Write Once, Deploy N: a Performance Oriented MDA Case Study

Pieter Van Gorp, Dirk Janssens
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
{pieter.vangorp,dirk.janssens}@ua.ac.be

Tracy Gardner
IBM Hursley Development Laboratory
IBM Hursley, Winchester, United Kingdom
tgardner@uk.ibm.com

Abstract

Optimizing the performance of distributed database applications is difficult to combine with middleware vendor independence since cache, transaction and cluster configuration is database and application server specific. The most promising approach is to use a flexible code generator. The Model Driven Architecture can become a successful standard for model based code generation by offering a set of high quality code generation languages. To focus the comparison of such languages on criteria that matter in practical development, there is an urgent need for more, and more realistic, case studies. Therefore, we present a complex middleware performance pattern that should be generated for at least two application servers. The code generator should have maximal reuse, separation of concerns and evolution support. From this case study, we derive the requirements for model refinement and code generation languages. We illustrate how existing UML standards can be integrated to satisfy these requirements and conclude with an overview of the language features that were needed to provide an elegant solution to the case study.

1 Introduction

In model-driven software engineering, the primary artifact for development are models, rather than conventional source code. Complex knowledge about best practices and deployment technologies is shared by using models of recurring patterns as first class programming artifacts. Code generators from domain specific modeling languages to pattern and deployment aware models and eventually to Java, C#, or XML, ... make sure that best practices are implemented with minimal effort. Additionally, adherence to architectural style rules is enforced by model consistency checkers. Domain specific modeling languages can have a visual and/or textual concrete syntax.

Code generation reduces the time spent on low-level coding and removes the possibility to make programming mistakes in implementing recurring code structures. The abstract models assists in software understandability and the style checkers support the manageability of evolving software.

Within the context of distributed database application development, the amount of code duplication due to framework constraints is enormous. Moreover, different database and application servers expose conceptually equivalent performance tuning services, yet in a syntactically different way. Hence model-driven software engineering is a promising approach for this application domain: recurring framework and performance patterns can be described in a (relatively) platform independent modeling language. Different versions of databases and application servers are automatically supported by mapping the platform independent models to more platform specific models and eventually to deployable code.

Whereas source code is input to a black-box compiler, models are input to white-box consistency checkers and transformation tools. Hence the need for developer-friendly languages for interacting with such tools. Without flexible transformation languages, architects cannot adapt their code generator to the unique constraints and opportunities of their enterprise application integration scenario. Consequently, they are still dependent on the platforms (operating systems, databases and/or application servers) imposed by the default mappings of their code generator. OMG has standardized on the OCL for model constraint evaluation and is currently working on standard languages for transforming models to other models [20] and to source code [21]. Several model transformation languages have been submitted in response to OMG's Query / Views / Transformations (QVT) request for proposal but criteria for objectively comparing these alternative languages are still premature. For example, several industrial submitters have conflicting opinions about the declarative nature of a model transformation language [13].

Without realistic case studies, it is not clear what language criteria really matter in practical MDA development. In this paper, we present the problem of developing distributed database applications that are optimized for concurrent data access. The problem constraints are to avoid lock-in on vendor extensions of a particular J2EE application server [6], to make a proper separation of concerns, and to enable tool support for domain evolution.

This paper is organized as follows. Section 2 introduces our running example: a performance oriented middleware pattern. Section 3 derives the requirements for a transformation language for model refinement and code generation. In Section 4, we illustrate how existing UML standards can be integrated to satisfy these requirements by developing parts of a code generator for our case study with a conceptual transformation language. We conclude this paper by generalizing the early results of our case study to desirable properties of transformation languages for practical MDA development and by discussing future work.

2 Write Once, Deploy N (WODN)

Distributed server components are deployed on an application server that delivers the middleware services from a platform like J2EE or .NET. These services are configured by attaching deployment attributes to the component sources. After inheriting from the appropriate component model classes (or interfaces), the remaining Java or C# code can focus on business logic, rather than non-functional aspects such as transaction demarcation, persistence, caching, and clustering.

2.1 Towards Portable Server Applications

The J2EE platform makes the distinction between vendor independent and vendor specific deployment attributes. The design goal "Write Once, Deploy Anywhere" has been accomplished for the vendor independent deployment descriptor and the Java source files. However, any realistic server component will require the usage of vendor specific files with at least some network distribution information and in most cases an object-relation mapping before it can be deployed on an application server. The good news is that this information tends to be very similar across all the components of a server application. Along with the repetitive structure of their Java sources, this makes J2EE components a natural candidate for code generation.

Today's MDA tools have built-in code generators for the leading server component models and their application servers [5, 8]. They reduce complexity and initial development time by providing reasonable defaults for most deployment properties. Kleppe et al. [17] describe a mapping for Enterprise JavaBeans that characterizes how exist-

ing MDA tools generate code for server components: each platform independent entity will eventually have one corresponding server deployment. Application server migration comes down to generating default deployment information for the new (version of the) product.

