XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language

Howard Chivers and Richard F. Paige

Department of Computer Science, University of York, York, YO10 5DD, UK
Fax: +44 1904 432767
hrchivers@iee.org, paige@cs.york.ac.uk

Abstract. Efficient tool support for transformations is a key requirement for the industrialisation of MDA. While there is substantial and growing support for unidirectional transformations (e.g., from PIM-to-PSM), for bidirectional transformations there is little. This paper presents tool support for bidirectional transformations, in the form of a language, called XRound, for specifying *reversible templates*. The language supports round-trip transformations between UML models and predicate logic. Its supporting tool also implements *model unification*, so that new information encoded in logic can be seamlessly integrated with information encoded in the model.

1 Introduction

Transformations are a critical component of Model-Driven Development, particularly in the MDA [2]. To this end, the Queries-Views-Transformations (QVT) [1] standard has been developed, in order to provide a precise, flexible mechanism for modelling transformations between models. Even though QVT is still in the process of standardisation, several tools and QVT-compatible (or QVT-like) languages have been developed for supporting the transformation process. Of note amongst these are QVTMerge [3] and the Atlas Transformation Language (ATL) [4], the latter of which provides substantial tool support for model transformation; similarly, XMF [5] provides modelling support for transformations based on an executable dialect of OCL. There are also transformation tools outside of arena of OMG standards; for example, the TXL [7] framework has some similarities to QVT, though it has been predominantly targeted at programming language transformation. The generative programming community has made use of templates to accomplish similar tasks [6], and the meta-programming language Converge [9] has been successfully used to implement a transformation language as an instance of a domain-specific language.

QVT transformations can be *unidirectional* (i.e., from one metamodel to a second, not necessarily new, metamodel) or *bidirectional* (i.e., reversible between two metamodels). The former is of critical use in MDA, e.g., for transforming platform independent models (PIMs) into platform specific models (PSMs). The latter is vital for supporting *rigorous analysis* of models: the results of analysis may need to be reflected in the source of a transformation. For example, a static analysis may be applied to a PSM, resulting in changes being made to that PSM. These changes may need to be reflected in the original PIM.

A. Hartman and D. Kreische (Eds.): ECMDA-FA 2005, LNCS 3748, pp. 205 – 219, 2005.

Limited tool support currently exists for bidirectional transformations; it can be partly supported using sequential application of unidirectional transformations, but this is not entirely satisfactory because information – e.g., diagram layout, detailed representations of platforms – is likely to be lost after each unidirectional transformation is applied.

Related to model transformation technology is *model composition* technology; this is sometimes also referred to as model *merging*, *weaving*, or *unification.* With model composition, two or more models (usually of parts of the same system) are combined into one, in the process resolving inconsistencies, overlaps, and nondeterminism. As of yet, there is minimal language and tool support for model composition; the Atlas Model Weaver [8] is one of the first generic prototypes. Model composition techniques could alleviate some of the problems with using unidirectional transformations for supporting round-trip engineering.

This paper presents a new, template language, called *XRound*, for supporting bidirectional transformations. This language is not QVT-compatible, as of yet, but it uses standard underpinning technology and mechanisms that suggests it could easily be made so. Moreover, the paper presents powerful tool support for this language that allows bidirectional transformations, as well as a form of model unification.

1.1 Context and Contribution

The template language described in this paper arose from the need to support round trip engineering from a specialized analytic tool. The tool, the *Security Analyst Workbench (SAW)*, carries out risk-based security analysis of UML system models, and provides an environment in which the user can interactively set and test security policies. The resulting policies (e.g., access controls) are part of the system design, so they must be re-integrated into the engineering documentation, i.e., the UML models.

SAW does not need the whole of a UML system model on which to perform risk-based analysis; it needs a view that describes certain features of the system (such as classes, operations, and stereotypes), and these are expressed as predicates. For example, *(class,foo)* would identify a class object named *foo*. Predicate representation is the basis of the model unification that we describe in the sequel.

