Tooling Metamodels with Patterns and OCL

D. H. Akehurst, O. Patrascoiu

University of Kent at Canterbury
{D.H.Akehurst, O.Patrascoiu}@kent.ac.uk

Abstract. Computing is moving to a new paradigm where models are first class artefacts. Metamodelling is one of the foundations of this future. However, it is all very well to have metamodels and languages with which to define them (i.e. MOF), but what should we do with them once they are defined? One task should be to populate the model described by the metamodel definition and ensure that the well-formedness constraints are correctly specified; another task may be to create a tool based on the metamodel. In order to enable experiments with variations in the metamodel an automated approach to building such tools is required. Judicious use of patterns can facilitate automatic generation of such tools. The ability to auto-generate a tool from a metamodel definition facilitates experimentation and general confidence in the definition of the metamodel. The tool generated can be subsequently used as the foundation for a more functionally rich hand-coded tool.

1 Introduction

Models are becoming 'primary artefacts' in modern computing. The Object Management Group's (OMG) Model Driven Architecture (MDA) [1] strategy envisages a world where models play a more direct role in software engineering and production. Metamodelling is a key facility in this new era; it provides the facilities to support the design of models, i.e. a metamodel is a model of a model. The OMG has defined the Meta Object Facility (MOF) [2] as a language for defining metamodels; but what do we do with a model specification (an expression in MOF) once it is defined and what support can be provided to aid the specification of models?

- Can we check that a population meets the well-formedness constraints?
- Can we implement the specification?
- Can we provide tools to support the defined model?

The aim of this paper is to demonstrate that, the use of programming patterns and the provision of support for the Object Constraint Language (OCL) [3], enables us to say yes to all of these questions.

OCL provides the Unified Modelling Language (UML) with a navigation and constraint expression facility. By implementing this language over the metamodel specification language MOF, a sub set of UML, we provide a facility to check the well-formedness constraints that typically form part of many model specifications. In addition the OCL implementation provides a very useable *query language* [4] for exploring metamodel populations.

Modelling raises the level of abstraction at which computing is done; programming patterns enable us to map modelling abstractions onto today's 'executable' languages. Traditionally higher level constructs are mapped onto lower level ones using a particular pattern. Much of this has in the past been about behavioural constructs

mapping to a pattern of lower level (assembly instruction) constructs. However, modelling is more of a structural facility; hence we form structural patterns in addition to behaviour patterns. Such patterns are becoming common place, although often implemented by hand. One of the most well-known is that of accessors and mutators as implementation patterns for class properties. Many tools support such patterns in their code generation facilities.

The mapping from metamodel onto implementation can be seen as an MDA exercise requiring a mapping from MOF (PIM) to a programming language (PSM). However, as there is as yet no standard specification language for transformations and more importantly, as the PSM is a programming language, we feel that specifying the mapping in a manner that resembles the target programming language expression, is more appropriate. There are many tools that will support code generation from UML models, e.g. Rational Rose [5], Poseidon [6], Together [7].

However, what we are proposing here is something more than simple code generation. We support generation of a complete framework useable for populating, manipulating and exploring the specified model (or metamodel).

To fully realise the power of modelling in an implementation we need to support complex patterns; such as Visitors, Factories, Repositories, support for bi-directional Associations (opposite ends should always refer to each other), and possibly many others. To achieve this in a fully flexible manner it is necessary to provide a tool that enables the specification of the required patterns in such a way that the instantiation of the pattern takes its parameter values from the specified model.

We already have a language for navigating metamodel specifications – OCL. This language facilitates the extraction of template parameters from the specified model; to provide a full template language we need to add mechanisms to:

a) compose the string values that define the implementation code
b) generate files and directories for storing the implementation code
c) call sub-templates with appropriate parameters

Ideally, an appropriate template language would be designed and specified (or an existing one altered) that incorporates OCL as an expression language. However, as an interim solution, we have found that a few minor extensions to our OCL implementation provide us with the required functions.

By defining templates for a number of interacting patterns we are able to generate, from the specification of a model, a tool that supports:

- Building and manipulating model populations
- Viewing model populations
- Evaluating OCL constraints and expressions over the population
- Provision of persistent storage as XMI [8], HUTN [9] (or other formats).