2.2 Performance Issues

While the default deployments generated by today's MDA tools are extremely useful for rapid prototyping on different application servers, it is a waste of server resources to run them as such in a production environment. Still, this antipattern occurs more than one may expect, even outside the case of generated systems.

Tyler Jewell [16] investigated this issue and observed that the reason for this performance problem is that a default deployment must have its cache and transactions configured conservatively for concurrent read-write data access while production systems tend to have as much as 85% read-only data access, only 10% read-write data access and 5% batch-update access.

A solution that is often suggested is to limit the use of the EJB component model to transactional read-write data access and take a shortcut to the database layer in the case of read-only or batch-update data access [25]. As with most performance hacks, this approach spoils the integrity of the overall architecture.

Jewell demonstrated that the problem can also be solved without dismissing the EJB component model. The proposed "Write Once, Deploy N times" (WODN) pattern makes optimal use of performance features like distributed caching that are already implemented in all leading application servers. By deploying the same Java sources multiple (n) times on the same server (or cluster), each client type can be served with a deployment that is optimized for its data access scenario. Clients that read data from an entity and write back new values in the same request will be served by a deployment configured for aggressive loading, while clients that directly overwrite the values of existing entities will be served by lazy loading deployments. Similarly, clients that only read data will be served by a deployment that is configured with transaction settings that are much looser than those of the write deployments. Depending on product and version, the application server can either send selective invalidation messages from the write deployments to the read-only cache or flush the read-only cache on a regular interval.

2.3 Goal

It has not yet been investigated how performance patterns, such as WODN, can be efficiently implemented simultaneously on different application server products. We

will investigate how model-driven engineering techniques can reconcile performance optimization with portability, separation of concerns and tool support for domain evolution. This paper should provide enough detail to adapt existing MDA tools for achieving these goals.

3 Transformation Language

In this section, we start from a concise taxonomy of model transformation to subsequently derive domain specific language constraints for the different kinds of transformations.

3.1 Model Transformation Taxonomy

Model transformation can be roughly divided into two subdomains: rephrasing and translation [27].

Rephrasing is transforming a program into a different program in the *same* language. The source and target metamodel of the transformations are the same. *Refactorings* (i.e. restructurings for object-oriented software) are probably the most widely known examples of rephrasings [10]. A *restructuring* is the transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics) [7]. Note that with a narrow definition of functionality, program optimizations are restructurings as well. Moreover, the aspect of behavior preservation is not a transformation language property, but a transformation specification property.

Based on the above definitions, we can define a *translation* as the transformation of a program across different metamodels. In program *refinement* (and its inverse *analysis*) a chain of translations is applied with metamodels at different abstraction levels. Translations are also applied to maintain consistency between specifications in different languages, yet at the same level of abstraction. In this context, Schürr [23] has demonstrated how correspondence rules between the graph grammars of two such languages can be used for bidirectional consistency maintenance.

3.2 Language Requirements for Solving WODN

Languages for rephrasing need to support model transformations within the same metamodel, whereas languages for translation need to support mappings between abstract and concrete metamodels. Graph rewriting [22] is an interesting formalism for specifying restructurings as its formal theory is a promising basis for correctness proofs [18] and existing tools can be used to execute the constraint and transformation specifications [26]. In Figure 1, the transformation of the Pull Up Method refactoring is specified in the
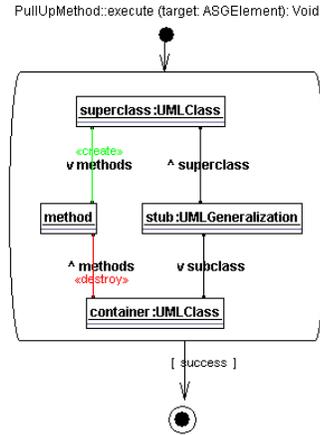

Figure 1. An executable refactoring specification in the Story Driven Modeling language. The execute method of the "The Pull Up Method" class has one parameter, "target", representing the model element to which the transformation applies. The node "method" contains this parameter after casting it to UMLMethod. The edges "methods", "subclass" and "superclass" specify a pattern that checks if the containing class has a superclass. If so, the link to this class is destroyed and a link to the superclass is created.

UML / graph rewriting language called Story Driven Modeling [11].

As illustrated by our case study, languages for refinement play a more crucial role in the generative MDA process: how can we specify an automated transformation from a platform independent business model to a detailed component specification that is optimized for heavy server load?

