Using W3C XML Schema for semantic validation of security policies

Dmitry Bragin, David Chadwick
University of Kent
[db91, dwe8]@kent.ac.uk

Abstract

Validation of semantics in data has always been a major issue in software development. Gradual adoption of XML as the main data exchange format has solved many of the interoperability problems. However, the issue of formally defining the semantics of XML data structures is still a major one. This paper focuses on the use of XML Schemas to validate the structure and semantics of XML data constructs used by existing applications, and the areas of semantic validation where XML Schemas can be used. The paper reviews the use of XML validation tools and then applies XML Schemas to the validation of PERMIS Security Policies. Through this exercise, current deficiencies in the XML Schema standards are highlighted and areas for future developments are presented.

1. Introduction

Processing text data in a programming language such as Java or C++ is an issue as these languages lack built-in tools for checking the syntax and semantics of the data. Structuring the text data as XML provides a standard syntax for the storage and transfer of the data, but validating the XML in a conventional programming language is worse still due to the large amount of verbose, difficult to maintain code that is required. The problems are evident even when all that is required is reading the content of an XML document into a simple object such as a Java Bean. Validating the incoming data is complicated as both structural and semantic validation has to be performed. The inherently recursive nature of XML parsing makes the process even more complicated. The W3C XML Schema Standard [1] (referred to as WXS in this paper1), first introduced almost seven years ago, tries to address part of the problems caused by the difficulty of semantically and syntactically validating XML data by providing a mechanism to decouple data validation from the application code, by placing it into a separate schema. Unlike other validation techniques such as DTDs, WXS allows some semantic validation to be moved to the schema, reducing the amount of application specific validation code that is required.

2. W3C XML Schema

2.1 Differences from DTDs and other schema languages

Document Type Definition (DTD) language is a format for specifying grammars of documents authored in SGML [17] or SGML-derived languages such as HTML [13] or XML [14]. Cheney [12] lists the following advantages of a DTD:

1. DTDs are simple, established, and widely adopted
2. DTD parsing is built-in to most XML parsers
3. DTD validation is easier to implement than for the other approaches

DTDs however have a number of disadvantages [2]:

1. Lack of modularity
2. Limited type system
3. Extremely limited semantic validation

Chu [2] has performed an extensive comparison of 6 XML Schema languages. This study highlights the following properties of WXS:

1. WXS is the most expressive schema language available
2. WXS contains the highest number of built-in datatypes
3. WXS contains the full set of features related to inheritance and uniqueness/referencing constraints

---

1 There is considerable ambiguity in using “XML Schema” to specifically refer to W3C XML Schema as there are a number of other XML Schema Languages available such as RELAX NG [29] and Schematron [28].
WXS is a powerful, but difficult to use schema language with extremely redundant syntax. Below we discuss each of these properties in greater detail.

2.1.1 Expressiveness. Chu [2] identifies WXS as “the most expressive schema language”. However this expressiveness mainly lies in the domains of data typing, structural validation and referencing. When it comes to semantic validation WXS is much less expressive than pattern-based languages such as Schematron [28]. WXS cannot express even most basic of semantic constraints such as co-occurrence (making the presence of an element mandatory based on the presence of some other element in the document tree).

2.1.2 Datatypes. Unlike DTDs, WXS places great emphasis on the data types of elements and attributes. The datatype specification is 101 pages long [3]. It defines a collection of 37 types along with a mechanism for the user to define custom types by extending or restricting previously defined types. In comparison, RELAX NG only contains two built-in data types – string and token (however, WXS data types can be also used). WXS distinguishes between complex and simple types. The latter can be further divided into primitive and derived datatypes, of which primitive datatypes can be loosely related to primitive data types such as integer, string and floating point number in programming languages like Java. An example of a derived datatype would be a non-zero positive integer. Complex types can be thought of as C structures encapsulating either primitive types or other structures. This advanced typing system is very similar to the one used in much earlier in the ASN.1 standard [4]. The ASN.1 type extension system also provides the ability for the user to define custom types by extension or restriction of a base type. One can argue whether this complex multilevel typing system (that can be considered much more complex than typing systems used in modern programming languages) is one of the barriers on the way to wider adoption of WXS.