This is demonstrated through the use of a tool called the Kent Modelling Framework (KMF) [10], developed and used at the University of Kent at Canterbury; the tool has been under development and use for over five years, supported by a number of different research projects. The tool has been used as part of those projects to generate support tools for a variety of different models, such as: the UML metamodel (versions 1.4, 1.5, 2.0); the OCL (abstract syntax model) version 2.0; a metamodel (discussed in [11]) based on the Reference Model for Open Distributed Processing (RM-ODP) [12], as part of the DSE4DS project [13, 14]; and diagrammatic language models, as part of the Reasoning with Diagrams project [15].

In addition, due to the use of OCL as the template language we are able to generate an implementation of KMF using itself.

The rest of this paper is organized as follows. Section 2 described the patterns generated by the latest version of our KMF tool. Section 3 describes how we use OCL as a template language. Section 4 discusses how the generated code is fitted together into a useful support tool for the original model specification. Section 5 discusses existing related work to that presented in this paper. Section 6 concludes the paper with a summary and discussion of future work.

2 The Patterns

The generation of our modelling tools from a metamodel is achieved through the use of a number of interacting programming patterns. Information is taken from the metamodel specification and used to populate the parameters of each pattern. Each of the major concepts from the metamodel, class, attribute, association and package, are used to instantiate a different set of patterns; described in the following sub sections. We currently use Java as the target implementation language; it would be easy to adapt the templates to additionally target other languages.

Patterns have been recognised as a useful programming technique for a number of years. In particular the book by the "gang of four" [16] discusses a number of widely used patterns. Most of our patterns are taken or adapted from those documented in this book.

2.1 Classes

Each metamodel class is implemented using a pattern of interface and implementation class. An interface is constructed to represent each class and the type hierarchy of the class; a key value of this pattern is that the implemented type hierarchy will support multiple inheritance. The name of the interface matches the name of the metamodel class. The implementation class is named after the metamodel class, but with an added extension to the name – such as 'Impl'.

2.2 Attributes and Associations

Each attribute of, and association end that is navigable from, a particular metamodel class is implemented by adding appropriate accessor and mutator methods in both the interface and implementation class. These method signatures follow the standard java pattern as shown below:

```
public <Type> get<CapitalisedName>()
public void set<CapitalisedName>(<Type> <name>)
```

The implementation of attributes and associations differ. Attributes are implemented simply by providing an appropriately typed private variable, which is returned or assigned in the accessor and mutator bodies.

Two alternative patterns are used to implement an association end. The choice between patterns is based on whether or not the association end (or its counter part at the other end of the association) is marked as 'navigable'.

If only one end is marked as 'navigable' then the same implementation pattern is used as that for attributes; the class navigated from will contain a private variable returned and assigned by the accessor and mutator.

If both ends of the association are navigable then we must ensure that setting one end will also set the other end appropriately. A variety of programming techniques could be used to achieve this; the simplest being for the mutator implementation to set the opposite end as well as setting this end, i.e.:

```
public void set<CapitalisedName>(<Type> <name>) {
  this.<name> = <name>;
  if (<name>.get<OtherName>() != this)
    if (<name>.get<OtherName>() != null)
      <name>.get<OtherName>().set<Name>(null)
    <name>.set<OtherName>(this);
}
```

This mutator first sets the private variable for this end of the association. The next conditional test is to eliminate a non-terminating recursive loop, without it, each end of the association would attempt to set the other end, which would set the other end, which would set the other end…etc. The second test determines if an existing object is already using the new object as one of the association ends, if so we should remove it (i.e. set it to null). Finally, the other end of the association is set to reference this object as the appropriate end of the association.

This pattern is fine for Association ends with multiplicity of one (or zero-to-one), however, when we have a collection of objects at an association end, something more complex is needed. There are two approaches to implementing this type of association; one is to provide 'Add' and 'Remove' methods on the class that references the collection; the other option is to use accessors and mutators which get and set collection objects.

To ensure the referential integrity of opposing association ends with the first of these implementation approaches, a similar technique can be used to that for single object association ends; the Add and Remove methods can be implemented so as to set the opposite ends of the association.