Originally, SAW used *xsl* templates to generate a predicate view from an XMI representation of a UML model. Template processing provided a bridge between tool-specific XMI and the analysis application, which remained UML-tool independent. This allows designers to use their preferred UML environment, and is preferable from the tool software perspective; for example, type checking of security properties is implemented once within the analytic tool, rather than in each UML environment.

Template processing provides an important bridge between different tools, but the available solutions are unable to support the reverse path of unifying the output data back into its original source. Round-trip engineering of analysis results back into the UML is therefore not straightforward with a conventional template processor, but is a significant requirement for specialist analytic tools.

XRound is designed to overcome this problem. Its objective is to maintain the advantages of template processing, including simple scripting of data transformations

and independence between input and output applications, while supporting bi-directional transfers, and unification, of data. This language and its supporting processor allows the SAW application to import UML designs in tool-specific XMI, and re-generates the XMI when the analytic model is changed.

The contribution of this paper is that it describes a new template language with the unique ability to support transformations in both directions, the general principles that underline its design, and a template processor for the new language.

This paper continues by describing XMISource, which is a Java-based processor for XRound. The processor is presented first to clarify how the language will be used, including its straightforward client application interface. The principles behind reversible templates are then introduced, and the structure of the template processing is described. The template language is then presented in detail; the core syntax is first described, followed by two worked examples. Further sections describe language features that support performance management and debugging, and summarise limitations in the current implementation.

2 The Template Processor

The purpose of this section is to clarify the system in which the template language will be used. The first implementation of an XRound processor is a Java class that encapsulates an XML file and allows its client application to important and export predicates from and to the XML source. The design of the processor is given in Fig. 1. Although this is named XMISource after its main application, there is nothing XMI-specific in XRound or in this processor.

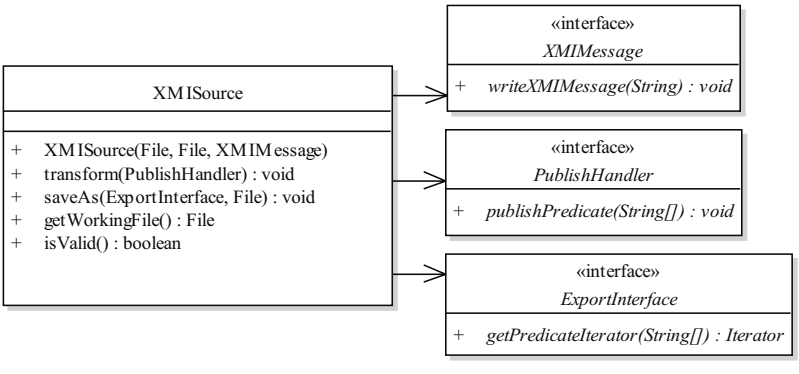

Fig. 1. The XRound Template Processor

The XRound processor has a single class, XMISource, which encapsulates an XML file whose name is provided in the constructor. Three interfaces are defined in the package, and these call-backs are provided by the application client to allow the processor to import and export predicates.

Predicates are represented as arrays of Strings, such as {class,foo}, which describe features in the XML input that are required by the application. The processor supports three transform operations: *validation*, *import* and *export*. (see Template Processing for further detail).

Validation. The XMISource constructor takes three parameters, the reference XML File, the Template File, and a message interface. (The Java File class encapsulates a file name.) The message interface is used to pass certain error messages back to the application, particularly those that report inconsistencies between the template and the XML input. A message interface is used in preference to a thrown exception, since it allows a sequence of messages to be reported during processing, which is valuable during template debugging.

The initialization process parses both the Template and XML input file, and executes a section of the template which is intended to validated the input. Methods are provided to allow the client application to check that the validation was successful (isValid) and to retrieve the name of the XML input file (getWorkingFile).

Import. A single method, transform, runs the import process, which extracts predicates from the XML input, as specified by the template, and publishes them to the client application. As each predicate is constructed the PublishHandler interface provided by the application client is called to transfer the predicate to the client.

Export. A single output method (saveAs) is provided to export predicates from the client application to a named XML file. The output filename is provided by the client, together with an interface (ExportInterface) which allows XMISource to obtain predicates from the application. This is slightly more functional than the other interfaces, but is still straightforward: the client is provided with an incomplete predicate, which is an array of Strings, some elements of which may be null. The client responds with an iterator, which encapsulates predicates matching this template.