2.1.3 Inheritance and referencing. WXS’s full support for object-oriented (OO) concepts makes it especially suited for representing class hierarchies used in OO programming languages. Inheritance makes schemas more maintainable by better encapsulating the natural hierarchies that occur in data. WXS’s support for inheritance and other OO features quite closely matches features found in Java with the only difference being that inheritance in Java works on classes whilst in WXS it works on types and elements. A WXS data type is essentially a Java class without any methods, a struct in C or a record in Pascal. WXS supports polymorphism for both types and elements (via substitution groups).

Referencing support in WXS is superior to the basic system of global IDs and IDREFs that is found in DTDs. WXS supports a subset of XPath [30] which allows keyrefs to unambiguously specify the elements they can refer to. This solves the problem of DTDs whereby an IDREF can refer to an ID of any element in the document. A system of defining scopes of where the key and uniqueness constraints should apply is also provided.

2.1.4 Ease of use. [2] identifies WXS to be the hardest schema language to use. This is understandable given that WXS uses a complex type structure with four separate hierarchies for simple and complex types as well as attributes and elements. This hierarchy is much more complex than type hierarchies used in object-oriented programming languages. Java for example only has 2 type hierarchies – for classes and interfaces with a single type – Object at the root of the type tree. Other schema languages use simpler approaches to typing. Schematron for instance hasn’t got any typing system as such. Complex type system used by WXS can be seen as one of the barriers to its wider adoption.

Geert Jan Bex [5] notes that “the [W3C] XML Schema specification is very hard to read”. Even experienced language designers familiar with DTDs and EBNF [6] might find it difficult to master WXS with its redundant syntax that is radically different from the concept of production rules used in EBNF or DTDs. RELAX NG tries to address this issue of migrating to a schema language smoother by introducing a compact non-XML syntax (RELAX NG Compact Syntax [35]) that is semantically equivalent to the XML version but is more similar to a DTD. Schemas expressed using compact syntax can be converted to an RELAX NG XML format using a software tool. There have also been attempts by Wilde et al [7] to develop a similar, compact yet human readable format for WXS, called XML Schema Compact Syntax (XSCS). This was done by defining an alternative syntax for WXS using EBNF productions and providing a tool that would convert between WXS and XSCS. Wilde [7] managed to reduce the size of processed schemas by more than 50% whilst making them more readable by removing the syntactic redundancies introduced by XML. Regrettably XSCS was never made part of the official
WXS specification and there doesn’t seem to be any further activity on the XSCS project with the project’s home page being last modified in 2004.

2.2 Industry adoption

Recent years have seen many applications migrating to using XML as their main format for data storage. XML is human readable, highly portable, can be easily generated and consumed programmatically and has a large collection of software tools available ([1] currently lists over 70 applications that provide WXS related functionality). However it has one significant disadvantage. Well-formed, valid XML document can be easily invalidated by a single typing error or corruption of a single bit of data. The error diagnostics provided by the XML tools in this case are often misleading or unintelligible, for instance, they may point to the end of the document instead of the line that caused the error. Additionally WXS lacks a mechanism to specify custom error messages on failed semantic checks found in other schema languages such as Schematron [28]. Below is an example validation error diagnostic caused by an invalid character in the OID attribute.

```
cvc-pattern-valid: Value 'o1.2.826.0.1.3344810.1.1.14' is not facet-valid with respect to pattern '\d+[(\..\d+)]*" for type 'OIDType'....
```

Because of the use of a global type, the error report is missing the name of the attribute that caused the error as well as the line and column numbers for the point in the document at which the error occurred. This makes it virtually impossible for the user to fix a broken document without in-depth knowledge of the schema that was used to validate the document.

Early XML based applications were reliant on DTDs to validate the syntax and structure of the data, with custom code having to be written to validate the semantics. Other applications performed validation entirely in program code. This would give the developers greater control over the format of data used by their software as the application code would be a single point of change on the way to modifying the format of data. Introduction of WXS as a W3C recommendation 7 years ago should have triggered a move from DTDs and manual validation to the use of schemas, which offer better expressiveness and degree of checking. Schemas also allow the data formats to be efficiently communicated to other developers thus increasing the interoperability between applications. Wide adoption of WXS however hasn’t taken place. Table 1 shows a number of modern applications that use XML data formats along with the type of validation they are using (Custom, DTD or WXS).