Performing the same actions when the second approach is used is more complex. Standard Java (or other language) collections are used to implement the model collections and the mutator will allow any collection that implements the collection interfaces to be used as an argument. Our approach to implementing the referential integrity is to provide a separate object that implements an associationEnd. Objects that take on the role of an associationEnd within a model implementation must implement an AssociationEnd interface, the functions of which are delegated to the separate associationEnd object. The AssociationEnd interface is shown below:

```
public interface AssociationEnd {
  Object getOtherEnd(String propertyName);
  void setOtherEnd(String propertyName,
                   Object value,
                   String otherName);
}
```

The accessor and mutator are qualified by a string value in order to distinguish between multiple associationEnd roles supported by an object. The mutator, in addition, takes a value for the opposing association end name as a parameter, so that

the referential integrity actions can be caused. The implementing (delegate) object for this interface performs actions similar to those discussed above, with some variation based on collection objects. Setting a collection object causes a new wrapper collection to be created. This delegates all collection functions to the original collection, but intercepts add and remove methods so as to cause additional actions for setting the other end of the association.

2.3 Packages

There are three patterns instantiated from metamodel packages – Factory, Repository and Visitor. Each of these patterns addresses the classes owned by the metamodel package and addresses any sub packages.

Factory

Based on the traditional pattern for a factory object [16], we provide a variation that scales more easily to large models. The idea of the pattern is essentially to provide an interface for constructors of the classes in a model (Java does not facilitate constructors in an interface specification). A factory interface contains a create method for each (non-abstract) class in the model, the implementation of the factory supports generation of an object of the appropriate class, the returned object being referenced by the implemented interface type (rather than the actual implementation class). We additionally provide a generic create method that takes a (model) class name as parameter.

Our original implementation provided a single factory for a whole model; however we discovered that this approach was not scaleable, causing problems with very large models; and meant that we had to treat the whole model as an entity, where as sometimes we wished to deal with a single package (and sub-packages) of the model. An alternative that we have investigated is to construct a factory implementation class (and interface) for each metamodel class; however, we found that this was unnecessary.

Our preferred evolution of the factory pattern is to provide a factory for each package. These factories are linked in a hierarchy that matches the package structure; any factory can be used to create objects from its own package or any sub-package.

We create an implementation Factory class for each package; the class implements and extends a common Factory interface and implementation which provide common behaviour. The template for the generated class is as follows:

```
public class <pkg_name>Factory extends FactoryImpl {

  public <pkg_name>Factory() {
    <for each subPackage>
     <spkg_name> = new <subFactory_name>();
  }

  <for each subPackage>
   public <subFactory_name> <spkg_name> = null;

  <for each class in this package>
   public <class_name> create<class_name>() {
     return new <classImplName>();
   }
   public void destroy<className>(<className> object) {
     <for each attribute or associationEnd>
```

```
        <if a.multiplicity.oclIsUndefined() then>
          object.<mutator_name>(null);
        <else>
          object.<accessor_name>().clear();
        <endif>
    }
}
```

Repository

A repository provides at its basis a similar function to a factory; it enables the creation of objects (in fact the repository uses the factory to provide this part of its implementation). However, the repository keeps track of all objects created and facilitates operations such as: saving its set of objects – to provide persistence for the model population; returning the set of all objects in the model that conform to (are instance of) a particular type; or deleting objects.

As with the factory pattern, we provide repositories on a per package basis, which are linked in accordance with the package hierarchy. Each Package registers a repository for its sub packages and registers a population for its own classes. The template is shown below:

```
public class <pkg_name>Repository extends RepositoryImpl {
  public <pkg_name>Repository() {
    super.setFactory( new <pkg_name>Factory(log) );
    <for each subPackage>
     super.registerSubRepository("<spkg_name>",
              new <spkgFullName>.<spkg_name>Repository());
    <for each class in this package>
     super.registerElementType("<cls_name>");
 }
 public void saveXMI(java.lang.String fileName) {
   super.saveXMI(fileName, new <pkg_name>VisitorImpl());
 }
 public java.lang.String toString() { return "<pkg_name>";}
}
```

The saveXMI method calls the generic saveXMI method, passing a bespoke package visitor. This visitor encodes the manner in which the model elements and their parts should be traversed in accordance with the original model specification.