The export function updates the reference XML input with predicates obtained from the application, and then writes the result to the named File. File naming strategies and backup files, etc, are implemented by the client application.

Because the input XML is retained, there is no need for the complete XML tree to be exported to the application; the transformation therefore includes only the features required by the application.

The important feature of the template processor is its straightforward client interface; this is a direct result of the reversible template model, since:

- The application only needs to obtain the predicates that it needs for its function, the rest of the input XML remains hidden.
- The application interface is independent of the tool used to generate the XML: any tool differences are accounted for in the template.
- The template includes an explicit validation section that is run at initialisation.

The internal design of the processor is beyond the scope of this paper, but an outline of how the three main operations relate to the template specification is discussed in the next section, before the language itself is given.

3 Template Processing Overview

This section introduces the key concepts behind a reversible template, then describes how the need for the processing operations described above motivate the coarse structure of the template language.

3.1 Bidirectional Transformations and Model Unification

Template processing is usually a one-way operation as shown in figure 2: the template processor locates elements in the input tree and publishes them, suitably formatted.

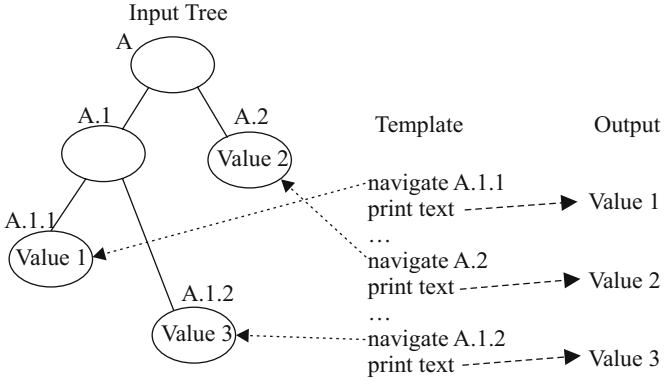

Fig. 2. Conventional Template Processing

In the case of XML data, such as XMI, the input to the template processor is a tree; the output may be XML, or it may be published in another format such as text or HTML. Conventional templates are capable of encapsulating comprehensive programming behaviour, but their fundamental structure is still to navigate to selected nodes in the input tree, extract information, and produce suitably formatted output. The benefit of a template over a standard programming language is usually that it is tailored to the particular type of input and output required.

Reversible templates defined in XRound are similar in structure to existing templates, but encapsulate a fundamentally different type of operation: unification. The operation of a reversible template is shown in Fig. 3.

A reversible template navigates to elements in the input tree, in a similar way to a conventional template, but it also references values in the application predicate. The fundamental operation is to match, or *unify*, values in the source tree with values in the predicate. Unification allows values to be determined from either the source tree, or the application predicate, or if values are set in both, to ensure that they are consistent. For example, in Fig. 3 the first value is not known in the source, but is available in a predicate; the opposite is true for the second value; and the third is the same in both source and predicate, so this unification succeeds.

This underlying unification process determines the design of the template language; as well as carrying navigation information to identify information in the source tree, each part of the template identifies a unification slot, and the fundamental operation is 'match', which is to unify the slot with either the XML input tree, or the application predicate.

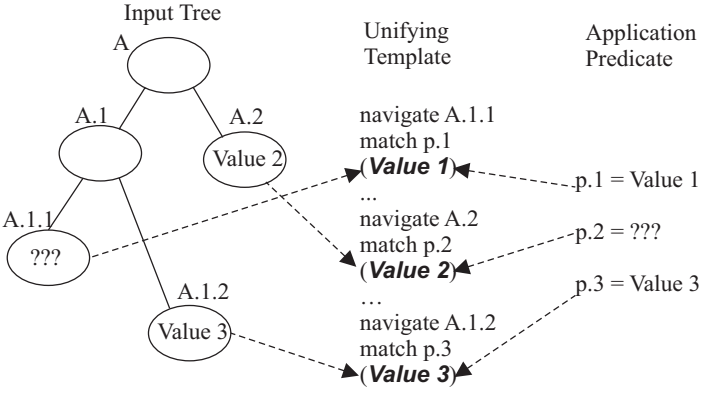

Fig. 3. The Template Unification Process

Unification is conceptually straightforward, but designing a template language that exploits this process does present some problems, including:

- The source navigation for a reversible processor is not quite the same as a conventional template processor, because is has to unify input nodes that do not exist. For example, in Fig. 3 it is not simply the case that the input node does not have the first value set, but that the whole node (A.1.1) is missing. The template language must allow the programmer to specify which nodes are allowed to be missing, and which areas in the source tree are fixed. In XRound, nodes that may be missing are marked as *mutable* and can also be created by the template processor during the process of reverse engineering.
- Because some nodes in the input tree may be missing, it is not straightforward to select nodes based on an attribute value, as is possible in an Xpath expression. In XRound this problem is solved by a general constraint mechanism, which constrains unification slots to specified values. Constraints are also unified as part of the matching process and can therefore be used to specify the types of predicate that can be generated, constrain XML node selection, and determine application predicates to be unified.

The underlying unification process determines some features that are needed in a reversible template language: the definition of unification slots and slot constraints. The next section describes how the main operations of the template processor are supported.

3.2 Template Processing

This section describes the operation of template processing in sufficient detail to introduce the clause structure of the template language.

The previous section described the process of unification, and this places some requirements on the sections, or clauses, of the template language. Essentially a clause must:

- Specify a number of unification slots.
- Allow the specification of constraining values for each slot.
- Unify values in the XML input and/or in application predicates with slot values and constraints.

In order to allow for a separate verification section, and also to allow the user to distinguish parts of the XML input that should be fixed, as opposed to those that may be rewritten, three types of clause are defined in XRound:

- *validate*
- *structure*
- *roundtrip*

A *validate* clause specifies validation checks, a *structure* clause references elements of the XML input that should not be modified, and a *roundtrip* clause includes input nodes that may be modified when the XML is regenerated from application predicates. The value of the *structure* clause is that it allows a wider range of navigation types and some performance optimisations compared to *roundtrip* clauses, because it does not have to account for missing nodes. However, it is not the case that all nodes visited by *roundtrip* clauses can or should be re-written; nodes that can be updated are specifically identified in XRound by a *mutable* attribute.

The three main processing operations can now be described:

Validation. Validation can be used to make any checks that the programmer requires, but its primary aim is to ensure that the template and XML input are compatible. Because XMI is tool specific, a particular template will apply to a limited range of tools and versions; validation clauses in the template are used to check that the input data (e.g., tool type and version number) are compatible with the current template.

After the XMI input and the template have been successfully opened and parsed, each *validate* clause is executed, and each must succeed for the validation to succeed. No other clauses are executed during validation, and the validation clauses are not executed as part of any other processing.

Import. The import operation is similar to normal template processing, it is used to assemble predicates from the XML input and provide them to the client application.

Any *structure* clauses are first executed, followed by *roundtrip* clauses. Each clause is unified with constraints specified within the clause, but not with any application predicates. The clauses have one or more *publish* attributes that mark completion; when these are reached the unification slots within the clause are checked and, if complete, a predicate is exported to the client.

Export (saveAs in XMISource, see Fig. 1). The export operation merges predicates from the client application back into the XML input, then saves the result. The purpose of the operation is to update the XML representation with any changes that have been made by the application, without the need for the application to manage the specific XML format, and without the need to write different templates for input and output processing.

The first processing stage executes all the *structure* clauses in the template; although this will not result in any updates to the XML output, it is necessary because it may build reference information that is used later (see Performance Management, below). There are two further processing stages, the second removes mutable nodes, assuming that nodes no longer present in the application have been deleted intentionally, and the third re-builds nodes from the application predicates. In both cases, the operation (remove, build) takes place only for mutable nodes that have been encountered during a successful unification of a roundtrip template clause. The values written to the rebuilt nodes are obtained from the unification slots in the template, and so may contain values from the application predicates, from the XML input, or directly from clause constraints. The relevant application predicate is obtained when the clause is encountered; essentially, the template processor builds a predicate mask that matches any fixed values specified in the constraint clauses, and requests the application for an iterator over all predicates that match the mask. The clause is then executed once for each predicate in the iterator.

