implSrc="ImplementationURL"
               dynamic="false"/>

</transformers>
```

Listing E.10: Transformer declarations example

A transformer is similar to a validator and binds a name identifier to a subtree

transformer implementation. Listing E.10 illustrates the declaration of two transformers. The `transformer` element introduces a transformer and it can only occur within a `transformers` element. Each XMLPipe transformer has a name, which is used as a reference within the transformation pipelines. Additionally, it has an implementation class which is the main transformer class and must implement the XMLPipe atomic transformation interface. The `implSrc` attribute provides a link to the file that contains the implementation class. Finally, the `dynamic` attribute specifies whether a transformer can have dynamic external input or not. If its input is dynamic, the transformer can be used as the combining transformation within a transformation pipeline. Otherwise, if it requires a parameter, an external document source (such as a stylesheet) must be explicitly specified whenever the transformer is used.

E.6.2 Pipeline declarations

All XMLPipe subtree transformations are described by transformation pipelines. A transformation pipeline is a nested composition of atomic transformations and validation primitives, according to three pipeline composition methods: sequence pipeline, dynamic pipeline and selection pipeline.

Listing E.11 illustrates a simple sequence pipeline. A transformation pipeline is introduced within a handler definition by the `pipe` element. Sequential composition is the default composition method and it does not have to be explicitly specified. The illustrated sequential pipeline consists of a handled construct subtree validation and an XSL-T transformation. `transform` introduces an atomic transformation by referencing a declared transformer (`ref` attribute) and defining an external information source, if necessary. The `validate` element can occur at all places where atomic transformations can occur, and it instructs the pipeline driver to perform a subtree validation process. When `wholeTree` is `false`, it performs handled construct validation. If `wholeTree` is `true`, it performs a deep subtree validation.

Listing E.12 illustrates a sequential composition of a deep subtree validation and a dynamic transformation pipeline. Dynamic transformation pipelines consist of two sub-pipelines and an atomic transformation. The `normal` and `option` elements contain transformations that independently process the input subtree. The final atomic transformation, illustrated in line 12, combines their output by applying its corresponding transformation to the output of `normal`, using the output of `option` as its external input. In the illustrated example, the dynamic pipeline applies the transformation stylesheet generated by the atomic transformation in line 9 to the output of the atomic transformation in line 5. The "`XMLPipe:dynamic`" value of the `src` `transform` attribute (line 13) is predefined and denotes that a transformation's input must be dynamically generated.

Listing E.13 illustrates a selection pipeline. Selection pipelines consist of a sequence of alternative sub-pipelines. Each sub-pipeline is introduced with the `case` element and can optionally have a set of adequacy expressions, enclosed in a `test` element. The pipeline transformation driver applies either the sub-pipeline with the greatest adequacy measure or the first sub-pipeline with no adequacy expressions, if all adequacy measures are zero. The first illustrated sub-pipeline tests the value of a single term and performs a single atomic transformation. The second sub-pipeline also performs a single atomic transformation, but it has no associated adequacy expressions. Consequently, if (`termName`, `termURI`) is associated to *true* or a positive numeric value, the first atomic transformation will be used. Otherwise, the second atomic transformation will be

```
<pipe>
  <validate wholeTree="false"/>
  <transform ref="XMLPipe_XSLT"
             src="stylesheet1URL"/>
</pipe>
```

Listing E.11: Sequence transformation pipeline example

```
<pipe>
   <validate wholeTree="true"/>
   <dynamic>
     <normal>
       <transform ref="XMLPipe_XSLT"
                  src="stylesheet1URL"/>
     </normal>
     <option>
       <transform ref="XMLPipe_XSLT"
                  src="stylesheet2URL"/>
     </option>
      <transform ref="XMLPipe_DXSLT"
                 src="XMLPipe:dynamic"/>
   </dynamic>
</pipe>
```

Listing E.12: Dynamic transformation pipeline example

```
<pipe>
  <selection>
    <case>
      <test>
        <expr name="termName" ns="termURI">
          <termVal/>
        </expr>
      </test>
      <transform ref="XMLPipe_XSLT"
                 src="stylesheet4URL"/>
    </case>
    <case>
       <transform ref="XMLPipe_XSLT"
                  src="stylesheet5URL"/>
     </case>
  </selection>
</pipe>
```

Listing E.13: Selection pipeline example

applied.

E.6.3 Top level handlers declaration

The handler declarations define the transformation semantics of languages, by associating handled constructs to transformation pipelines, according to the adaptation requirements. A handler declaration consists of a name, a list of relevant handled constructs, an optional applicability expression, an optional set of adequacy expressions, an optional set of adaptation statements and a transformation pipeline. Listing E.14 illustrates a an example handler declaration.

```
<handler name="Doc_handler">
  <hcList ns="LanguageURI">
    <hcRef name="document"/>
    <hcRef name="em"/>
    <hcRef name="img"/>
    <hcRef name="p"/>
  </hcList>
  <applicability>
    <contains>
      <termVal ns="XMLPipeURI/Terms"
               name="termNamesupported"/>
      <val type="String">Language2URI</val>

    </contains>
  </applicability>
  <adequacy>
    <expr  ns="XMLPipeURI/Terms"
           name="supported">
      <contains>
        <termVal/>
        <val type="String">Language2URI</val>
      </contains>
    </expr>
    <expr  ns="XMLPipeURI/Terms"
           name="deviceType">
      <equals>
        <termVal/>
        <val>desktop</val>
      </equals>
    </expr>
  </adequacy>
  <context>
    <statm ns="termURI" name="termName">
    Value
    </statm>
  </context>
  <pipe>
    <transform ref="XMLPipe_XSLT"
               src="stylesheetURL"/>
  </pipe>
</handler>
```

Listing E.14: Handler declaration example

The `handler` element introduces a handler declaration and its `name` attribute specifies the handler's name. The only required children are the `hcList` and `pipe` elements. The former specifies the list of handled constructs that the handler can process. The latter introduces the transformation pipeline that processes them.

The binding adaptation specification consists of the optional adequacy and applicability expressions. If none exists, XMLPipe considers the handler to be universally applicable and adaptation requirements independent. If there is an adequacy expression, XMLPipe uses it to evaluate the handler's adequacy measure, in order to choose optimal handler for each document subtree. If there is an applicability expression, XMLPipe uses it to evaluate the handler's applicability measure, which specifies whether it is applicable for an adaptation profile. If there is no applicability expression, XMLPipe considers a handler applicable when all adequacy expressions evaluate to non-zero adequacy measures.

The optional `context` element provides a straightforward way to introduce temporary adaptation profile modifications, and its main application is the introduction of language specific adaptation statements. `context` contains a composite adaptation profile. XMLPipe integrates the introduced profile with the existing adaptation profile, before applying the pipeline transformations. If no language specific statements are introduced, all changes are discarded after the subtree transformation. Otherwise, they remain visible by all transformations within the same subtree that also correspond to the same language.

Appendix F

Case study sources

Section 10.4 illustrated the feasibility of the XMLPipe preprocessing model by describing a comprehensive case study that covered the validation and transformation of a presentation document, according to three adaptation profiles. It only described the details of a representative subset of the processing semantics and it abbreviated lengthy constructs (such as the resource URIs), in order to comply with main text layout.

This appendix contains all the case study information and processing semantics, in an unabbreviated form. Section F.1 includes the three case study input documents. Section F.2 describes the declaration of case study wide and XMLPipe specific semantics, such as the built-in XSL-T atomic transformation. Section F.3 illustrates the three used composite profiles. The subsequent sections F.4, F.5, F.6 and F.7 describe the necessary processing semantics for the L_{imp}, L_{alt}, L_{doc} and L_{xl} languages, respectively. semantics. Finally, Section F.8 illustrates the processing semantics of the additional L_{cd} language.

F.1 Input document

The case study input consists of three separate documents. `document.xml` illustrates the main processed document, which contains references to `imp.xml` and `authors.xml`.