<table>
<thead>
<tr>
<th>Application</th>
<th>XML Usage</th>
<th>Validation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td>Validation of deployment descriptors</td>
<td>WXS</td>
<td>HQ WXS</td>
</tr>
<tr>
<td>Hibernate</td>
<td>Config. files, XML-relational persistence</td>
<td>Limited use of DTD.</td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>Data exports in XML format</td>
<td>DTD</td>
<td></td>
</tr>
<tr>
<td>MS Office 2007</td>
<td>Office Open XML format</td>
<td>WXS</td>
<td>HQ WXS with support for user extensibility</td>
</tr>
<tr>
<td>Sun Web Server 7</td>
<td>Web application descriptors</td>
<td>DTD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Server config. files</td>
<td>Custom</td>
<td></td>
</tr>
<tr>
<td>Java Persistence API</td>
<td>Persistence config. files</td>
<td>WXS</td>
<td>HQ WXS</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>Build files</td>
<td>Custom</td>
<td></td>
</tr>
<tr>
<td>Windows Vista gadgets</td>
<td>Gadget metadata files</td>
<td>Custom</td>
<td></td>
</tr>
<tr>
<td>Netbeans 6</td>
<td>Plugin manifests</td>
<td>Custom</td>
<td></td>
</tr>
<tr>
<td>iTunes</td>
<td>Music library</td>
<td>DTD</td>
<td></td>
</tr>
</tbody>
</table>

Toman [15] performed extensive analysis of 16534 XML documents available on the Internet to illustrate trends for various XML-related criteria such as size of documents or validation technology used. Figure 1 graphically illustrates the findings in [15].

---

2 HQ WXS is an abbreviation for high quality WXS, meaning that the Schema is not simply a DTD expressed using WXS syntax, but rather uses a full set of validation features that are available to the standard.
Evidently DTDs still have the majority of the market share for XML validation. We believe that this can be explained by the steep learning curve of the WXS Schema standard, the limited availability of Schema Validators in the past and the general reluctance of programmers/businesses to change the architecture of their production code to include WXS validation. It is also worth noting that DTDs have been standardized together with SGML [17] more than 20 years ago, amassing a significant user base in this period.

XML languages such as XSLT [18] or DocBook [16] would be another obvious domain of application for XML Schemas. Publishing an official schema for the language enables software to easily check the validity of documents and provides additional benefits that arise from the data type information available in the WXS. Those can be as diverse as providing code auto-completion or giving access to documentation features available in WXS. Table 2 shows the availability of Schema language definitions for a number of modern XML languages.

Table 2. Availability of Schemas for XML based languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Official Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSS 2.0</td>
<td>None available</td>
</tr>
<tr>
<td>Atom</td>
<td>Relax NG</td>
</tr>
<tr>
<td>RDF</td>
<td>WXS</td>
</tr>
<tr>
<td>DocBook</td>
<td>Relax NG (Instances still use a DTD)</td>
</tr>
<tr>
<td>SVG</td>
<td>‘None available’</td>
</tr>
<tr>
<td>SAML</td>
<td>WXS</td>
</tr>
<tr>
<td>XML-RPC</td>
<td>None available</td>
</tr>
<tr>
<td>WSDL</td>
<td>WXS</td>
</tr>
<tr>
<td>XUL</td>
<td>None available</td>
</tr>
<tr>
<td>XHTML</td>
<td>WXS</td>
</tr>
<tr>
<td>WXS</td>
<td>WXS and RELAX NG</td>
</tr>
</tbody>
</table>

Most of the languages we looked at are defined using WXS. RELAX NG however is rapidly catching up with an increasing number of working groups (most notably SVG [8] and XHTML [9]). This is quite alarming given that the WXS specification has been developed by W3C. Data gathered by Wim Martens [11] from a population of 819 schemas shows even worse trends. Figure 2 shows our rendering of the data from [11].

With only 4 percent of schemas using features specific to WXS it is evident that the majority of schemas in use were automatically (and often incorrectly) generated from DTDs. These results agree with those in [5] – only 15% of the schemas looked at used data typing features not available in DTDs. There is also strong correlation with [15] that states that “85% of XSDs define local tree languages (languages that can be defined by DTDs as well)”. It is clear that despite the availability of a high number of novel features such as inheritance, modularity or polymorphism in WXS, the majority of developers failed to embrace the W3C standard. The minority that did switch to WXS used the subset of features that was readily available in DTDs, making the migration unnecessary.