In summary, the process that allows a template to be interpreted in both directions is unification; this has implications for the types of navigation that can be carried out within a template and determines the need for other structure in each clause: unification slots and constraints. The three key operations of validation, import and export are supported by the clause structure in XRound, allowing the programmer to specify validation checks (*validate*), elements of the XML that should not change (*structure*), and parts of the XML tree that can be modified (*roundtrip*).

4 The XRound Language

This section describes the XRound language. It begins by describing how an XRound template is organised in terms of clauses and how they support unification slots, constraints, and transformations. This is followed by a detailed description of transformations, and two examples of template clauses. This section concludes by describing language features that support performance management and debugging.

4.1 Basic Template Structure

The top-level structure is most easily described with an abbreviated DTD:

```
<!ELEMENT tpl.template
((tpl.validate|tpl.structure|tpl.roundtrip)*)>

<!ELEMENT tpl.validate (tpl.constraint*,tpl.specification+)>
```

```
<!ATTLIST tpl.validate  length CDATA #IMPLIED
                      auxLengthCDATA  #IMPLIED>

<!ELEMENT tpl.stucture (tpl.constraint*,tpl.specification+)>
…
<!ELEMENT tpl.roundtrip (tpl.constraint*,tpl.specification+)>
…
<!ELEMENT tpl.constraint (tpl.value+)>
<!ATTLIST tpl.constraint position CDATA #REQUIRED>
<!ELEMENT tpl.value (#PCDATA)>
```

A template is a well-formed XML document containing three node types that may occur in any number and any order: *tpl.validate, tpl.structure* and *tpl.roundtrip.* These are the *clauses* introduced in the previous section. Attributes in each clause node specify the number of unification slots (*length* + *auxLength*) and these are simply indexed as an array in the subsequent template (e.g. *position = "0"*). The slots are divided into two, and the first section (specified by *length*) is mapped directly to an application predicate.

Each clause may have any number of constraints; each constraint has a *position* attribute that specifies the associated unification slot, and a number of values.

A clause therefore specifies the unification space, or number of slots, and gives constrained values to those slots. One or more *specification* nodes in each clause determine the correspondence between the XML input and unification slots in the template, and hence the application predicates.

4.2 Template Specifications

A template specification is well-formed XML, but unlike some template languages it follows a tree structure, rather than a sequence. Depth in the tree indicates subsequent operations and breadth allows the specification of alternatives. A *publish* attribute can appear anywhere in the tree, and its effect is to test that unification is complete, and if so mark that result as successful. For example:

```
<first>
<second tpl.publish="TRUE"/>
<third>
<fourth tpl.publish="TRUE"/>
</third></first>
```

This would find all instances of first…second and first…third…fourth that unified. (first, etc, are not of course valid node names)

There are three types of node in a template specification: Source Nodes, Navigation Nodes, and Matching Nodes. Source and Navigation Nodes may carry the attribute *tpl.mutable="TRUE"* in a *roundtrip* clause. This specifies that the node that can be removed or re-written when predicates from the client application are exported back into XML.

Source nodes simply name a node in the XML input tree. They cause the template to evaluate all nodes of that name from the current position in the XML input.

At present the language supports five types of navigation statement, two of which are concerned with performance management. Examples of the three core types are:

```
<tpl.select node="UML:ClassifierRole">

<tpl.selectFromChildren
          node="UML:AssociationEnd" position="0">

<tpl.moveUp steps="2”>
```

The *tpl.select* node evaluates all nodes in the input tree with the specified node name, the example selects all *UML:ClassifierRole* nodes in an XMI tree.

The *tpl.selectFromChildren* node is intended to select child nodes from the present position in a specified order. Each occurrence of *tpl.selectFromChildren* specifies the position (i.e. index) and name of the child node to be selected. In this example the first occurrence of a UML:AssocationEnd node is selected.