1. It is desirable that different aspects of the refinement process can be specified independently. This decomposition of transformations enhances their understandability and reusability. Moreover, our case study requires a domain metamodel that has no notion of the performance pattern, neither from other aspects such as object to relational mapping. We need a language to build automated translations between all models describing the system from different viewpoints and levels of abstraction.

   Therefore, we would expect the transformation language to support mappings across different metamodels. Note that one could technically combine all these domain specific metamodels into one large auxiliary metamodel. In this case, metamodel packages would

be required for avoiding ambiguity among model elements from different viewpoints and levels of abstraction. For understandability, we use the multi-metamodel approach in the remainder of this paper.

By using strongly typed metamodels with associated well-formedness rules as the in- and output of transformations, one can compose transformation primitives with both static and dynamic validation. The types of in- and output metamodels constitute signatures for transformation primitives. After statically verifying whether the output metamodel of one transformation matches the input metamodel of the subsequent transformation, one can dynamically evaluate well-formedness rules to check whether each transformation produces valid instances of its output metamodels.

2. It must be possible to parameterize the refinement process *between* the abstract and concrete metamodels (instead of *on* one of them).

Kleppe et al. [17] illustrate this with an object/relational transformation: the parameter that defines the length of a VARCHAR that is derived from a UML string does neither belong to the source (UML) nor target (RDBMS) metamodel. The former wrongly assumes that all future refinements of the object model will need to decide on the database length of a string. The latter requires to annotate the target models with the transformation parameters. This has a negative impact on their readability. Storing transformation parameters in attributes from dedicated transformation instances appears to be the best approach, since it makes the configurability of a code generator explicit through its transformation models.

3. Application servers from the same component model often extend the standard with the same conceptual services. However, these services are configured by different deployment parameters, in different files. It must be possible to model the specialization of this common transformation behavior in the refinements for these similar application servers.

This is especially relevant in the context of the J2EE platform where one is confronted with a practical boundary on the level of standardization that vendors are able and willing to commit. Although each generation of middleware provides first class support for the common usage patterns of the previous generation, middleware extensions first need to be tried in practice before they can be standardized. Our case study illustrates how MDA provides a way of encoding such patterns for use on today's middleware, rather than waiting for the next generation. Finally, patterns that are overly specific to a particular context will never be supported by generic middleware.

4. A refinement language should support the resolution of evolution conflicts where possible. Among the possible scenarios are manual manipulation of generated artifacts and integration with other software engineering tools.

Ideally, a refinement language allows variability in the conflict resolution, such that tools can either resolve a conflict behind the scenes or prompt for user interaction. The transformation writer should be able to configure what strategy should be employed. When a transformation requesting automatic resolution is deployed to an engine, the latter should only accept this transaction if it can fulfill the contract! If an engine cannot generate conflict resolution code from the transformation specification, it should be able to accept extra code from the transformation writer and execute it when a particular conflict is detected. This approach allows transformation writers to develop transformations with automatic conflict resolution even for complex cases where an engine could not offer this service. It may be a good strategy for building lightweight implementations of the QVT standard rapidly and extend the standard as more and more tools catch up.

5. It should be possible to query transformation specifications stored in a "refinement repository" [26]. This is required when, for example, a Rename Entity refactoring is executed on a business model. After regenerating all derived models and code, one needs to execute a series of primitive Rename Class refactorings for updating all the manual code that makes use of the generated classes. We want to get the relationship between Class names and Entity names as a result of querying all transformation specifications.

Note that the Rename Entity refactoring could be implemented more easily if all model elements sharing a particular property would be stored in a common repository and point to a shared value. In such an architecture, only the shared value would have to be updated to implement the discussed refactoring. However, since reality dictates that practitioners use different tools for requirement engineering, design and implementation, we cannot assume that all models of a given software product are stored in a common repository.

Requirements 4 and 5 are very difficult to achieve with strict imperative transformation languages since the relationship between source and target metamodels is too implicit in that case.

6. Transformation specifications should be repository independent. This can be achieved by applying the MDA principle of code generation from platform independent models to transformation models themselves. As an initial experiment, we have defined a UML profile for specifying refactorings as graph rewritings (similar to Figure 1.) We use the AndroMDA tool [5] to drive code templates (model to code transformations) that generate repository access code from instances of the profile. Figure 2 displays a part of such a code template we are developing for JMI based repositories.

JMI [14] is the Java implementation of the MOF standard for repository access, which should make our transformations already executable on a number of repositories. However, we can extend our repository support to non-MOF repositories like EMF [19] by adding a new set of code templates.

3.3 QVT Success Criteria

OMG's MDA initiative needs a standard language for implementing rephrasings and translations (i.e. transformations within one or across more metamodels) [20]. Past research illustrated that refactorings (being a particular kind of rephrasings) can be implemented using a general purpose programming language like Java, potentially generated from graph rewriting rules in UML syntax [26].