2.3 Overview of WXS related technology

2.3.1 Parsers. The early days of XML saw the rise of a number of competing software packages providing XML related functionality (parsing, schema validation, XSL transformations). Clark [19] performs a comparison of 6 different parsers and [20] notes that “as of March 2006, Apache has several Java parsers: Crimson, Xerces 1, and Xerces 2”. Those two sources show considerable diversity in the parser market that can be considered a major disadvantage – certain application might use a parser that is not available on all operating systems the application is targeted at. Additionally, migrating from one parser to another...
becomes a problem because of API differences. The Apache Xerces project successfully tried to address these issues by providing a single open-source cross-platform fully standards-compliant parser. Xerces is bundled with JRE since version 1.5 and supports a range of W3C XML standards and recommendations. Implementations of Xerces are available in Java, C++ and Perl.

2.3.2 XML Editors. WXS is a valuable addition to any XML editor. It transparently provides features such as code auto-completion, inline documentation and immediate validation of documents. An editor with WXS support can give the user immediate feedback on any structural and datatype errors. All those features however are subject to availability of the WXS for the target XML language. Almost all XML editors on the market include support for WXS. Those also can be used to efficiently develop WXSs themselves as an official schema for the WXS standard is available [31].

2.3.3 Automatic WXS generation. The complexity of WXS makes it extremely difficult to migrate to it from other Schema Languages or DTDs. The situation is somewhat improved when an existing DTD is available. There are a number of applications that allow WXSs to be automatically generated from DTDs. These tools often are also able to extract WXSs from raw XML data. This approach however is extremely error prone. Chidlovskii [21] argues that the extracted schema should both “tightly represent the data, and be concise and compact” in order not to break upon small variations that naturally occur in data (e.g. optional presence or absence of some elements or attributes). These two requirements are in conflict with each other. This makes the task of WXS extraction extremely complex. [21] describes the first attempt to infer WXSs from “heterogeneous collections of XML documents”. Software able to do this is not readily available, so most developers have to rely on schemas generated from pre-existent DTDs. We believe that this is the main reason for 85% of schemas being equivalent to DTDs [15]. Those automatically generated schemas have a number of disadvantages. Because of limited data type support in DTDs, the software cannot infer the exact datatype (an integer for instance) and is forced to either use the DTD type available in WXS such as xs:IDREF or the most generic datatype, normally xs:string. This doesn’t make any use of the extended datatype collection [3] provided with WXS. Additionally these automatic tools only produce WXSs where every element is declared at the global level. This is somewhat undesirable as WXS lacks a way to specify the root element of a document and assumes that any globally defined element in a schema might serve as a root. Figure 3 graphically represents this issue.

![Figure 3. Example XML tree with valid sub trees](image)

If an automatically generated schema is used, a document containing only one of the 5 sub trees rooted at B, C, D, E or F (circled) from the example document A will be declared valid when they all are actually semantically incomplete i.e. represent only a part of a document. Automatically generated schemas however are a good starting point on the way to producing a high quality WXS, provided that the machine generated schema is then hand edited to more tightly represent the data using the WXS data typing features.

2.3.4 OO Data Binding. Once a schema instance is validated, every element and attribute in the instance can be assigned a particular type based on the schema definition for the element or attribute. This information then can be used to generate a class that faithfully represents the data in an XML document complete with all the data types [36]. This enables the programmer to query the XML data represented by this class using standard programming language methods removing the need to use an intermediate API such as DOM in the business logic. This approach is similar to that of object-relational mapping systems and can significantly improve clarity and maintainability of the code that accesses the XML data.

3. WXS in the context of PERMIS RBAC policies

In order to show by example the path to conversion of XML data validation from a DTD and custom code to WXS, we undertook the exercise of converting the PERMIS role based access control (RBAC) policy DTD to WXS. This exercise highlighted areas of
semantic validation that still cannot be effectively undertaken by WXS and the problems that are associated with such a migration.