The *tpl.moveUp* node simply moves the present position in the XML input tree up by a number of steps. This command has been included because it provides a very compact way of navigating certain tree structures, but there is a restriction on its use: it must never follow a mutable node. Nodes marked as mutable can unify with nodes that are not present in the XML input; allowing a step up from such a node may be non-deterministic, depending on how the mutable node has been reached, so this navigation is not permited from a mutable node.

There are three matching node types within the template language, and they each instruct the template processor to unify an element in the XML input tree with one of the unification slots, any previously specified constraints and, depending upon the process mode, a predicate retrieved from the client application. Examples of these are:

```
<tpl.match nodeType="ATTRIBUTE_NODE"
                      attribute="name" position="1">

<tpl.match nodeType="TEXT_NODE" position="0">

<tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                      attribute="myLunch‟  tagIndex="1‟
                      length="2‟ position="3" >
```

Each *tpl.match* node specifies the index of the unification slot that must be matched (*position*). The relationship between the unification slots and the client predicates is fixed, so this does not need to be specified. The node to be matched from the XML input is always the current node, reached by the last navigation. The first two match types unify the value of an attribute by name, or node text data, respectively. The third is more specialized and provides the ability to pack several parts of a predicate into a single XML attribute.

A multiple attribute match node unifies one value in an attribute list of separated values. For example, given the attribute *mylunch=”fish,chips”* , the example above would correctly match the number of values in the attribute (length=”2 “) and attempt to unify the value ‘chips’ (*tagIndex=”1”*) with the template slot 3.

This is the core of the reversible template language. A small number of specialized language statements are omitted from this paper for reasons of space, they include the creation of xmi.ids and additional forms of constraint. Language features to support

performance management and debugging are described below, but first the essentials of the language will be illustrated by some worked examples.

4.3 Examples

This section provides two examples of template clauses, which demonstrate how well the template language is able to hide the round-trip complexity. The first example is a complete *structure* clause:

```
<tpl.structure length="2">
   <tpl.constraint position="0">
       <tpl.value>data</tpl.value>
       <tpl.value>service</tpl.value>
   </tpl.constraint>
<tpl.specification>
   <tpl.select node="UML:Class">
   <tpl.match nodeType="ATTRIBUTE_NODE"
                        attribute="name" position="1">
   <UML:ModelElement.stereotype><UML:Stereotype>
   <tpl.match nodeType="ATTRIBUTE_NODE"
        attribute="name" position="0" publish="TRUE"/>
   </UML:Stereotype></UML:ModelElement.stereotype>
   </tpl.match></tpl.select>
</tpl.specification>
</tpl.structure>
```

There are two unification slots in the template, and these correspond directly to a client predicate with two values. The constraint section of this clause limits the first slot position to the values ‘data’ or ‘service’.

The specification searches all the nodes in the XML input for UML:Class nodes. For each node of this type it extracts the name attribute, which is unified with the second unification slot position. The template then searches child nodes for the stereotype (UML:ModelElelement.stereotype/UML:Stereotype) and it unifies the attribute name of the stereotype with first unification slot. Of course, this slot is constrained, so the only values that succeed are ‘data’ or ‘service’. The effect of this clause, therefore, is to search the XML input for UML:Class nodes with stereotype of ‘data’ or ‘service’ and, depending on mode, publish predicates of the form (datalservice,name). The form of this template is very similar to other template languages, demonstrating that although reversible templates are theoretically quite different to conventional templates, their programming form can be made familiar.

The specification of mutable XMI nodes is essentially the same. The following is part of template clause for the Security Analyst Workbench:

```
<!—Slots:
   (tagname 1st_value className 2nd_value)(xmi.id)  -->
<!—Client use:
   (PermitAccess fromClass inClass toOperation)     -->

<tpl.roundtrip length="4" auxLength="1">
   <tpl.constraint position="0">
   <tpl.value>PermitAccess</tpl.value>
   </tpl.constraint>
   …
```

```
<tpl.specification>
   <XMI><XMI.content><UML:TaggedValue
                                        tpl.mutable="TRUE">
   <tpl.match nodeType="ATTRIBUTE_NODE"
                    attribute="tag" position="0">
     <tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                    attribute="value"
                    tagIndex="0" length="2" position="1">
      <tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                     attribute="value"
                     tagIndex="1" length="2" position="3" >
      <tpl.match nodeType="ATTRIBUTE_NODE"
                    attribute="modelElement" position="4" >
     <tpl.selectNode node="UML:Class">
      <tpl.match nodeType="ATTRIBUTE_NODE"
                     attribute="xmi.id" position="4">
      <tpl.match nodeType="ATTRIBUTE_NODE"
                     attribute="name" position="2"
                     publish="TRUE">
      …
```