Visitor

The visitor pattern is one that supports traversal of an object graph that forms the population of a model. The standard visitor pattern provides an implementation of a technique known as ‘double dispatch’. This enables a particular method to be called based on the runtime type of two objects (as opposed to the basic method call that depends on the runtime type of a single object). The two objects are typically: one that is the object being visited (known as the host); and one that is providing a particular piece of behaviour (known as the visitor).

Implementation of the standard visitor pattern provides a visitor interface. This interface typically contains a visit method for each object type for which it provides behaviour. Implementations of the interface visit methods are called indirectly by

calling an accept method on the host object, which takes the visitor implementation as a parameter, this accept method subsequently calls the visit method on the passed visitor using itself as a parameter. The behaviour of *visit* methods typically perform some actions specific to the purpose of the visitor and call accept methods on the objects that form the next hosts in the traversal order.

As with the factory and repository patterns, providing such a visitor for the whole model is not scaleable, or useable if we wish to provide a visitor implementation for only a portion of the model. Thus we also split our implementation of the visitor pattern by package and link them according to the package hierarchy.

2.4 Visitor Implementations

We have found that much of the implementation of tools to support a particular metamodel is provided by implementations of the visitor interface that navigate the model population according to attributes and the various types of association end – *composition*, *aggregation* and *none*. The specific behaviour actions of the visitor implementations can be characterised by grouping the actions into:

- those related to the host
- those related to the attributes
- those related to each type of association end
- those related to linkage between the host and the attributes or association ends

To support this set of visitor implementations, we have defined an implementation pattern for a model navigation visitor. This visitor implementation takes as an additional argument, an object that carries the actions as described in the list above (the visitActions). The interface for this is as follows:

```
public interface VisitActions {

  Object hostAction( String hostTypeName,
                     Object host,
                     Object data,
                     Visitor visitor );

  Object attributeAction( String modelPropertyName,
                          String modelPropertyTypeName,
                          Class implPropertyType,
                          Class collType,
                          Object implPropertyValue,
                          Object data,
                          Visitor visitor);

  Object associationEndAction( String modelPropertyName,
                               String modelPropertyTypeName,
                               Class implPropertyType,
                               Class collType,
                               Object implPropertyValue,
                               Object data,
                               Visitor v );

  Object aggregateEndAction( String modelPropertyName,
                             String modelPropertyTypeName,
                             Class implPropertyType,
                             Class collType,
                             Object implPropertyValue,
```

```
                              Object data,
                              Visitor v );

  Object compositeEndAction( String modelPropertyName,
                              String modelPropertyTypeName,
                              Class implPropertyType,
                              Class collType,
                              Object implPropertyValue,
                              Object data,
                              Visitor v );

  Object linkAttribute( Object propValue,
                        Object hostValue,
                        Visitor v );

  Object linkAssociationEnd( Object propValue,
                              Object hostValue,
                              Visitor v );

  Object linkAggregateEnd( Object propValue,
                           Object hostValue,
                           Visitor v );

  Object linkCompositeEnd( Object propValue,
                           Object hostValue,
                           Visitor v );

}
```

The actions for each implemented visit method are defined to be those that call methods from the visitActions object interspersed with actions that appropriately navigate the population in accordance with the metamodel definition. E.g.:

```
Object visit( <HostType> host,
              VisitActions actions,
              Object data ) {
  Object node =
           actions.hostAction("<HostType>",host,data,this);
  actions.linkAttribute(
    actions.attributeAction("<HostType>.<attName>",
                            "<AttType>",
                            <AttType>.class,
                            <CollectionTypeOrNull>,
                            host.get<AttName>(),
                            data,
                            this
    ),
    node,
    this
  );
  actions.linkAggregateEnd(
    actions.aggregateEndAction("<HostType>.<endName>",
                               "<EndType>",
                               <EndType>.class,
                               <CollectionTypeOrNull>,
                               host.get<EndName>(),
                               data,
                               this
    ),
    node,
    this
  );
}
```

3 Using OCL as a Template Language

As previously mentioned, we make use of OCL as an (interim) template language. To achieve this we have extended OCL in the following ways:

1. Addition of the '+' operator for String values; this is evaluated as a concatenation of the two string arguments.
2. Addition of facility to construct any (model) object. Achieved using the full type name of the object, with constructor arguments contained in following braces '{…}'. This is similar to the syntax for constructing collection objects.
3. Provision of a File class. This object requires a file name as a constructor argument (directories are constructed if necessary); there are two methods, read and write, which either get the file contents (as a String) or write a String argument to the file.
4. Provision of an Expression class. This class enables evaluation of and OCL String value as an OCL expression; arguments are passed to the constructor and evaluate method to provide the environment (free variables) for the expression.
5. Addition of a new query 'context' for OCL expressions that facilitates multiple context variables.