```
<doc:document
    xmlns:doc="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/document"
    xmlns:imp="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/import"
    xmlns:alt="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt"
    xmlns:xl="http://www.w3.org/1999/xlink/">
 <alt:alt>
  <alt:case test="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
      XMLPipe/Terms#deviceType = mobile">
   <doc:title>Mobile example</doc:title>
  </alt:case>
  <alt:case>
   <doc:title>Desktop example</doc:title>
  </alt:case>
 </alt:alt>
 <imp:import href="authors.xml"
     select="//*/*[@id='MP_DHS']"/>

 <doc:section>
  <doc:title>The doc language</doc:title>
  <doc:p>The root language allows <doc:em>emphasized</doc:em> text,
    images <doc:img href="xmlPipe.gif"/> and nested sections.</doc:p>
```

```
21    <doc:section>
22     <doc:title>Nested section</doc:title>
23     <doc:p>This is a nested section</doc:p>
24    </doc:section>
25   </doc:section>
26   <doc:section>
27    <doc:title>Mixed namespace support</doc:title>
28    <doc:p>A foreign namespace SMC construct to import textual content:
29      (<imp:import href="imp.xml"  select="//*/text/text()"/>), a FOC XLink
30      attribute for <doc:em xl:type="simple" xl:href="http://www.cs.kent.ac.
            uk">links</doc:em>
31      and an SMC subtree that allows  adaptation sensitive content:
32    </doc:p>
33    <alt:alt>
34     <alt:case test="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe/Terms#deviceType=mobile">
35      <doc:p>This is a mobile</doc:p>
36     </alt:case>
37     <alt:case>
38      <doc:p>This is NOT a mobile</doc:p>
39     </alt:case>
40    </alt:alt>
41   </doc:section>
42  </doc:document>
```

Listing F.1: `document.xml`

```
1  <root>
2    <text>Text node 1</text>
3    <text>Text node 2</text>
4  </root>
```

Listing F.2: `imp.xml`

```
1  <collection
2        xmlns:doc="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
             document">
3   <doc:authors id="MP_DHS">
4    <doc:author first="M" last="Ped"
5        mail="mp49@kent.ac.uk"/>
6     <doc:author first="D" last="Shr"
7        mail="dhs@kent.ac.uk"/>
8   </doc:authors>
9  </collection>
```

Listing F.3: `authors.xml`

F.2 XMLPipe specific semantics

XMLPipe specific semantics are predefined adaptation terms and transformations that are used by the XMLPipe implementation. For the purposes of this case study, the necessary XMLPipe specific semantics consist of four adaptation terms and two atomic transformations. The adaptation terms are used to represent the set of natively supported languages, the device type and the maximum image dimensions. The atomic transformations are the static and dynamic versions of an XSL-T transformer. The latter is necessary for using XSL-T stylesheets within dynamic transformation pipelines.

F.2.1 Top level binding

The principal RDDL-based location mechanism is used for the adaptation term semantics, and it allows the semantics location using the namespace URI of the adaptation terms. Listing F.4 illustrates the RDDL link that points to the XMLPipe specific term semantics.

```
1 <rddl:resource xlink:type="simple"
2   xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
      XMLPipe#termSemantics"
3   xlink:href="TermInfo.xml"/>
```

Listing F.4: RDDL link to XML specific semantics

There is no binding mechanism for the XMLPipe specific atomic transformations, because they must be declared in each semantics document, according to the pilot implementation semantics representation.

F.2.2 Adaptation terms

Listing F.5 illustrates the semantics of the XMLPipe specific adaptation terms. The term `XMLPipeURI/Terms:supported` is used to specify the natively supported languages and representations. Its values are sets of strings, its default value is an empty set and its default resolution expression is a union of the conflicting sets. `XMLPipeURI/Terms:deviceType` is a *String* term that is used to specify the type of a device. Its default value is "desktop", because desktop computers can be considered as the most common device for accessing Web resources. `XMLPipeURI/Terms:maxImageX` and `XMLPipeURI/Terms:maxImageY` specify the maximum size of displayed images and they must both be associated to numeric values. The default image size of "800x600" pixels represents the accustomed window size that most Web pages are optimised for.

```
1  <terms ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/XMLPipe
       /Terms">
2    <term name="supported" type="SetOfStrings">
3      <default/>
4      <resolution>
5        <union>
6          <termVal/>
7          <prevTermVal/>
8        </union>
9      </resolution>
10   </term>
11
12   <term name="deviceType" type="String">
13     <default>desktop</default>
14   </term>
15
16   <term name="maxImageX" type="Numeric">
17     <default>800</default>
18   </term>
19   <term name="maxImageY" type="Numeric">
20     <default>600</default>
21   </term>
22
23 </terms>
```

Listing F.5: XMLPipe specific adaptation terms

F.2.3 Atomic transformations

The two XMLPipe specific atomic transformations allow the use of XSL-T stylesheets for both static and dynamic transformation specifications. Any processing semantics specification can use the built-in XSL-T transformer by incorporating the atomic transformation declarations illustrated in Listing F.6. Both atomic transformations use the same Java class, but the former includes the attribute `dynamic="true"`, which denotes that its external information can be dynamically generated.

```
1  <transformers>
2    <transformer name="XMLPipe_DXSLT"
3      implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Transformers.
           XSLTTransformer"
4      implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           Transformers/XSLT.class"
5      dynamic="true"/>
6
7    <transformer name="XMLPipe_XSLT"
8      implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Transformers.
           XSLTTransformer"
9      implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           Transformers/XSLT.class"
10     dynamic="false"/>
11   </transformers>
```

Listing F.6: XMLPipe specific atomic transformations

F.3 The composite adaptation profiles

Listing F.7 illustrates a single processing semantics document that contains all composite adaptation profiles used in the case study. The mobile profile includes two adaptation statements and also imports the two profiles illustrated in Listings F.8 and F.9. The former illustrates the default device adaptation profile, and the latter illustrates the necessary additional statements for a WBMP software upgrade.

```
1  <config name="CaseStudyProfiles"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3    <profiles>
4      <profile name="aDesktop">
5        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
             XMLPipe/Terms"
6                name="deviceType">desktop</statm>
7        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
             XMLPipe/Terms"
8                name="supported">
9          <item>http://www.w3.org/1999/xhtml</item>
10         <item>http://www.isi.edu/in-notes/iana/assignments/media-types/
               image/gif</item>
11         <item>http://www.isi.edu/in-notes/iana/assignments/media-types/
               image/jpeg</item>
12       </statm>
13     </profile>
14
15     <profile name="aMobile">
16       <include
17         ref="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/profiles/
               mobileDefault.xml"
```

```
18          name="DefaultMobileProfile"/>
19        <include
20          ref="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/profiles/
                mobileUpdate.xml"
21          name="SoftwareUpdateMobileProfile"/>
22        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
23          name="maxImageX">96</statm>
24        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
25          name="maxImageY">100</statm>
26      </profile>
27
28      <profile name="XSLFOPrinter">
29        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
30          name="deviceType">printer</statm>
31        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
32          name="supported">
33          <item>http://www.w3.org/1999/XSL/Format</item>
34        </statm>
35      </profile>
36    </profiles>
37  </config>
```

Listing F.7: All case study composite profiles

```
1   <config name="DefaultMobileProfile"
2           xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe">
3     <profiles>
4       <profile name="aMobile">
5         <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
6           name="deviceType">mobile</statm>
7         <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
8               name="supported">
9           <item>http://www.wapforum.org/DTD/wml_1.1.xml</item>
10        </statm>
11        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
12          name="maxImageX">150</statm>
13        <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
14          name="maxImageY">150</statm>
15      </profile>
16    </profiles>
17  </config>
```

Listing F.8: `mobileDefault.xml`

```
1   <config name="SoftwareUpdateMobileProfile"
2           xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe">
3     <profiles>
4       <profile name="aMobile">
5         <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
6               name="supported">
```

```
7           <item>http://www.isi.edu/in-notes/iana/assignments/media-types/
                image/vnd.wap.wbmp</item>
8         </statm>
9       </profile>
10    </profiles>
11  </config>
```

Listing F.9: `mobileUpdate.xml`

F.4 L_{imp} language

The L_{imp} language introduces a single handled construct that imports external document portions, according to an XPath expression. The processing semantics of L_{imp} consist of its handled construct information, an atomic validation and an adaptation requirements independent transformation pipeline.

F.4.1 Top level binding

The binding of L_{imp} processing semantics uses the principal RDDL-based location mechanism, because it is a case study specific language and we have control over the associated URI. The Web page that corresponds to
`http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/import`
is an XHTML document that contains the three RDDL links illustrated in Listing F.10. The first, second and third RDDL links point to XML files that contain the handled construct information, the validation semantics and the transformation semantics, respectively.

```
1
2  <rddl:resource xlink:type="simple"
3      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#intModelSemantics"
4      xlink:href="ImpHCInfo.xml"/>
5
6  <rddl:resource xlink:type="simple"
7      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#valSemantics"
8      xlink:href="ImpValSem.xml"/>
9
10 <rddl:resource xlink:type="simple"
11     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#transSemantics"
12     xlink:href="ImpTransSem.xml"/>
```

Listing F.10: RDDL links to the L_{imp} processing semantics

F.4.2 Handled constructs

L_{imp} contains a single element, which is an SMC handled construct. Its handled construct information is illustrated in Listing F.11.

```
1  <config name="ImpHC"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3    <constructs ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
         import">
```

```
4       <hc name="import" class="smc" node="el"/>
5     </constructs>
6   </config>
```

Listing F.11: L_{imp} handled construct information: `ImpHCInfo.xml`

F.4.3 Validation semantics

The validation semantics of L_{imp} consist of a schema specification that validates the `imp:import` element and an atomic validation that associates the language namespace with that schema. Listing F.12 illustrates the common XML Schema validator declaration and the atomic validation declaration. Listing F.13 illustrates the referenced XML Schema specification, which defines the syntax of the `imp:import` element.