The comments at the start of this extract describe the use of the unification slots and the resulting application predicate. This template matches an XMI tag, which is attached to a UML class. The name of the tag is 'PermitAccess' and the tag has two separated values (e.g. *PermitAccess="subject,object"*). The application predicate contains the same information as the tag, but also includes the name of the class in which the tag was declared (inClass). The first four template slots correspond to the values in the application predicate, and the fifth is used for the xmi.id of the class. The header to this clause specifies the number of unification slots, and constrains the first to the single value 'PermitAccess'.

The specification navigates directly from the document root (XMI) to a tagged value, which is marked as mutable. This specifies that any tagged values that match this clause will be re-written on export. This navigation identifies all possible tagged values, but only those that unify as far as the 'publish' tag at the end of this fragment will be rewritten.

The next three match statements unify the three elements of the tag (name plus two values) with their respective template slots. An important feature of this language is that the programmer is not concerned with the underlying operations. These statements are able to both extract data from the XMI tree and publish them to the client application, and also obtain predicates from the client and re-write it into an XMI tag, depending upon the operational mode of the template processor.

The fourth match operation unifies the modelElement attribute value with an auxiliary slot in the unification template (i.e. one that is not part of the application client's predicate). This value is the xmi.id of the class in which the tag is placed, and the next section of the template navigates to the corresponding class by selecting all the class nodes in the XML input, and matching the one with the correct xmi.id. The final match statement unifies the class name associated with this xmi.id with the third template slot. At this point the publish attribute tests if the unification process is complete, causing publication to the client, or the addition of a node to the XMI tree.

This fragment illustrates the extent that the underlying semantics of unification and reversible working are hidden from the template programmer, who is still able to think of the template as little more than a 'select and publish' script.

One notable feature of this fragment is the relative lack of constraint checking. In the Security Analyst Workbench, the two values in the tag are known types, the first corresponding to a class with a specific association to the class in which the tag appears, and the second to an operation within that class. It would be quite straightforward to navigate the XMI input tree and use the unification process to check that these values correspond to correct types. However, there are good reasons for avoiding these checks at this stage. Firstly, the template is specific to the tool that generated the XML input, but given that the template processor delivers tool-independent predicates, the type checking could be coded once, in the application, rather than separately for each supported tool. There is also a second consideration, which is that in its normal operation the template processor will often fail to unify, since it will attempt to match nodes and predicates that are not compatible. If constraint checking is included in the template, then badly constructed types will not unify, and will not be passed to the application. However, the result of a constraint failure in a template processor is silence, whereas constraint failures in the application can generate warnings to the user. The programming philosophy adopted has therefore been to specify the minimum in the template language, consistent with establishing an accurate relationship between the XML input and application predicates, and to carry out more extensive type checking in the application.

These examples illustrate the core language, two further issues, performance and debugging, add extra features, which are discussed in the next sections.

4.4 Performance Management

The main performance problem in template processing is the need to repeatedly scan all the nodes in a document. This problem occurs in the *roundtrip* example above. It is necessary to scan the entire document for UML:Class nodes, in order to match the xmi.id in the tag with the correct class name. Since Classes are in user-defined packages they can occur at any level of the XMI hierarchy, so it is not feasible to limit the search size by navigating from the tree root.