The first of these additional features enables more succinct composition of string values; the second and third features enable us to create files and directories; and the combination of the second, third, fourth and fifth enable us to call sub templates with appropriate parameters.

The templates, shown in the previous subsection, map to a particular pattern of OCL expression. This pattern starts with a query context, defining the parameter object types (free variables) for the expression. Then a number of let statements are given, which define the specific template variables, based on the parameters. The template text is mapped to a series of String and variable concatenations; and finally the concatenated text is written to a file.

To illustrate this, the OCL template (query expression) for constructing package visitors is given below:

```
context
   self : uml::Model_Management::Package,
   properties : uk::ac::kent::cs::kmf::browser::Properties

query:

let
 root_dir = properties.get('root_generation_directory'),
 dir = ''+root_dir+'/'
       +
       Expression {
         String,
         TupleType( self:uml::Foundation::Core::Namespace,
                    sep:String ),
         uk::ac::kent::cs::kmf::util::File {
            properties.get('templates_directory')
            + '/GetFullName.ocl'}.read()
       }.evaluate( Tuple{self=self, sep='/'} ),
 pkg_name = self.name.replaceAll('[^0-9a-zA-Z_]','_'),
 file_name = ''+dir+'/'+pkg_name+'Visitor.java',
 pkg_fullName
```

```
   = Expression {
         String,
         TupleType(self:uml::Foundation::Core::Namespace,
                   sep:String ),
         uk::ac::kent::cs::kmf::util::File {
            properties.get('templates_directory')
            +'/GetFullName.ocl'}.read()
     }.evaluate( Tuple{self=self, sep='.'} ),
 subPackages = self.ownedElement->
    select(e|e.oclIsKindOf(uml::Model_Management::Package)),
 classes:Set(uml::Foundation::Core::Class)
   = self.ownedElement->
       select(e|e.oclIsKindOf(uml::Foundation::Core::Class)),
 nonAbstractClasses = classes->select(c | not c.isAbstract ),

 result_str =
'
package '+pkg_fullName+';

import uk.ac.kent.cs.kmf.patterns.Visitor;
import uk.ac.kent.cs.kmf.patterns.VisitActions;

public interface '+pkg_name+'Visitor
  extends Visitor
{
'+
nonAbstractClasses->collect( cls |
  Expression {
    String,
    TupleType(
      self:uml::Foundation::Core::Class,
      properties:uk::ac::kent::cs::kmf::browser::Properties),
    uk::ac::kent::cs::kmf::util::File {
       properties.get('templates_directory')
       +'/Class__VisitorMethodSignature.ocl'}.read()
  }.evaluate( Tuple{self=cls, properties=properties} )
+';
'
)->including('')->sum()
+'
}
'

in
 if result_str.oclIsUndefined() then
  'Error generating Visitor interface for - '+pkg_fullName
else
  uk::ac::kent::cs::kmf::util::File{file_name}
   .write(result_str)
endif
```

To aid reading this expression the strings defining the generated code are highlighted in bold and the expression parameter variables and defined template variables are highlighted in italics. The expression illustrates all of the five additions to the OCL language.

4 Providing Tools Based on the Patterns

The set of implementation classes generated by these patterns could be used in a variety of applications. An application that we find to be particularly useful is that of

a Browser. The browser is used to create populations of the model and to evaluate OCL constraints over that population.

4.1 Browser

A browser can be created by piecing together various implementations of Visitor Actions, reference to a Repository, our OCL implementation, and a number of actions for invoking operations on these component parts.