```
1  <config name="ImpVal"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3
4    <validators>
5      <validator
6            name="XMLPipe_Validators_XMLSchema"
7            implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Validators.
                 XMLSchema"
8            implSource="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe
                 /XMLPipe/Validators/XMLSchema/XMLSchema.class"/>
9    </validators>
10
11   <validation
12       ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/import">
13     <atomVal ref="XMLPipe_Validators_XMLSchema"
14               src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                     import/xmlpipe/imp.xsd"/>
15   </validation>
16 </config>
```

Listing F.12: L_{imp} validation semantics: `ImpValSem.xml`

```
1  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2              xmlns:ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                   import"
3              targetNamespace="http://www.cs.kent.ac.uk/projects/XMLhandling
                   /XPEx/import"
4    elementFormDefault="qualified">
5    <xsd:element name="import">
6      <xsd:complexType>
7        <xsd:attribute name="ref" type="xsd:anyURI"
8                       use="required"/>
9        <xsd:attribute name="select"
10                      type="xsd:string"/>
11     </xsd:complexType>
12   </xsd:element>
13 </xsd:schema>
```

Listing F.13: L_{imp} validation semantics: schema specification

F.4.4 Transformation semantics

L_{imp} transformation semantics consist of an XSL-T stylesheet and handler declaration, which associates the `imp:import` handled construct to a dynamic transformation

pipeline. Listing F.14 illustrates the declaration of the handler and the two built-in XSL-T transformers (since they must always be declared prior to their use). Listing F.15 illustrates the XSL-T stylesheet used by the `option` atomic transformation. It generates a stylesheet that statically contains the imported content XPath expression, which is specified by the `select` attribute of the `imp:import` element.

```
<config name="ImpVal"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">

  <transformers>
    <transformer name="XMLPipe_DXSLT"
                 implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                     Transformers.XSLTTransformer"
                 implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                     XMLPipe/Transformers/XSLT.class"
                 dynamic="true"/>

    <transformer name="XMLPipe_XSLT"
                 implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                     Transformers.XSLTTransformer"
                 implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                     XMLPipe/Transformers/XSLT.class"
                 dynamic="false"/>
  </transformers>

  <handler name="Generic Import Handler">
    <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/import"
        >
      <hcRef name="import"/>
    </hcList>
    <pipe>
      <dynamic>
        <normal/>
        <option>
          <transform ref="XMLPipe_XSLT"
                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/
                         XPEx/importing/XPEx/import/xmlpipe/import.xsl"/>
        </option>
        <transform ref="XMLPipe_DXSLT"
                   src="XMLPipe:dynamic"/>
      </dynamic>
    </pipe>
  </handler>
</config>
```

Listing F.14: L_{imp} transformation semantics: `ImpTransSem.xml`

```
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
                xmlns:n = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XPEx/import"
                xmlns:xslout="this value is not used"
                version = "1.0">

  <!-- Alias needed when generating stylesheets -->
  <xsl:namespace-alias stylesheet-prefix="xslout" result-prefix="xsl"/>

  <xsl:template match="n:import">
    <xslout:stylesheet version="1.0">
      <xslout:template match="n:import">
```

```
12            <xslout:copy-of select="document(@href){@select}"/>
13          </xslout:template>
14      </xslout:stylesheet>
15    </xsl:template>
16
17 </xsl:stylesheet>
```

Listing F.15: L_{imp} transformation semantics: XSL-T stylesheet specification `import.xsl`

F.5 L_{alt} language language

Language L_{alt} allows the introduction of adaptation requirements dependent content. It introduces the `alt:alt` element that can contain a sequence of `alt:case` elements, which can be associated with a set of adequacy expressions. `alt:alt` semantics is to substitute itself with the contents of the optimal `alt:case` element, according to the input adaptation profile.

F.5.1 Top level binding

In a similar manner to L_{alt}, the binding of L_{alt} processing semantics uses the principal RDDL-based location mechanism, because it is a case study specific language and we have control over the associated URI. The Web page that corresponds to
`http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt`
is an XHTML document that contains the three RDDL links illustrated in Listing F.16. The first, second and third RDDL links point to XML files that contain the handled construct information, the validation semantics and the transformation semantics, respectively.

```
1
2 <rddl:resource xlink:type="simple"
3     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
          XMLPipe#intModelSemantics"
4     xlink:href="AltHCInfo.xml"/>
5
6 <rddl:resource xlink:type="simple"
7     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
          XMLPipe#valSemantics"
8     xlink:href="AltValSem.xml"/>
9
10 <rddl:resource xlink:type="simple"
11     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
          XMLPipe#transSemantics"
12     xlink:href="AltTransSem.xml"/>
```

Listing F.16: RDDL links to the L_{imp} processing semantics

F.5.2 Handled constructs

L_{alt} contains a single *SMC* handled construct. Its handled construct information is illustrated in Listing F.17.

```
1 <config name="AltHC"
2         xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe">
```

```
3   <constructs ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt">
4     <hc name="alt" class="smc" node="el"/>
5   </constructs>
6 </config>
```

Listing F.17: L_{alt} handled construct information: `AltHCInfo.xml`

F.5.3 Validation semantics

The validation semantics of L_{alt} consist of a schema specification that validates the `alt:alt` rooted subtrees and an atomic validation that associates the language namespace with that schema. Listing F.18 illustrates the common XML Schema validator declaration and the atomic validation declaration. Listing F.19 illustrates the referenced XML Schema specification, which defines the syntax of the L_{alt} constructs.

```
1 <config name="AltVal"
2         xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe">
3
4   <validators>
5     <validator
6           name="XMLPipe_Validators_XMLSchema"
7           implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Validators.
                XMLSchema"
8           implSource="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe
                /XMLPipe/Validators/XMLSchema/XMLSchema.class"/>
9   </validators>
10
11   <validation
12     ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt">
13     <atomVal ref="XMLPipe_Validators_XMLSchema"
14              src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt/
                    xmlpipe/alt.xsd"/>
15   </validation>
16
17 </config>
```

Listing F.18: L_{alt} validation semantics: `AltValSem.xml`

```
1 <?xml version="1.0"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3             xmlns:ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                  alt"
4             targetNamespace="http://www.cs.kent.ac.uk/projects/XMLhandling
                  /XPEx/alt"
5   elementFormDefault="qualified">
6
7   <xsd:element name="alt">
8     <xsd:complexType>
9       <xsd:sequence>
10        <xsd:element name="case" type="ns:Case" maxOccurs="unbounded"/>
11      </xsd:sequence>
12    </xsd:complexType>
13  </xsd:element>
14
15  <xsd:complexType name="Case" mixed="true">
16    <xsd:choice minOccurs="0" maxOccurs="unbounded">
17      <xsd:any processContents="lax"/>
18    </xsd:choice>
```

```
19      <xsd:attribute name="test" type="xsd:string"/>
20    </xsd:complexType>
21
22  </xsd:schema>
```

Listing F.19: L_{alt} validation semantics: schema specification

F.5.4 Transformation semantics

L_{alt} transformation semantics consist of a Java transformer and a handler declaration, which associates the `alt:alt` handled construct to a transformation pipeline that contains a single atomic transformation. Listing F.20 illustrates the declaration of the hander. Listing F.21 illustrates the Java implementation of the atomic transformation. It implements the atomic transformation interface and uses the adaptation requirements information to choose the optimal `alt:case` element.

```
1  <config name="AltVal"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3
4    <transformers>
5      <transformer name="XMLPipe_XPEx_ALT"
6                   implClass="uk.ac.kent.cs.projects.XMLHandling.XPEx.
                        Transformers.AltHandler"
7                   implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                        XPEx/alt/xmlpipe/AltHandler.class"
8                   dynamic="false"/>
9    </transformers>
10
11
12   <handler name="Generic_Alt_Handler">
13     <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt">
14       <hcRef name="alt"/>
15     </hcList>
16     <pipe>
17       <transform ref="XMLPipe_XPEx_ALT"
18                  src=""/>
19     </pipe>
20   </handler>
21 </config>
```

Listing F.20: L_{alt} transformation semantics: `AltTransSem.xml`

```
1
2  package uk.ac.kent.cs.projects.XMLHandling.XPEx;
3
4  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.transform.*;
5  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.*;
6  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.terms.*;
7  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.parsing.DOMUtil;
8
9  import org.w3c.dom.*;
10
11
12 /** Checks the expressions in the alternatives and copies
13     the content of the first one that matches.
14 namespace: http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/alt
15 Handled Construct: alt
16 Cases: alt:case test="..."
```

```
17
18  where the syntax is: ns#name = string
19
20  */
21
22  public class AltHandler extends Transformer{
23
24      public static final String TEST_ATT = "test";
25
26      public AltHandler()
27      {
28          super(false, false);
29      }
30
31
32      public boolean transformCustom(ComponentInterface ci,
33                                     Node root,
34                                     boolean isAttr,
35                                     TransformationContext cnt)
36          throws XMLPipeTransformException,
37                 XMLPipeTransformTempException
38
39      {
40          //Get the alt element
41          Element alt =  DOMUtil.getFirstEl(root);
42
43          //Loop through the children
44          Element caseEl = DOMUtil.getFirstEl(alt);
45
46          while (null != caseEl) {
47              //Get the test attribute
48              String test = caseEl.getAttribute(TEST_ATT);
49
50              //If default, use this one
51              if (null == test || test.length() == 0) break;
52
53              //Get the term
54              test = test.trim();
55              int idx = test.indexOf('#');
56
57              if (idx <= 0) {
58                  throw new XMLPipeTransformException(ErrorHandler.ERROR,
59                                                      "Namespace expected in
                                                            : " + test,
60                                                      caseEl);
61              }
62
63              String ns = test.substring(0, idx);
64
65              // Get the name: only recognizing = as operator, for now
66              int idx2 = test.indexOf('=');
67              if (idx2 <= idx) {
68                  throw new XMLPipeTransformException(ErrorHandler.ERROR,
69                                                      "Expected operator =
                                                            in: " + test,
70                                                      caseEl);
71              }
72
73
74              String name = test.substring(idx + 1, idx2).trim();
```