3.1 Use of XML in PERMIS

PERMIS [23] is an application independent modular Java framework that validates user credentials and makes access control decisions based on RBAC policies. The PERMIS software suite also provides tools for managing RBAC policies, role assignments and delegations between users. Policies in PERMIS are stored as XML files, which can be edited with a bundled Policy Editor or manually by the user. PERMIS initially defined a DTD that governed the structure of the RBAC policies. Later, a WXS was automatically generated from this DTD. Neither however was used in production code because of associated integration difficulties. The flow diagram in Figure 4 illustrates the use of XML in PERMIS.

Figure 4. XML parsing in PERMIS

Semantic validation is mandatory for all the policies that are read in and is performed using custom Java code. However the structure of the policy isn’t validated as the code navigates the document tree using element names only. Parsing a policy with arbitrary structure and the same element names would cause unpredictable results.

The Policy Editor (PE) bundled with PERMIS is used to produces policies in XML format and guards the user from the difficulties of manually editing XML data. Policies generated by PE are guaranteed to be semantically and syntactically valid as a range of checks is performed on the data the user inputs. Figure 5 illustrates this process.

Figure 5. XML policy output in PERMIS

WXS isn’t used for user input validation as the Validator errors would be virtually impossible for the users to understand. Additionally it would be necessary to regenerate the policy XML on every user change, which would require significant performance overheads. Instead validation is performed as part of the GUI and provides helpful error messages.

3.2 Design principles

Semantic validation in PERMIS is performed manually in policy parsing code over 3000 lines long. Extensibility of the format is achieved using dynamic instantiation of classes representing elements within the policy using reflection. No DTD or Schema was used to validate the structure of policies. This was a potentially risky approach as the policy parsing code would produce unpredictable results if the policy conformed to a different XML structure. Therefore we considered it important to introduce schema validation as the first step of the policy parsing process. This approach would allow structural validation to identify any errors before semantic validation commences, thus eliminating any structural errors at the lowest level possible.

In parallel with introducing WXS validation it was also decided to improve the semantic structure of the policies. Similar to many other projects such as Ant, PERMIS would store lists of actions that can be applied to a target and several other parameters as a comma separated strings of values in attributes (Apache Ant used the same approach for storing arguments for javac, jvm and the like. This is now deprecated [24]). This is a poor design as it introduces
data escaping problems and also limits the maximum number of attributes as parsers normally limit the maximum length of an attribute they can handle. Instead, we decided to store actions as a list of zero or more elements within the Target element [25].

3.3 Integration

WXS was integrated in several steps as follows:

1. Develop and test a self contained schema validation package using JAXP (Java API for XML Processing) [20]. This should include provision for parsing other schema languages as and when they become part of JAXP.
2. Integrate the schema validation package into the policy parsing code with minimum changes.
3. Provide ways for the end users to override the default schema with their own versions in order to provide validation of the generic Obligations format.
4. Provide tools to automatically convert old policies to the new format.

Point 3 needs further explanation. PERMIS RBAC policies allow the concept of Obligations to be included in them. The XML structure of these conforms to the XACML standard [27], in that they are a sequence of attribute assignments. This is a very generic structure, whilst obligations are actually application dependent actions that must be undertaken if the user is to be granted (or denied) access to the requested target resource. Since the obligations are application dependent, different users of PERMIS may need to define different obligation schemas. Hence they need a mechanism to extend the policy and its schema. Extending the schema however should be optional and the default schema should allow documents with content conforming to the XACML Obligation element.

During the integration we encountered several minor problems related to the structure of the JAXP API. The most notable one was the inability to validate a policy if it was parsed using an InputStream object rather than a File or a String. This was due to the nature of InputStream objects in Java – they can only be used once. We overcame this problem by reading the input stream to a String. No public API changes were introduced as a result of this change. Another issue was the Document Type Definitions used by some of the policies in the PERMIS regression test suite. The standard Entity Resolver in the Xerces DTD Validator in some cases wasn’t able to resolve a reference to one of the DTDs and terminated with an error. JAXP provides a way to override the standard entity resolver in a parser with a custom one, preventing these errors. The schema checking API however lacks this feature, so we had to read the problematic policies using DOM before passing them to the validation API as DOM Document objects. DOM Level 3 [38] declares the documentType attribute as read-only, preventing the invalid doctype declarations from being corrected automatically using standard DOM API.