A generic browser implementation has been written that can be used in conjunction with any model and set of classes generated using the templates described above. XMI and HUTN visitor actions are written to facilitate saving a population (others could be just as easily written). A generic XMI reader is provided, enabling a previously saved population to be restored. A set of visit actions that construct a JTree has been written; this provides a tree view on the population of a repository. These parts are linked together in a common application along with our implementation of OCL. An image of the generated browser can be seen in **Figure 1**. The panel on the left shows a JTree view of the metamodel components and the instances of those components in the current population. The Console panel on the right shows (in this image) that the metamodel components have been registered with the generated repository. The OCL evaluation panel shows an Invariant, to be evaluated in the context of the class (with name attribute set to "Library") highlighted on the left.

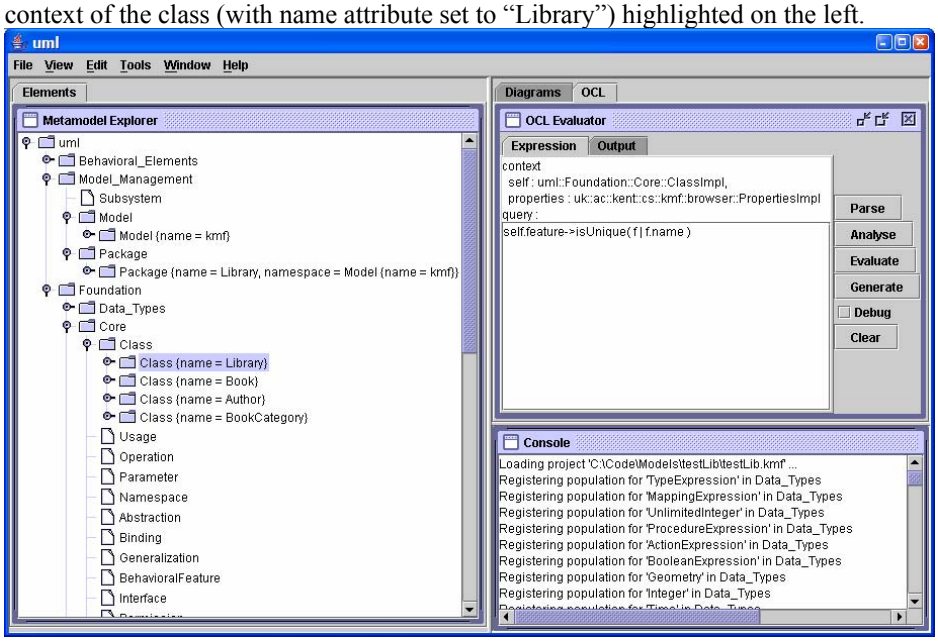

Figure 1 – Generic KMF Browser, browsing a population of UML 1.5 metamodel

5 Related Work

Existing tools such as MetaEdit+ [17] and the Eclipse Modelling Framework (EMF) [18] go some way towards providing similar types of tool to KMF. However they fall short in certain areas. EMF does not support constraints or query

expressions. MetaEdit+ is a more general tool for developing domain specific languages and also does not support OCL.

MetaEdit+ does not directly support generation from class diagrams; it uses its own concepts for metamodel specifications, Graph, Object, Property, Relationship and Role, which are similar to a very small subset of MOF. Metamodels are entered through a series of property boxes, rather than using a visual notation such as class diagrams. The metamodel specification facility enables the definition of a visual language for entering populations of the metamodel (i.e. expressions/specifications in the domain specific language). UML is provided as one of the example domain specific languages supplied with MetaEdit+; however it does not provide an environment that we find easily useable for editing UML models. In addition there does not seem to be a mechanism for adding support for OCL. MetaEdit+ is a commercial tool with development and maintenance support.

Frameworks such as the Eclipse Modelling Framework provide a similar level of generation facility to that provided by our KMF. However, the current release of EMF does not support OCL; we have been working with IBM to provide a version of our OCL library that operates with their EMF generated code. This work has been very successful and we have succeeded in providing facility to:

a) directly evaluate OCL constraints over a population of an EMF model; and
b) generate java code from an OCL expression that when compiled will evaluate the expression.

UMMF - UML Meta-Model Framework [19] is an open source framework written in Perl; it can be used to generate class and interface templates for programming languages Perl and Java; and it will import from XMI versions 1.0 and 1.2.