```
75
76              //Get the value
77              String value = test.substring(idx2 + 1).trim();
78
79              //Check if the term exists
80
81              try {
82                  if (compareValue(ns, name, value, ci, cnt))
83                      break;
84              } catch (XMLPipeTransformTempException e) {
85                  throw new XMLPipeTransformException(e.getType(),
86                                                      e.getMessage(),
87                                                      caseEl);
88              }
89
90              caseEl = DOMUtil.getNextEl(caseEl);
91          }
92
93          if (null == caseEl) {
94                  throw new XMLPipeTransformException(ErrorHandler.ERROR,
95                                                      "No case is applicable
                                                          and there is no
                                                          default",
96                                                      alt);
97          }
98
99          NodeList ls = caseEl.getChildNodes();
100
101         for (int i = 0; i < ls.getLength(); ++i) {
102             Node n = caseEl.removeChild(ls.item(0));
103             root.insertBefore(n, alt);
104         }
105
106         root.removeChild(alt);
107
108         return true;
109
110     }
111
112
113     public boolean transformCustom(ComponentInterface ci,
114                                    Node root,
115                                    boolean isAttr,
116                                    TransformationContext cnt,
117                                    Node par)
118         throws XMLPipeTransformException,
119                XMLPipeTransformTempException
120     {
121         throw new XMLPipeTransformTempException(ErrorHandler.ERROR,
122             "Cannot call ALT handler with a parameter");
123     }
124
125     /** Compares the value of the ns:name term with the value
126         assuming that it is of the same type. If the term has
127         not been declared, it throws an exception. If it has not
128         been defined it returns false.
129
130     */
131     public boolean compareValue(String ns, String name,
132                                 String value,
```

```
133                                     ComponentInterface ci,
134                                     ContextInformation cnt)
135         throws XMLPipeTransformTempException
136     {
137         //Get the term
138
139         Term term = (Term)ci.getConfigTable().getTerm(Term.getQName(ns,
                name));
140
141         if (null == term)
142             throw new XMLPipeTransformTempException(ErrorHandler.ERROR,
143                                                     "Term: " + ns + ":" +
                                                        name + " has not
                                                        been declared");
144
145         //Check if there is any statement
146         Statement st = cnt.getStatement(term);
147
148         if (null == st) return false;
149
150         //Assume a stringvalue
151
152         if (!(st.getValue() instanceof StringValue)) {
153             throw new XMLPipeTransformTempException(ErrorHandler.ERROR,
154                                                     "Expected a
                                                        stringValue term")
                                                        ;
155         }
156
157         StringValue val = (StringValue)st.getValue();
158
159         ci.getErrorHandler().error(ErrorHandler.INFO,
160                 "ALT HANDLER: Comparing: '" + val.getValue() + "' to
                        '" + value + "'");
161
162         return val.getValue().equals(value);
163
164     }
165
166 }
```

Listing F.21: L_{alt} transformation semantics: Java atomic transformation implementation

F.6 L_{doc} language

The L_{doc} language introduces the necessary constructs for defining the layout of a document and contains constructs that introduce formated text and images. Its transformation semantics are responsible for creating an adequate transformation layout for the target device. Additionally, an image converter is used to convert the document images to the restricted WBMP representation, which is supported by the example mobile device. The image converter handler uses a language specific adaptation term, in order to avoid infinite transformation recursion. Consequently, L_{doc} processing semantics consists of its handled construct information, the language specific term declaration, an atomic validation and the adaptation requirements dependent transformation pipelines.

F.6.1 Top level binding

The binding of L_{doc} are similar to the previously introduced bindings, but they also include a link to the adaptation term semantics. The Web page that corresponds to `http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/doc` is an XHTML document that contains the four RDDL links illustrated in Listing F.22.

```
1
2  <rddl:resource xlink:type="simple"
3      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#intModelSemantics"
4      xlink:href="DocHCInfo.xml"/>
5
6  <rddl:resource xlink:type="simple"
7      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#termSemantics"
8      xlink:href="DocTermSem.xml"/>
9
10 <rddl:resource xlink:type="simple"
11     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#valSemantics"
12     xlink:href="DocValSem.xml"/>
13
14 <rddl:resource xlink:type="simple"
15     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#transSemantics"
16     xlink:href="DocTransSem.xml"/>
```

Listing F.22: RDDL links to the L_{doc} processing semantics

F.6.2 Handled constructs

L_{doc} contains four *COC* handled constructs. Its handled construct information is illustrated in Listing F.23.

```
1  <config name="DocHC"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3    <constructs ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
         document">
4      <hc name="document" class="coc" node="el"/>
5      <hc name="em" class="coc" node="el"/>
6      <hc name="img" class="coc" node="el"/>
7      <hc name="p" class="coc" node="el"/>
8    </constructs>
9  </config>
```

Listing F.23: L_{doc} handled construct information: `DocHCInfo.xml`

F.6.3 Validation semantics

The validation semantics of L_{doc} are similar to the previously introduced validation semantics, and they consist of a schema specification and an atomic validation that associates the language namespace with that schema. Listing F.24 illustrates the common XML Schema validator declaration and the atomic validation declaration. Listing F.25 illustrates the referenced XML Schema specification, which defines the syntax of all L_{doc} constructs.

```
<config name="DocVal"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">

  <validators>
    <validator
         name="XMLPipe_Validators_XMLSchema"
         implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Validators.
             XMLSchema"
         implSource="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe
             /XMLPipe/Validators/XMLSchema/XMLSchema.class"/>
  </validators>

  <!-- The atomic validations -->
  <validation
     ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/document">
    <atomVal ref="XMLPipe_Validators_XMLSchema"
             src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                 document/xmlpipe/doc.xsd"/>
  </validation>
</config>
```

Listing F.24: L_{doc} validation semantics: `DocValSem.xml`

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                document"
            targetNamespace="http://www.cs.kent.ac.uk/projects/XMLhandling
                /XPEx/document"
            elementFormDefault="qualified">

  <xsd:element name="document">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="ns:title"/>
        <xsd:element ref="ns:authors"/>
        <xsd:element name="section" type="ns:sectionType" minOccurs="0"
            maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="title" type="xsd:string"/>

  <xsd:element name="authors">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="author" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="first" type="xsd:string" use="required"/>
            <xsd:attribute name="last" type="xsd:string" use="required"/>
            <xsd:attribute name="mail" type="xsd:anyURI" use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="sectionType">
```

```
35      <xsd:sequence>
36        <xsd:element ref="ns:title"/>
37        <xsd:element ref="ns:p" minOccurs="0" maxOccurs="unbounded"/>
38        <xsd:element name="section" type="ns:sectionType" minOccurs="0"
39        maxOccurs="unbounded"/>
40      </xsd:sequence>
41    </xsd:complexType>
42
43    <xsd:element name="p" type="ns:flowType"/>
44
45    <xsd:group name="flowElement">
46      <xsd:choice>
47        <xsd:element ref="ns:em"/>
48        <xsd:element ref="ns:img"/>
49      </xsd:choice>
50    </xsd:group>
51
52    <xsd:complexType name="flowType" mixed="true">
53      <xsd:choice minOccurs="0" maxOccurs="unbounded">
54         <xsd:group ref="ns:flowElement"/>
55         <xsd:any processContents="lax"/>
56      </xsd:choice>
57    </xsd:complexType>
58
59    <xsd:element name="em" type="ns:flowType"/>
60
61    <xsd:element name="img">
62      <xsd:complexType>
63        <xsd:attribute name="href" type="xsd:anyURI" use="required"/>
64      </xsd:complexType>
65    </xsd:element>
66
67  </xsd:schema>
```

Listing F.25: L_{doc} validation semantics: schema specification

F.6.4 Langage specific term

The language specific term `doNotRecurse` is a boolean adaptation term and controls the recursive execution of the image converter. After a `doc:img` construct has been processed, the image converter pipeline sets the value of `doNotRecurse` to *true*, in order to avoid the re-execution of the converter. Listing F.26 illustrates the `doNotRecurse` term's definition.

```
1  <config name="DocTerm"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3
4    <terms ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/document">
5      <term name="doNotRecurse" type="Boolean">
6        <default>false</default>
7      </term>
8    </terms>
9  </config>
```