PERMIS previously used the Crimson parser which was bundled with Java 1.4.2. We have removed it from the code base in order to force PERMIS to use the modern Xerces parser bundled with JREs since version 1.5. Transition to Xerces went smoothly thanks to JAXP hiding the parser implementation details from the application code. This easy migration process clearly illustrates the benefits of JAXP.

In the initial stages of the project we considered converting the XML parser class to using DOM parsing instead of SAX. However, upon inspection of the Xerces sources [34] we discovered that both DOM Documents and SAX InputSource files are parsed natively using SAX or DOM without intermediate conversions. Thus no benefit would have been gained from this conversion and it would have required a significant amount of work to implement.

3.4 Schema design considerations

Upon starting the project we identified the following requirements for the PERMIS WXS:

1. The PERMIS WXS should use the complete set of features applicable to identify as many structural and semantic errors as possible before passing the policy XML to the policy parser.
2. WXS should provide mechanisms for the user to extend the PERMIS XML policy format by allowing user defined XML content in the Obligation element.
3. WXS should be extensible to enable the user to optionally validate custom XML content in the Obligation element.
4. The WXS part of the code should be portable and only use functionality specified by JAXP.

As our starting point we used a simple WXS automatically generated from the latest version of the PERMIS DTD. When choosing a schema structure to use we considered 4 possible options described by [26].
1. **Russian Doll** – one global element, subsequent elements are located inside along with nested anonymous complex types
2. **Salami Slice** – all elements are declared globally and are referenced by anonymous complex types. Automatic WXS generators produce schemas with this structure
3. **Venetian Blind** – all complexTypes are global. Elements are declared locally with the exception of the root element.
4. **Garden of Eden** – Both elements and attributes are declared globally.

We agree with [26] that adhering to a particular design pattern in our schema would not bring any functional benefits and decided to use a more freeform approach. For the sake of readability we defined most documents at the global level as this is a structure familiar to most DTD and EBNF users. Reusable complex types were also declared at the global level to provide an inheritance hierarchy for elements such as IF statements. In our design we tried to make maximum use of the OO-influenced features such as inheritance and polymorphism. Figure 6 illustrates the representation of the IF statement syntax as used in the policies.

![Figure 6. UML diagram of IF statement grammar](image)

Previously the Policy DTD declared the IF statement by enumerating every single operator that was supported. This has resulted in long winded, difficult-to-extend definitions. The developed WXS solved this problem by introducing a three layer inheritance hierarchy with a single abstract `BooleanOp` element. This hierarchy and the provision for polymorphism in WXS have allowed us to define the IF statement as a single `BooleanOp`, greatly improving clarity and extensibility of the IF grammar. This grammar can be easily extended by creating new operators that use the `BooleanOp` substitution group.

### 3.5 Providing WXS extensibility in PERMIS

When choosing the mechanism to make PERMIS aware of the WXS it should use for validating policies, we decided to bundle the WXS in the PERMIS package. This has solved a number of issues.

- Loading a schema from the URL specified in the `schemaLocation` attribute on an instance document is insecure as it might trigger accessing a potentially malicious resource. Access might also be blocked by firewalls resulting in PERMIS not being able to validate the policies.
- In JAXP the WXS validation process is separate from parsing, so referencing the WXS URL in documents would still require a separate validation procedure to take place.
- Hosting the schema at a single URL introduces a single point of failure for PERMIS installations – if the web server is unavailable, PERMIS will not be able to validate the policies.
- It’s easier to make the schema user extensible when it’s stored locally on the user’s machine. The user can deploy his custom WXS by changing a single configuration file without the need to manually edit each of his policies to include references to his custom schema.

In order to match the OASIS XACML specification we had to define the Obligation element as a complexType with the attributes defined in [27] along with any arbitrary XML content. This approach allows PERMIS users to optionally define validation rules for their arbitrary content by modifying the bundled WXS. If validation is not required by the user, his custom policies will still pass validation by the default schema as any arbitrary XML content is allowed in the Obligation element.