However, in UML templates, the types of node that are revisited often in this way are a relatively limited number of fixed design points, primarily the classes and objects. If it were possible to simply remember the location of these nodes then these auxiliary searches could be made much more efficient. This, quite simply, is what the performance management statements in XRound implement. There are two statements, one that records fixed points, and one that navigates to previously recorded nodes. For example, in the two examples above, the first, which identifies specific Classes, could include the statement:

```
<tpl.registerNode>
```

This registers the current node (in this case UML:Class), allowing it to be efficiently revisited later. The second example could then replace the selectNode navigation to a UML:Class with:

```
<tpl.selectRegisteredNode node="UML:Class">
```

The result is the same, but considerably faster. The only restriction on the use of these statements is that mutable nodes cannot be registered, and that nodes must be registered before they can be selected. Normal practice is to register nodes in early *structure* clauses, they can then be referenced anywhere in the template.

4.5 Debugging

Finally there are two important features in the language that are an aid to debugging – message and debug attributes – which can be added to any node.

The message attribute (*tpl.message="…")* can be included in any node, and sets a message for the template tree below that node. If any errors are issued from the processing of that section of the template, then the message will be included in the error report. It is good programming practice to include messages in every clause header; they provide useful comments and invaluable narrowing of the problem space when an error is reported.

The debug attribute (*tpl.debug="TRUE"),*is not intended to be a permanent feature of a template. Whenever a node is encountered with the debug attribute, the status of the unification slots is printed, together with the current template and XML nodes. Although this provides sufficient information to debug a template, usually the existence of debug output is sufficient: when a template fails the most common problem is detecting the node that caused the failure, so the most common use of this feature is as a probe to detect where a template succeeds or fails.

5 Limitations

There are few inherent limitations in the XRound language; the practical limits arise from two sources: variability in XMI between different UML tools, and limits to the scope of the current template processor.

Differences in XMI between UML tools was one of the main motivating factors in the design of XRound, and has been discussed at several points in the paper; different templates are required for different UML tools, but the use of a reversible template isolates the application logic from this variability. The XMI import behaviour of tools can also vary in detail; for example, some tools regenerate missing xmi.id fields, where others fail. The design of a template may therefore go beyond the need to understand the XMI tool dialect. Although this is an inconvenience, it has not yet proved a major problem, or required tool-specific language features.

At present the XMIsource processor is more limited than the language. Although it supports all the language features it does not support unification of nodes that are interdependent: for example it would not be able to unify both missing classes and associations between those classes. This is not an inherent limitation in the XRound language, but reflects the initial applications for XMISource, in which the class structure is stable but other elements of a model can be varied by analysis tools.

6 Conclusion

XRound adds a new dimension to template processing of models: the ability to transform data in both directions with a single descriptive template. Reversible template processing solves the problem of maintaining independence between UML and analytic tools, while retaining the benefit of easily scripted transformations. Reversible templates could provide a clean implementation mechanism for bi-directional transformations specified in QVT, and could help in the definition of model unification languages as well.

This paper has outlined the theory behind reversible templates, and presented a mature template language, XRound, that has been used in practice and is supported by a Java template processor. As well as including transformations the language includes performance management and debugging facilities.

The examples presented here illustrate the extent that the underlying semantics of unification and reversible transformation are hidden from the template programmer, who is still able to think of the template as a 'select and publish' script.

The successful implementation and use of a template processor demonstrates that it is feasible to implement a reversible processor that interprets the XRound template language, and the specification of the processor shows how straightforward the reversible interface is from the perspective of the client application.

References

1. Object Management Group. *Queries-Views-Transformations Specification*, available at http://www.omg.org, last accessed June 2005.
2. Object Management Group. *Model-Driven Architecture Specification,* available at http://www.omg.org, last accessed June 2005.
3. QVT-Merge Group, *Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)*, 2004, www.omg.org.
4. ATLAS Transformation Language web page. http://www.sciences.univ-nantes.fr/lina/atl/, last accessed June 2005.
5. Xactium Inc. *XMF Reference Guide 0.1,* available at http://www.xactium.com, last accessed June 2005.
6. J. Herrington. *Code Generation in Action*, Manning, 2004.
7. TXL Web Page, available at http://www.txl.ca, last accessed June 2005.
8. M. Del Fabro, J. Bezevin, F. Jouault, E. Bertan, G. Guillaume. AMW: a Generic Model Weaver. In *Proc. IDM 2005*, July 2005.
9. L. Tratt. *The Converge Programming Language*, King's College Technical Report TR-05-01, 2005.