Listing F.26: `doNotRecurse` adaptation term semantics: `DocTermSem.xml`

F.6.5 Transformation semantics

L_{doc} transformation semantics consist of four handlers that use four XSL-T stylesheets and a Java-based image converter. Listing F.27 illustrates the declaration of the XSL-T transformers, the image converter transformer and the four handlers. Listings F.28, F.29, F.31 illustrate the XSL-T stylesheets that map the constructs of L_{doc} to their desktop, mobile and XSL-FO printer interpretations, respectively. Listing F.30 illustrates the Java implementation of the WBMP image converter. Finally, Listing F.32 illustrates a namespace removal stylesheet, which is used in the mobile pipelines, since WML 1.1 mobiles do no necessarily support XML namespaces.

```
1  <config name="ImpVal"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3
4    <transformers>
5      <transformer name="XMLPipe_DXSLT"
6                   implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                        Transformers.XSLTTransformer"
7                   implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                        XMLPipe/Transformers/XSLT.class"
8                   dynamic="true"/>
9
10     <transformer name="XMLPipe_XSLT"
11                  implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                        Transformers.XSLTTransformer"
12                  implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                        XMLPipe/Transformers/XSLT.class"
13                  dynamic="false"/>
14   </transformers>
15
16
17   <transformers>
18     <transformer name="XMLPipe_XPEx_DOC_imageConverter"
19                  implClass="uk.ac.kent.cs.projects.XMLHandling.XPEx.
                        WBMPConverter"
20                  implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                        XPEx/document/xmlpipe/WBMPConverter.jar"
21                  dynamic="false"/>
22   </transformers>
23
24   <handler name="XHTML_doc_handler">
25       <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
             document">
26         <hcRef name="document"/>
27         <hcRef name="em"/>
28         <hcRef name="img"/>
29         <hcRef name="p"/>
30       </hcList>
31       <applicability>
32         <contains>
33           <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                 XMLPipe/XMLPipe/Terms"
34                    name="supported"/>
35           <val type="String">http://www.w3.org/1999/xhtml</val>
36
37         </contains>
38       </applicability>
39       <adequacy>
```

```
40          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
41                  name="supported">
42              <contains>
43                  <termVal/>
44                  <val type="String">http://www.w3.org/1999/xhtml</val>
45              </contains>
46          </expr>
47          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
48                  name="deviceType">
49              <equals>
50                  <termVal/>
51                  <val>desktop</val>
52              </equals>
53          </expr>
54        </adequacy>
55        <pipe>
56          <transform ref="XMLPipe_XSLT"
57                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /document/xmlpipe/doc.xsl"/>
58        </pipe>
59    </handler>
60
61    <handler name="WML_mobile_doc_handler">
62        <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
              document">
63          <hcRef name="document"/>
64          <hcRef name="em"/>
65          <hcRef name="img"/>
66          <hcRef name="p"/>
67        </hcList>
68        <applicability>
69          <contains>
70              <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XMLPipe/XMLPipe/Terms"
71                       name="supported"/>
72              <val type="String">http://www.wapforum.org/DTD/wml_1.1.xml</val
                    >
73          </contains>
74        </applicability>
75        <adequacy>
76          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
77                  name="supported">
78              <contains>
79                  <termVal/>
80                  <val type="String">http://www.wapforum.org/DTD/wml_1.1.xml</
                        val>
81              </contains>
82          </expr>
83          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
84                  name="deviceType">
85              <equals>
86                  <termVal/>
87                  <val>mobile</val>
88              </equals>
89          </expr>
90        </adequacy>
```

```
91        <pipe>
92          <transform ref="XMLPipe_XSLT"
93                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /document/xmlpipe/mobile.xsl"/>
94          <transform ref="XMLPipe_XSLT"
95                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /document/xmlpipe/removeNamespaces.xsl"/>
96        </pipe>
97    </handler>
98
99    <handler name="WBMP_image_convertion">
100     <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
            document">
101       <hcRef name="img"/>
102     </hcList>
103     <applicability>
104       <and>
105         <and>
106           <not>
107             <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XPEx/document"
108                      name="doNotRecurse"/>
109           </not>
110           <contains>
111             <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XMLPipe/XMLPipe/Terms"
112                      name="supports"/>
113             <val type="String">http://www.isi.edu/in-notes/iana/
                    assignments/media-types/image/vnd.wap.wbmp</val>
114           </contains>
115         </and>
116         <not>
117           <contains>
118             <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XMLPipe/XMLPipe/Terms"
119                      name="supports"/>
120             <val type="String">http://www.isi.edu/in-notes/iana/
                    assignments/media-types/image/jpeg</val>
121           </contains>
122         </not>
123       </and>
124     </applicability>
125     <adequacy>
126       <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
127             name="supported">
128           <contains>
129             <termVal/>
130             <val type="String">http://www.isi.edu/in-notes/iana/
                    assignments/media-types/image/vnd.wap.wbmp</val>
131           </contains>
132         </expr>
133         <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
              XMLPipe/Terms"
134             name="supported">
135           <not>
136             <contains>
137               <termVal/>
138               <val type="String">http://www.isi.edu/in-notes/iana/
                      assignments/media-types/image/jpeg</val>
```

```
139                 </contains>
140               </not>
141             </expr>
142             <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                    document"
143                   name="doNotRecurse">
144                 <not>
145                   <termVal/>
146                 </not>
147             </expr>
148       </adequacy>
149       <context>
150         <statm ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                document"
151                name="doNotRecurse">true</statm>
152       </context>
153       <pipe>
154           <transform ref="XMLPipe_XPEx_DOC_imageConverter"
155                      src=""/>
156       </pipe>
157     </handler>
158
159     <handler name="XSL-FO_printer_doc_handler">
160         <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                document">
161           <hcRef name="document"/>
162           <hcRef name="em"/>
163           <hcRef name="p"/>
164           <hcRef name="img"/>
165         </hcList>
166         <applicability>
167           <contains>
168             <termVal ns="http://www.cs.kent.ac.uk/projects/XMLhandling/
                    XMLPipe/XMLPipe/Terms"
169                      name="supported"/>
170             <val type="String">http://www.w3.org/1999/XSL/Format</val>
171           </contains>
172         </applicability>
173         <adequacy>
174          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
175                 name="supported">
176             <contains>
177               <termVal/>
178               <val type="String">http://www.w3.org/1999/XSL/Format</val>
179             </contains>
180          </expr>
181          <expr   ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                XMLPipe/Terms"
182                 name="deviceType">
183             <equals>
184               <termVal/>
185               <val>printer</val>
186             </equals>
187          </expr>
188         </adequacy>
189         <pipe>
190           <transform ref="XMLPipe_XSLT"
191                      src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /document/xmlpipe/XSLFOprinter.xsl"/>
```

```
192       </pipe>
193   </handler>
194 </config>
```

Listing F.27: L_{doc} transformation semantics: `DocTransSem.xml`

```
1  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
2                  version = "1.0"
3                  xmlns = "http://www.w3.org/1999/xhtml"
4                  xmlns:n = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                       XPEx/document"
5                  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/
                       XMLhandling/XMLPipe/XMLPipe/Representation/Internal">
6
7  <xsl:strip-space elements="*"/>
8
9    <xsl:template match="n:document">
10     <html>
11       <head><title><xsl:value-of select="n:title"/></title></head>
12       <body>
13
14         <h1><xsl:value-of select="n:title"/></h1>
15
16         <!-- Authors information -->
17         <xsl:for-each select="n:authors/n:author">
18             <div>
19               <a href="mailto:{@mail}">
20                 <xsl:value-of select="@first"/>
21                 <xsl:text> </xsl:text>
22                 <xsl:value-of select="@last"/>
23               </a>
24             </div>
25         </xsl:for-each>
26
27         <xsl:apply-templates select="n:section"/>
28       </body>
29     </html>
30   </xsl:template>
31
32
33   <xsl:template match="n:section">
34     <xsl:variable name="level" select="count(ancestor-or-self::n:section)"
         />
35     <xsl:variable name="no">
36       <xsl:number  level="multiple" count="n:section" format="1.1.1"/>
37     </xsl:variable>
38     <xsl:element name="h{$level}">
39       <xsl:value-of select="$no"/><xsl:text> </xsl:text>
40       <xsl:value-of select="n:title"/>
41     </xsl:element>
42
43     <xsl:apply-templates select="n:p"/>
44
45     <xsl:apply-templates select="n:section"/>
46
47
48   </xsl:template>
49
50
51   <xsl:template match="n:p">
52     <p>
```

```
53        <xsl:apply-templates select="*|text()"/>
54      </p>
55    </xsl:template>
56
57    <xsl:template match="n:em">
58      <em><xsl:apply-templates select="@*|*|text()"/></em>
59    </xsl:template>
60
61    <xsl:template match="n:img">
62      <img src="{@href}"/>
63    </xsl:template>
64
65
66    <xsl:template match="n:sp">
67      <xsl:text> </xsl:text>
68    </xsl:template>
69
70    <xsl:template match="n:br">
71      <br/>
72    </xsl:template>
73
74    <!-- Copy any unknown content -->
75    <xsl:template match="*">
76      <xsl:copy>
77         <xsl:copy-of select="@*"/>
78         <xsl:apply-templates select="*|text()"/>
79      </xsl:copy>
80    </xsl:template>
81
82    <xsl:template match="@*">
83      <xsl:copy-of select="."/>
84    </xsl:template>
85
86  </xsl:stylesheet>
```