### 3.6 Deficiencies of WXS

Even though WXS is the most powerful schema language of the ones currently available it still lacks certain important features. We will now look at the issues we encountered during the development of the PERMIS schema.
3.6.1 Keying and referencing. WXS has got excellent support for these features. Keys and keyrefs are a step forward from ID and IDREF data types found in DTDs. They allow the element to which a certain attribute refers to be explicitly specified with an XPath expression thus removing any ambiguity and essentially creating a foreign key relationship between the two elements. Regrettfully those are one of the least used features in the WXS standard. According to [11] linking features are only used in 4.1% of the valid WXSs looked at in the study. We believe that this can be explained by the complexity of the WXS approach to referencing. WXS tries to address deficiencies of ID/IDREF system found in DTDs by defining scopes in which uniqueness for key elements and reference integrity for keyref elements is to be maintained. Keys and keyrefs have to be declared on the root of the tree in which the corresponding constraint is to hold. Thus most keyrefs are defined in the root element. If the document uses many such references, the definition for the root element becomes very big and hard to read. Figure 7 graphically shows this model.

![Referencing model used by WXS](image)

Figure 7. Referencing model used by WXS

We believe that a more intuitive design would be to define keyrefs directly on the elements to which the constraint applies (D elements in our example) rather than the elements that contain the scope of the constraint. This would eliminate the need for numerous keyref definitions in the root element and make the referencing system more intuitive. One drawback however will be the inability to define relationships where key/keyref constraint should selectively apply to elements with the same name based on element’s position within the document tree.

We also encountered a number of problems related to WXS only supporting a subset of the XPath specification [30]. For example it is impossible to use the descendant’s syntax (//some-element) in WXS. This results in complete xpaths needing to be used in keyrefs within the root element. For example defining a keyref that requires that every AllowedAction element should refer to one or more existing Action elements with their ID attribute requires the full path from the policy root to the AllowedAction element to be specified. It is difficult and error prone to do this with an XPath expression e.g. ./TargetAccessPolicy/TargetAccess/TargetList/Target/AllowedAction. Full support for the XPath standard would allow this to be written in a much more readable way as //AllowedAction.

3.6.2 Choice construct for attributes. WXS has no choice construct for attributes. A typical usage of this can be illustrated by the following example: a sensor uses an xml file to define the threshold values for the environment in which it operates. An example of this would be the following:

```xml
<threshold min="1" max="50"/>
```

The threshold should include either min or max or both attributes. WXS can be used to make min and max optional. This however will not prevent the case where both attributes are absent. WXS provides a solution when min and max are re-declared as elements. This however might be problematic if the data format is already widely deployed.

3.6.3 Semantic validation. Semantic validation in WXS is limited to uniqueness and referencing constraints. There is no way to specify even the simplest semantic rules such as restricting the value of the max attribute to be greater or equal to the value of the min attribute. Co-occurrence constraints such as making the presence of an element mandatory based on the presence of some other element cannot be expressed either. There is no way to make the content of the element dependent on the context in which it appears. For instance in a PERMIS policy it’s impossible to declare the type for the OID (Object Identifier) type in such a way that if it appears on the root element of the policy it can take the value of an OID or a URN OID, whilst in all other locations in the policy it should be a simple OID. Many of those issues are related to the absence of support for the full XPath specification [30] in WXS. XSLT and Schematron
provide greater semantic flexibility by allowing conditions with XPath node tests for template and pattern definitions respectively. Introducing a similar approach to WXS would greatly enhance the semantic validation capabilities of the language. For instance it would allow the use of XPath expressions to determine the context of the current element and use the correct type for the element definition based on that context. Another benefit of this addition would be the ability to use a large set of functions pre-defined in XPath for tests on WXS data types such as `<xs:date>`, `<xs:string>` or `<xs:integer>.

3.6.4 Representing context free grammars. When developing a WXS for PERMIS policies we have come across a number of problems related to representing context free grammars such as LDAP DNs. Those are usually defined using either BNF [32] or EBNF. WXS is very limited when it comes to such data, as the only mechanism of representing it is the use of a single regular expression or a generic `<xs:string>`. Regular expressions can normally only be used for simple data such as postcodes or OIDs, leaving more complex formats for custom validation in the program code. Having support for BNF within the schema would allow for easy representation of LDAP DNs and similar formats. A form of BNF should be simple to integrate into WXS as the BNF definition of BNF format is only 8 lines long [33]. One possible argument against this change is whether there is sufficient demand to express context free grammars within WXSs.

4. Conclusion

WXS is an extremely powerful tool for validating XML documents. Despite its steep learning curve, highly verbose syntax and unintuitive specification of some of its most advanced features, it can be successfully used as an augmentation to semantic validation performed in a conventional programming language.