6 Conclusion

The primary facility offered by KMF, not offered by other tools is the OCL evaluation functionality. In addition, KMF accepts, as input, standard XMI, generated from any appropriate modelling tool (unlike MetaEdit+). Also, KMF provides a fully flexible and user adaptable mechanism for generating code from the provided metamodel specification (possible in MetaEdit+, EMF supports a single, fixed, Java implementation). The KMF tool is based entirely on the concepts and languages of the OMG, making use of XMI, HUTN, MOF, UML and OCL.

This paper has illustrated the significance of patterns as a means to aid the automatic production of tools that to support the specification of a metamodel. The implementation of OCL is used to check well-formedness constraints on populations of the model. In addition by using the OCL as the basis for a template language we have demonstrated that code can be generated from template specifications. The generated code implements standard coding patterns, which are put together to form component parts of a modelling tool.

We show the patterns that we have used within the KMF project and show how we have varied from the standard patterns in order to make the scaleable. We also show the template expressions that will generate a Java implementation for these patterns.

Other work started in [20] and [21] is being continued in order to extend the generated patterns so that they will support an implementation of model transformations. We are looking for a suitable template language, based on OCL, to use instead of directly using our variation of OCL. Additionally, we feel that

providing an implementation in an aspect oriented language such as AspectJ [22] may provide useful facilities for linking the interacting patterns.

Acknowledgements

David Akehurst acknowledges support of the EPSRC project "Design Support for Distributed Systems" (GR/M69500/01) and its investigators J.Derrick and A.G.Waters. Octavian Patrascoiu acknowledges support of the EPSRC project "Reasoning with Diagrams" (GR/R63509/01) and its investigator P.Rodgers.

References

1. OMG: Model Driven Architecture (MDA). Object Management Group, ormsc/2001-07-01 (2001)
2. OMG: Meta Object Facility (MOF) Specification, Version 1.4. formal/2002-04-03 (2002)
3. OMG: Response to the UML 2.0 OCL Rfp (ad/2000-09-03), Revised Submission, Version 1.6. Object Management Group, ad/2003-01-07 (2002)
4. OMG: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. Object Management Group, ad/2002-04-10 (2002)
5. IBM: Rational Rose. http://www.rational.com (2003)
6. Gentleware: Poseidon UML tool, version 1.4. www.gentleware.org (2003)
7. Borland: Together. http://www.borland.com/together/index.html (2003)
8. OMG: XML Metadata Interchange (XMI), v2.0. Object Management Group, formal/03-05-02 (2003)
9. OMG: Human-Usable Textual Notation (HUTN) Specification. Object Management Group, ptc/02-12-01 (2003)
10. KMF-team. Kent Modelling Framework (KMF).[Online]. Available: www.cs.kent.ac.uk/projects/kmf
11. Akehurst, D. H., Derrick, J., Waters, A. G.: Addressing Computational Viewpoint Design. In: Proc. EDOC 2003 (2003)
12. X.901-5: Information Technology - Open Distributed Processing - Reference Model: All Parts. ITU-T Recommendation (1996-99)
13. DSE4DS-team. Design Support for Distributed Systems (DSE4DS) Project Home Page.[Online]. Available: http://www.cs.ukc.ac.uk/projects/dse4ds/index.html
14. Akehurst, D. H., Bordbar, B., Derrick, J., Waters, A. G.: Design Support for Distributed Systems: DSE4DS. In: Proc. 7th Cabernet Radicals Workshop (2002)
15. RWD-team. Reasoning with Diagrams (RWD) project.[Online]. Available: www.cs.kent.ac.uk/projects/rwd
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
17. MetaCase: MetaEdit+. http://www.metacase.com/ (2003)
18. IBM: Eclipse Modeling Framework. http://www.eclipse.org/emf/ (2003)
19. Stephens, K. UMMF - UML Meta-Model Framework.[Online]. Available: http://kurtstephens.com/pub/uml2code/current/htdocs/
20. Akehurst, D. H.: Model Translation: A UML-based specification technique and active implementation approach. University of Kent at Canterbury (2000)
21. Akehurst, D. H., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel. In: Proc. The Unified Modeling Language 5th International Conference (2002) 305-320
22. AspectJ-team: AspectJ. http://www.eclipse.org/aspectj/ (2003)