Listing F.28: L_{doc} transformation semantics: Desktop XSL-T stylesheet specification `doc.xsl`

```
1  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
2                  version = "1.0"
3                  xmlns:n = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                       XPEx/document"
4                  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/
                       XMLhandling/XMLPipe/XMLPipe/Representation/Internal">
5    <xsl:strip-space elements="*"/>
6
7    <xsl:template match="n:document">
8      <wml xpipe:sysId="http://www.wapforum.org/DTD/wml_1.1.xml"
9           xpipe:pubId="-//WAPFORUM//DTD WML 1.1//EN">
10       <!-- Generic WML template -->
11       <template>
12         <do type="reset" label="start">
13           <go href="#{generate-id(.)}"/>
14         </do>
15         <do type="prev" label="back">
16           <prev/>
17         </do>
18       </template>
19
20       <!-- The root card -->
```

```
21        <card id="{generate-id(.)}">
22          <xsl:attribute name="title"><xsl:value-of select="n:title"/>
23        </xsl:attribute>
24
25          <xsl:for-each select="n:authors/n:author">
26              <p>
27                <a href="mailto:{@mail}">
28                  <xsl:value-of select="@first"/>
29                  <xsl:text> </xsl:text>
30                  <xsl:value-of select="@last"/>
31                </a>
32              </p>
33          </xsl:for-each>
34
35        <xsl:call-template name="generateLocalIndex"/>
36        </card>
37
38        <xsl:apply-templates select="n:section"/>
39      </wml>
40    </xsl:template>
41
42    <xsl:template match="n:section">
43      <xsl:variable name="level" select="count(ancestor-or-self::n:section)"
            />
44      <xsl:variable name="no">
45        <xsl:number  level="multiple" count="n:section" format="1.1.1"/>
46      </xsl:variable>
47
48      <card id="{generate-id(.)}">
49        <p>
50          <xsl:value-of select="$no"/><xsl:text> </xsl:text>
51          <xsl:value-of select="n:title"/>
52        </p>
53
54        <xsl:call-template name="generateLocalIndex"/>
55      </card>
56
57      <xsl:apply-templates select="n:p"/>
58
59      <xsl:apply-templates select="n:section"/>
60
61
62    </xsl:template>
63
64
65    <xsl:template match="n:p">
66      <card id="{generate-id(.)}">
67        <xsl:if test="following-sibling::n:p">
68          <do type="accept" label="next">
69              <go href="#{generate-id(following-sibling::n:p[1])}"/>
70          </do>
71        </xsl:if>
72        <p>
73          <xsl:apply-templates select="*|text()"/>
74        </p>
75      </card>
76    </xsl:template>
77
78
79    <xsl:template match="n:em">
```

```
80      <xsl:text> *</xsl:text>
81      <xsl:apply-templates select="*|text()"/>
82      <xsl:text>* </xsl:text>
83    </xsl:template>
84
85
86    <xsl:template match="img">
87      <img src="{@href}" alt="something"/>
88    </xsl:template>
89
90    <xsl:template match="n:br">
91      <br/>
92    </xsl:template>
93
94    <xsl:template match="n:sp">
95      <xsl:text> </xsl:text>
96    </xsl:template>
97
98    <xsl:template match="n:br">
99      <br/>
100   </xsl:template>
101
102
103   <!-- Copy any unknown content -->
104   <xsl:template match="*">
105     <xsl:copy>
106       <xsl:copy-of select="@*"/>
107       <xsl:apply-templates select="*|text()"/>
108     </xsl:copy>
109   </xsl:template>
110
111   <xsl:template name="generateLocalIndex">
112     <xsl:for-each select="n:p">
113       <p>
114       <a href="#{generate-id(.)}">
115           Text section
116       </a>
117     </p>
118     </xsl:for-each>
119     <xsl:for-each select="n:section">
120       <p>
121       <a href="#{generate-id(.)}">
122           <xsl:number count="n:section"/>
123           <xsl:text> </xsl:text>
124           <xsl:value-of select="n:title"/>
125       </a>
126     </p>
127     </xsl:for-each>
128  </xsl:template>
129 </xsl:stylesheet>
```

Listing F.29: L_{doc} transformation semantics: Mobile XSL-T stylesheet specification `mobile.xsl`

```
1  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.transform.*;
2  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.*;
3  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.terms.*;
4  import  uk.ac.kent.cs.projects.XMLHandling.XMLPipe.parsing.DOMUtil;
5
6  import com.unwiredtec.wmcreator.SimpleImageFile;
```

```
7  import com.unwiredtec.wmcreator.WBMPfile;
8  import com.unwiredtec.wmcreator.WBMPFilter;
9  import com.unwiredtec.wmcreator.WBMPConstants;
10 import com.unwiredtec.wmcreator.WBMPDimension;
11
12 import java.io.*;
13
14 import java.net.*;
15
16 import org.w3c.dom.*;
17
18 /** A simple converter from common image files
19 to WBMP. It uses the WBMPCreator libary.
20 */
21
22
23 public class WBMPConverter extends Transformer {
24
25     static final String CAP_NS = "example.org/terms/capabilities";
26     static final String CAP_MAXX = "maxImageX";
27     static final String CAP_MAXY = "maxImageY";
28
29     public WBMPConverter()
30     {
31         super(false, false);
32     }
33
34     public boolean transformCustom(ComponentInterface ci,
35                                    Node root,
36                                    boolean isAttr,
37                                    TransformationContext cnt)
38         throws XMLPipeTransformException,
39                XMLPipeTransformTempException
40
41     {
42
43
44         //Get the handled construct
45         Element img =  DOMUtil.getFirstEl(root);
46
47         //Get the referenced image
48         String imgURI = absoluteInputURI(img.getAttribute("href"), img).
               toString();
49
50         //Default maximum image dimensions
51         int maxX = getIntValue(CAP_NS, CAP_MAXX,
52                                ci, cnt);
53         if (-1 == maxX) maxX = 100;
54
55
56         int maxY = getIntValue(CAP_NS, CAP_MAXY,
57                                ci, cnt);
58         if (-1 == maxY) maxY = 100;
59
60
61         //Check if there is any maximum image specification
62         //in the context
63         ErrorHandler err = ci.getErrorHandler();
64         try {
65
```

```
66              err.error(ErrorHandler.INFO,
67                        "Converting image file: " + imgURI);
68
69              SimpleImageFile sif = new SimpleImageFile(new URL(imgURI));
70
71              WBMPFilter wff=new WBMPFilter();
72
73              WBMPfile w = new WBMPfile(sif, -1, -1, -1);
74
75              err.error(ErrorHandler.INFO,
76                        "Maximum image size: (" + maxX + ", " + maxY + ")");
77
78              //Resize if bigger than the screen size
79              if (w.getWBMPLength() > maxX || w.getWBMPWidth() > maxX) {
80                  //Compute the ratio to multiply
81                  double ratio = computeRatio(w.getWBMPLength(), w.
                        getWBMPWidth(), maxX, maxY);
82
83                  err.error(ErrorHandler.INFO,
84                        "Resizing image, ratio: " + ratio);
85
86                  w.resize((int)(w.getWBMPLength() * ratio), (int)(w.
                        getWBMPWidth() * ratio));
87              }
88
89              //Create an output file
90
91              String relURI = "generatedImg/" +
92                  imgURI.hashCode() + ".wbmp";
93
94
95              w.saveWBMPFile("WBMP", absoluteOutputURI(relURI).getPath());
96
97              //Simply change the href to point to the new file
98              img.setAttribute("href", relURI);
99
100             err.error(ErrorHandler.INFO,
101                       "Generated new image: " + relURI);
102
103             return true;
104
105
106         } catch (IllegalArgumentException e) {
107             throw new XMLPipeTransformException(ErrorHandler.ERROR,
108                                                 "Cannot construct absolute
                                                        from URI: " + imgURI
                                                         + ": " + e.getMessage
                                                        (),
109                                                 img);
110         } catch (MalformedURLException e) {
111             throw new XMLPipeTransformException(ErrorHandler.ERROR,
112                                                 "Cannot construct URL from
                                                        URI: " + imgURI + ":
                                                        "+ e.getMessage(),
113                                                 img);
114         } catch (NoClassDefFoundError e) {
115             throw new XMLPipeTransformTempException(ErrorHandler.ERROR,
```

```
116                                                     "The WBMP Creator
                                                            library doesnt
                                                            seem to be
                                                            accesible: " + e.
                                                            getMessage());
117
118         } catch (Exception e) {
119             throw new XMLPipeTransformException(ErrorHandler.ERROR,
120                                                 "Cannot tranform image: "
                                                        + e.getMessage(),
121                                                 img);
122         }
123
124     }
125
126     public boolean transformCustom(ComponentInterface ci,
127                                    Node root,
128                                    boolean isAttr,
129                                    TransformationContext cnt,
130                                    Node par)
131         throws XMLPipeTransformException,
132                XMLPipeTransformTempException
133
134
135     {
136         throw new XMLPipeTransformTempException(ErrorHandler.ERROR,
137             "Cannot call WBMPTransformer with a parameter");
138     }
139
140     /** Convenience methid to extract context statemeent
141         values. It returns -1 if there is no definition
142     */
143     public int getIntValue(String ns, String name,
144                            ComponentInterface ci,
145                            ContextInformation cnt)
146     {
147         //Get the term
148
149         Term term = (Term)ci.getConfigTable().getTerm(Term.getQName(ns,
                name));
150
151         if (null == term) return -1;
152
153         //Check if there is any statement
154         Statement st = cnt.getStatement(term);
155
156         if (null == st   ||
157             !(st.getValue() instanceof NumericValue)) {
158             return -1;
159         }
160
161         NumericValue val = (NumericValue)st.getValue();
162
163         return (int)val.getValue();
164
165     }
166
167     public double computeRatio(int x, int y, int maxX, int maxY)
168     {
169         //Which is the worst?
```

```
170         boolean xIsWorst = ((double)x / maxX > (double)y / maxY);
171
172         if (xIsWorst) {
173             return (double)maxX / x;
174         } else {
175             return (double)maxY / y;
176         }
177
178     }
179
180 }
```