WXS validation was successfully integrated into PERMIS with a minimum of source code changes. An additional benefit of this was the standardisation of the new, more semantically clear policy structure and the conversion of all the legacy policies into this format.

As WXS was unable to cover even the simplest of some semantic constraints, we propose introducing support for the full XPath specification in a variety of contexts in the next version of the WXS standard. Having XPath support together with conditional type definitions for elements and attributes would allow for much more accurate representation of many constraints found in real world data.

In addition to this we propose the addition of mechanisms for user friendly feedback similar to that found in Schematron. This would allow concentrating both the semantic constraints and advice on their resolution in a single WXS schema, removing the need to do this in custom code.

We are aware that most features in WXS that are new with respect to DTDs aren’t being heavily used. However we hope that further improvements to the WXS standard together with the availability of quality software tools such as Xerces should win WXS a wider support in the developer community in the future.

5. References


6. Appendix 1: Increasing the expressiveness of the W3C XML Schema Language by introducing XPath support into the standard.

6.1 Introduction

When working on integrating WXS checking into PERMIS we have extensively studied the W3C specifications documents in order to make efficient use of the maximum number of features relevant to semantic constraints used in PERMIS policies. We found that WXS support for semantic validation was more than adequate in respect to uniqueness and referencing constraints. However when compared with rule based languages such as Schematron WXS had very limited ability to express even the simplest semantic rules such as limiting the value of the Min attribute to be less than the value of the Max attribute. We are aware that Schematron supports the full set of the XPath language and can be used in conjunction with WXS. Unfortunately, lack of out of the box support for Schematron on programming platforms such as Java or .Net limits the area of its application. We therefore propose a set of XPath features to be added to various WXS element as follows.

6.2 Modifications to the current WXS standard

1. Support XPath expressions in minInclusive, maxExclusive attributes in simpleType definitions. This would allow values of simple types to be dependent on other element and attribute values.

```xml
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive xpath="../some-element/@count"/>
  </xs:restriction>
</xs:simpleType>
```

2. Provide extended XPath support in definitions for key, keyref and unique elements. This will enable more compact XPaths to be written using the descendants syntax (//=element) as well as give developers the ability to selectively enforce reference constraints based on values of attributes and other parameters.

```xml
<xs:keyref name="SOASpecRef" refer="SOASpecKey">
  <xs:selector xpath="//SOA"/>
  <xs:field xpath="@ID"/>
</xs:keyref>
```

3. Introduce an if construct with semantics similar to the xsl:if element in the XSLT specification. This should be allowed in the xs:complexType definitions and will enable content models to be defined based on the results of XPath node tests. WXS itself is defined using conditional type definitions [37], which however cannot be expressed in WXS itself. This change will enable easy definition of co-occurrence constraints and the like.

```xml
<xs:complexType name="DateType">
  <xs:if test="../country/@code = 'us'">
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="us-date"/>
    </xs:sequence>
  </xs:if>
  <xs:if test="../country/@code = 'uk'">
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="uk-date"/>
    </xs:sequence>
  </xs:if>
</xs:complexType>
```

4. Support the complete set of XPath built-in functions in all of the above.

5. Provide a construct to define custom error output in all WXS elements that define semantic constrains.

6.3 Implications of proposed changes

Introducing XPath support will require introducing an extra level of semantic checking into the WXS tools and Validators [1]. Linking elements with their constraints using XPath requires that all XPath expressions are semantically correct in the context of the schema document in question. We also realize that the current version of the WXS standard is optimized for schema processing performance. Full XPath support will somewhat worsen the Validator performance by virtue of the additional XPath processing required. This might be especially noticeable on mobile and embedded devices. However given that in almost 7 years that have passed since the introduction of the WXS standard the speed of the hardware has improved, those changes may have only a limited impact on most systems’ performance. On the other hand, the fact that most of the current semantic checking functionality is not widely used is one strong argument against adding additional checking to the
standard. Efficiently writing a semantic schema with most of the checks performed using XPath expressions will require familiarity with XPath from the developers. This will only make the learning curve for WXS steeper. When used properly however, the proposed functionality will be able to remove the majority of semantic checking from the program code thus improving software extensibility and maintainability. Providing XPath functionality in WXS should create a single expressive schema language that builds on two established W3C standards.