Listing F.30: L_{doc} transformation semantics: WBMP image converter

```
1  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
2                  version = "1.0"
3                  xmlns = "http://www.w3.org/1999/XSL/Format"
4                  xmlns:n = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                       XPEx/document"
5                  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/
                       XMLhandling/XMLPipe/XMLPipe/Representation/Internal">
6
7  <xsl:strip-space elements="*"/>
8
9    <xsl:template match="n:document">
10     <root xmlns:fo="http://www.w3.org/1999/XSL/Format">
11       <layout-master-set>
12         <simple-page-master master-name="my-page">
13           <region-body margin="1in"/>
14           <region-before extent="1.5in"
15             padding="6pt 1in"
16             border-bottom="0.5pt silver solid"
17             display-align="after"/>
18         </simple-page-master>
19       </layout-master-set>
20
21       <page-sequence master-reference="my-page">
22 <static-content flow-name="xsl-region-before">
23   <block text-align="end">Page <page-number/></block>
24 </static-content>
25         <flow flow-name="xsl-region-body">
26
27           <!-- The tile -->
28           <block space-after="1em" font-size="25pt" text-align="center"><
                 xsl:value-of select="n:title"/></block>
29
30         <!-- Authors information -->
31         <xsl:for-each select="n:authors/n:author">
32           <block text-align="center">
33             <xsl:value-of select="@first"/>
34             <xsl:text> </xsl:text>
35             <xsl:value-of select="@last"/>
36             (<xsl:value-of select="@mail"/>)
37           </block>
38         </xsl:for-each>
39
40         <xsl:apply-templates select="n:section"/>
41       </flow>
42       </page-sequence>
43     </root>
44
```

```
45
46    </xsl:template>
47
48    <xsl:template match="n:section">
49      <xsl:variable name="level" select="count(ancestor-or-self::n:section)"
          />
50      <xsl:variable name="no">
51        <xsl:number  level="multiple" count="n:section" format="1.1.1"/>
52      </xsl:variable>
53      <block font-style="bold" font-size="18pt">
54        <xsl:value-of select="$no"/><xsl:text> </xsl:text>
55        <xsl:value-of select="n:title"/>
56      </block>
57
58      <xsl:apply-templates select="n:p"/>
59
60      <xsl:apply-templates select="n:section"/>
61
62    </xsl:template>
63
64
65
66    <xsl:template match="n:p">
67      <block text-align="justify">
68        <xsl:apply-templates select="*|text()"/>
69      </block>
70    </xsl:template>
71
72    <xsl:template match="n:em">
73      <inline font-style="italic"><xsl:apply-templates select="*|text()"/></
          inline>
74    </xsl:template>
75
76    <xsl:template match="n:img">
77      <external-graphic src="url({@href})"/>
78    </xsl:template>
79
80    <xsl:template match="n:sp">
81      <xsl:text> </xsl:text>
82    </xsl:template>
83
84    <xsl:template match="n:br">
85      <block/>
86    </xsl:template>
87
88
89    <!-- Copy any unknown content -->
90    <xsl:template match="*">
91      <xsl:copy>
92        <xsl:copy-of select="@*"/>
93        <xsl:apply-templates select="*|text()"/>
94      </xsl:copy>
95    </xsl:template>
96
97  </xsl:stylesheet>
```

Listing F.31: L_{doc} transformation semantics: XSL-FO printer XSL-T stylesheet specification `XSLFOPrinter.xsd`

```
1
```

```
2  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
3                  version = "1.0">
4
5  <xsl:strip-space elements="*"/>
6
7  <xsl:template match="*">
8    <xsl:element name="{local-name()}">
9      <xsl:for-each select="attribute::*">
10       <xsl:attribute name="{local-name(.)}">
11         <xsl:value-of select="."/>
12       </xsl:attribute>
13
14     </xsl:for-each>
15     <xsl:apply-templates select="*|text()"/>
16   </xsl:element>
17 </xsl:template>
18
19 </xsl:stylesheet>
```

Listing F.32: L_{doc} transformation semantics: Namespace declaration removal stylesheet `removeNamespaces.xsl`

F.7 L_{xl} language

The L_{xl} language represents the simple XLink links, and it consists of two *FOC* attributes. Its processing semantics consist of its handled construct information, an atomic validation and three adaptation requirement dependent transformation pipelines.

F.7.1 Top level binding

The binding of L_{xl} processing semantics uses a secondary location mechanism, because it is a W3C language and the corresponding Web page does not contain RDDL links to XMLPipe processing semantics. For the case study processing, the L_{xl} semantics were included within a local repository file, in a similar manner to all other semantics, since the pilot XMLPipe implementation only supports a local file secondary location mechanism.

F.7.2 Handled constructs

L_{xl} contains two *FOC* attributes. Its handled construct information is illustrated in Listing F.33.

```
1  <config name="XLHC"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3    <!-- Handled constructs declarations -->
4    <constructs ns="http://www.w3.org/1999/xlink/">
5      <hc name="href" class="foc" node="at"/>
6      <hc name="type" class="foc" node="at"/>
7    </constructs>
8  </config>
```

Listing F.33: L_{xl} handled construct information: `XLHCInfo.xml`

F.7.3 Validation semantics

The validation semantics of L_{xl} consist of a schema specification that validates the `xl:href` and `xl:type` attributes, within the predefined `foc` element, and an atomic validation that associates the language namespace with that schema. Listing F.34 illustrates the common XML Schema validator declaration and the atomic validation declaration. Listing F.35 illustrates the referenced XML Schema specification.

```
<config name="XLVal"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">

  <validators>
    <validator
          name="XMLPipe_Validators_XMLSchema"
          implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Validators.
              XMLSchema"
          implSource="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe
              /XMLPipe/Validators/XMLSchema/XMLSchema.class"/>
  </validators>

  <validation
      ns="http://www.w3.org/1999/xlink/">
    <atomVal ref="XMLPipe_Validators_XMLSchema"
             src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/xl/xl
                 .xsd"/>
  </validation>
</config>
```

Listing F.34: L_{xl} validation semantics: `XLValSem.xml`

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns="http://www.w3.org/1999/xlink/"
            targetNamespace="http://www.w3.org/1999/xlink/"
            elementFormDefault="qualified"
            xmlns:pipe="http://www.cs.kent.ac.uk/projects/XMLhandling/
                XMLPipe/XMLPipe">
  <xsd:attribute name="type" type="xsd:string"
                 fixed="simple"/>
  <xsd:attribute name="href" type="xsd:string"/>

  <xsd:element name="foc">
   <xsd:complexType>
     <xsd:attribute ref="ns:type" use="required"/>
     <xsd:attribute ref="ns:href" use="required"/>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing F.35: L_{xl} validation semantics: schema specification

F.7.4 Transformation semantics

L_{xl} transformation semantics consist of three handlers and three corresponding XSL-T stylesheets. Listing F.36 declares the three handlers that correspond to the XHTML, WML and non-interactive links interpretation. Listings F.37, F.38 and F.39 illustrate the corresponding XSL-T stylesheets.

```
<config name="XLTran"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">

  <transformers>
    <transformer name="XMLPipe_XSLT"
                 implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                     Transformers.XSLTTransformer"
                 implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                     XMLPipe/Transformers/XSLT.class"
                 dynamic="false"/>
  </transformers>

  <handler name="XLinks_for_XHTML">
    <hcList ns="http://www.w3.org/1999/xlink/">
      <hcRef name="href"/>
    </hcList>
    <adequacy>
       <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe/Terms"
             name="supports">
          <contains>
             <termVal/>
             <val>http://www.w3.org/1999/xhtml</val>
          </contains>
       </expr>
       <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe/Terms"
             name="supports">
          <not>
             <contains>
                <termVal/>
                <val>http://www.w3.org/1999/xlink/</val>
             </contains>
          </not>
       </expr>
    </adequacy>
    <pipe>
        <transform ref="XMLPipe_XSLT"
                   src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                       /XL/xlinkXHTML.xsl"/>
    </pipe>

  </handler>

  <handler name="XLinks_for_WML">
    <hcList ns="http://www.w3.org/1999/xlink/">
      <hcRef name="href"/>
    </hcList>
    <adequacy>
       <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe/Terms"
             name="supports">
          <contains>
             <termVal/>
             <val>http://www.wapforum.org/DTD/wml_1.1.xml</val>
          </contains>
       </expr>
       <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe/Terms"
```

```
53              name="supports">
54            <not>
55              <contains>
56                  <termVal/>
57                  <val>http://www.w3.org/1999/xlink/</val>
58              </contains>
59            </not>
60          </expr>
61      </adequacy>
62      <pipe>
63          <transform ref="XMLPipe_XSLT"
64                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /XL/xlinkWML.xsl"/>
65      </pipe>
66
67    <handler name="XLinks for printers">
68      <hcList ns="http://www.w3.org/1999/xlink/">
69        <hcRef name="href"/>
70      </hcList>
71      <adequacy>
72          <expr ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
                    XMLPipe/Terms"
73              name="deviceType">
74            <equals>
75                <termVal/>
76                <val>printer</val>
77            </equals>
78          </expr>
79      </adequacy>
80      <pipe>
81          <transform ref="XMLPipe_XSLT"
82                     src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx
                           /XL/xlinkNonInteractive.xsl"/>
83      </pipe>
84    </handler>
```

Listing F.36: L_{xl} transformation semantics: `XLTransSem.xml`

```
1
2  <xsl:stylesheet
3     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
4  version = "1.0"
5  xmlns = "http://www.w3.org/1999/xhtml"
6  xmlns:xlink = "http://www.w3.org/1999/xlink/"
7  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
       XMLPipe/Representation/Internal">
8
9    <xsl:strip-space elements="*"/>
10
11   <xsl:template match="*[@xlink:href]">
12     <a href="{@xlink:href}">
13       <xsl:copy>
14         <xsl:apply-templates select="@*|*|text()"/>
15       </xsl:copy>
16     </a>
17   </xsl:template>
18
19   <xsl:template match="@xlink:*"/>
20
21   <xsl:template match="@*">
22     <xsl:copy-of select="."/>
```

```
23    </xsl:template>
24
25    <xsl:template match="*">
26      <xsl:copy-of select="."/>
27    </xsl:template>
28
29  </xsl:stylesheet>
```

Listing F.37: L_{xl} transformation semantics: XHTML XSL-T stylesheet specification `xlinkXHTML.xsl`

```
1  <xsl:stylesheet
2    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
3  version = "1.0"
4  xmlns:xlink = "http://www.w3.org/1999/xlink/"
5  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
       XMLPipe/Representation/Internal">
6
7    <xsl:strip-space elements="*"/>
8
9    <xsl:template match="*[@xlink:href]">
10    <a href="{@xlink:href}">
11       <xsl:copy>
12         <xsl:apply-templates select="@*"/>
13         <xsl:copy-of select="*|text()"/>
14       </xsl:copy>
15    </a>
16   </xsl:template>
17
18   <xsl:template match="@xlink:*"/>
19
20   <xsl:template match="@*">
21     <xsl:copy-of select="."/>
22   </xsl:template>
23 </xsl:stylesheet>
```

Listing F.38: L_{xl} transformation semantics: Mobile XSL-T stylesheet specification `xlinkWML.xsl`

```
1
2  <xsl:stylesheet
3  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
4  version = "1.0"
5  xmlns = "http://www.w3.org/1999/xhtml"
6  xmlns:xlink = "http://www.w3.org/1999/xlink/"
7  xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
       XMLPipe/Representation/Internal">
8
9    <xsl:strip-space elements="*"/>
10
11   <xsl:template match="*[@xlink:href]">
12     <xsl:copy>
13       <xsl:apply-templates select="@*|*|text()"/>
14       (<xsl:value-of select="@xlink:href"/>)
15    </xsl:copy>
16   </xsl:template>
17
18   <xsl:template match="@xlink:*"/>
19
20   <xsl:template match="@*">
```

```
21      <xsl:copy-of select="."/>
22    </xsl:template>
23
24    <xsl:template match="*">
25      <xsl:copy>
26        <xsl:copy-of select="@*"/>
27        <xsl:apply-templates select="*|text()"/>
28      </xsl:copy>
29    </xsl:template>
30  </xsl:stylesheet>
```

Listing F.39: L_{xl} transformation semantics: Non interactive XSL-T stylesheet specification `xhtmlNonInteractive.xsl`

F.8 L_{cd} language

The L_{cd} language introduces a single *COC* construct that introduced information about a compact disk. The processing semantics of L_{cd} consist of its handled construct information, an atomic validation and an adaptation requirements independent transformation pipeline.

F.8.1 Top level binding

The binding of L_{cd} processing semantics uses the principal RDDL-based location mechanism, because it is a case study specific language and we have control over the associated URI. The Web page that corresponds to
`http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/cd`
is an XHTML document that contains the three RDDL links illustrated in Listing F.40. The first, second and third RDDL links point to XML files that contain the handled construct information, the validation semantics and the transformation semantics, respectively.

```
1
2  <rddl:resource xlink:type="simple"
3      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#intModelSemantics"
4      xlink:href="CDHCInfo.xml"/>
5
6  <rddl:resource xlink:type="simple"
7      xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#valSemantics"
8      xlink:href="CDValSem.xml"/>
9
10 <rddl:resource xlink:type="simple"
11     xlink:role="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
           XMLPipe#transSemantics"
12     xlink:href="CDTransSem.xml"/>
```

Listing F.40: RDDL links to the L_{cd} processing semantics

F.8.2 Handled constructs

L_{cd} contains a handled construct, is a *COC* handled construct. Its handled construct information is illustrated in Listing F.41.

```
<config name="CDHC"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">
  <constructs ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/cd">
    <hc name="cd" class="coc" node="el"/>
  </constructs>
</config>
```

Listing F.41: L_{cd} handled construct information: `CDHCInfo.xml`

F.8.3 Validation semantics

The validation semantics of L_{cd} consist of a schema specification and an atomic validation that associates the language namespace with that schema. Listing F.42 illustrates the common XML Schema validator declaration and the atomic validation declaration. Listing F.43 illustrates the referenced XML Schema specification.

```
<config name="CDVal"
        xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
            XMLPipe">

  <validators>
    <validator
         name="XMLPipe_Validators_XMLSchema"
         implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.Validators.
             XMLSchema"
         implSource="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe
             /XMLPipe/Validators/XMLSchema/XMLSchema.class"/>
  </validators>

  <validation
      ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/cd">
    <atomVal ref="XMLPipe_Validators_XMLSchema"
             src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                 import/xmlpipe/cd.xsd"/>
  </validation>
</config>
```

Listing F.42: L_{cd} validation semantics: `CDValSem.xml`

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                cd"
            targetNamespace="http://www.cs.kent.ac.uk/projects/XMLhandling
                /XPEx/cd"
            elementFormDefault="qualified">

  <xsd:element name="cd">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="artists">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="artist" type="xsd:string" maxOccurs="
                  unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
```

```
17          <xsd:element name="songs">
18            <xsd:complexType>
19               <xsd:sequence>
20                 <xsd:element name="song" type="xsd:string" maxOccurs="
                       unbounded"/>
21               </xsd:sequence>
22            </xsd:complexType>
23          </xsd:element>
24        </xsd:sequence>
25        <xsd:attribute name="title" type="xsd:string" use="required"/>
26        <xsd:attribute name="coverImg" type="xsd:anyURI"/>
27        <xsd:attribute name="uri" type="xsd:anyURI"/>
28      </xsd:complexType>
29    </xsd:element>
30  </xsd:schema>
```

Listing F.43: L_{cd} validation semantics: schema specification

F.8.4 Transformation semantics

L_{cd} transformation semantics consist of an XSL-T stylesheet and a handler declaration, which associates the `cd:cd` handled construct to a transformation that reuses the other case study languages. Listing F.44 illustrates the declaration of the handler and the built-in XSL-T transformer, and Listing F.45 illustrates the XSL-T stylesheet.

```
1  <config name="CDVal"
2          xmlns="http://www.cs.kent.ac.uk/projects/XMLhandling/XMLPipe/
               XMLPipe">
3
4    <transformers>
5
6      <transformer name="XMLPipe_XSLT"
7                   implClass="uk.ac.kent.cs.projects.XMLHandling.XMLPipe.
                        Transformers.XSLTTransformer"
8                   implSrc="http://www.cs.kent.ac.uk/projects/XMLhandling/
                        XMLPipe/Transformers/XSLT.class"
9                   dynamic="false"/>
10   </transformers>
11
12   <handler name="CD_Handler">
13     <hcList ns="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/cd">
14       <hcRef name="cd"/>
15     </hcList>
16     <pipe>
17       <transform ref="XMLPipe_XSLT"
18                  src="http://www.cs.kent.ac.uk/projects/XMLhandling/XPEx/
                        importing/XPEx/cd/xmlpipe/cd.xsl"/>
19     </pipe>
20   </handler>
21 </config>
```

Listing F.44: L_{cd} transformation semantics: `CDTransSem.xml`

```
1  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
2                  version = "1.0"
3                  xmlns = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                       XPEx/document"
4                  xmlns:x = "http://www.w3.org/1999/xlink/"
5                  xmlns:n = "http://www.cs.kent.ac.uk/projects/XMLhandling/
                       XPEx/cd"
```

```
6                   xmlns:xpipe = "http://www.cs.kent.ac.uk/projects/
                        XMLhandling/XMLPipe/XMLPipe/Representation/Internal">
7
8     <xsl:template match="n:cd">
9       <p>
10        <img href="{@coverImg}"/><sp/><sp/>
11          <em x:type="simple" x:href="{@uri}">
12            <xsl:value-of select="n:title/text()"/>
13          </em>
14      </p>
15      <p><em>Artists:</em><br/>
16        <xsl:for-each select="n:artists/n:artist/@name">
17          <sp/><xsl:value-of select="."/><br/>
18        </xsl:for-each>
19      </p>
20      <p><em>Songs:</em><br/>
21        <xsl:for-each select="n:songs/n:song">
22          <sp/><xsl:value-of select="position()"/>) <xsl:value-of select="."
                /><br/>
23        </xsl:for-each>
24      </p>
25
26    </xsl:template>
27
28  </xsl:stylesheet>
```

Listing F.45: L_{cd} transformation semantics: XSL-T stylesheet specification `cd.xsl`