In the context of the WODN case study, we investigated the requirements for a refinement language. Again, refinements can be implemented in a general purpose programming language using libraries for hiding repository details. The advantage of object-oriented languages is their proven support for parameterization and reuse with specialization (inheritance with overriding). Therefore, we suggest that the upcoming QVT specification for refinement standardizes a small declarative language that is complementary to existing general purpose languages. If this language supports services like traceability automatically, it can avoid a lot of tedious manual programming and should become succesful. Finally, this approach can only work if transformations written in the refinement DSL integrate seamlessly with transformations written in mainstream languages like Java. This can be realized by applying the concept of pluggable code generation to the refinement DSL itself.

4 Generating WODN components

As mentioned in Section 2.3, our goal is to use model-driven engineering techniques to reconcile performance optimization with portability, separation of concerns and tool support for domain evolution. In this section, we illustrate how existing UML standards can be integrated to satisfy

```
1  ## Velocity Template
2  ##
3  import java.util.*;
4
5  import javax.jmi.model.MofClass;
6  import javax.jmi.reflect.RefPackage;
7  import javax.jmi.reflect.RefClass;
8
9  /**
10  * Code generated by JCMTG
11  */
12  public class $class.name {
13
14  ...
15    /**
16    * Find JMI Class Proxy with specified name in
17    * specified package (or subpackage).
18    */
19    private RefClass jcmtg_findClassProxy(String name,
20                             RefPackage pkg) {
21      Collection classes = pkg.refAllClasses();
22      for (Iterator it = classes.iterator();
23                       it.hasNext(); ) {
24        RefClass c = (RefClass) it.next();
25        if (((MofClass) c.refMetaObject()).getName()
26                               .equals(name))
27          return c;
28      }
29      // not found in package
30      // now look in subpackages
31      Collection subpkgs = pkg.refAllPackages();
32      for (Iterator it = subpkgs.iterator();
33                       it.hasNext(); ) {
34        RefClass c = jcmtg_findClassProxy(name,
35                          (RefPackage) it.next());
36        if (c != null) return c;
37      }
38      return null;
39    }
40
41  #set ($transOps=$transform.getTransformationOperations(
42                               $class))
43  #foreach ($transOp in $transOps)
44  #set ($transFlow=$transform.getTransformationFlow(
45                               $transOp))
46  #parse ("templates/TransFlow.vsl")
47  #end
48  }
```

Figure 2. Code Template fragment. Any ordinary text, such as the Java code from lines 3 through 11 (or 15 through 40), will be copied verbatim to the generated file. Line 12 shows how properties from model elements can be accessed with the $ notation. Lines 41 through 48 illustrate the use of scripting commands for assignments, control flow and delegation to other scripts. The template is taken from a code generator that translates UML based graph rewriting rules to JMI code and is intended as an example of code template languages only. Its exact meaning is outside the scope of this paper.

the transformation language requirements we have derived by presenting parts of a conceptual code generator for the "Write Once, Deploy N" pattern. We focus on generating all vendor specific artifacts from an abstract business model. Pattern and vendor independent client access (from which a model checker derives whether a read-only, read-write or batch-update deployment needs to be called) is outside the scope of this paper.
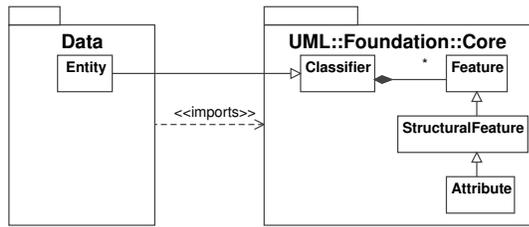
4.1 Designing PIM, PSM & Intermediate Metamodels

Figure 3. The Data metamodel.

Figure 3 presents the metamodel for the pattern and vendor independent specification of models from the problem domain. It does not add any properties to the standard UML model elements. Each "Entity" should be automatically refined to an EJB component following the WODN pattern.

At the next level of abstraction, we want to specify the transaction and caching attributes for the three participants in the WODN pattern. Figure 4 displays the metamodel for specifying such models, without locking in on concrete vendor attributes. This enables us to reuse the pattern refinement across code generators for different application servers.

Another aspect of our application domain is persistence. Let us consider the mapping to a relational database characterized by the metamodel in Figure 5. The persistence aspect can be refined independently from the transaction and caching aspect.

The next level of abstraction joins the above two aspects and maps them to a concrete application server. Instead of building a full-fledged metamodel for each application server and refining our models one more time, we choose to transform the models to text. This text is input for xDoclet [28], a popular code generator that transforms annotated Java files into all component sources of the leading J2EE application servers.

We thus reuse the mappings from a compact bean description to the remote and local bean interfaces, their abstract factories (remote and local home interface) and possibly a primary key and data transfer class. We hereby il-
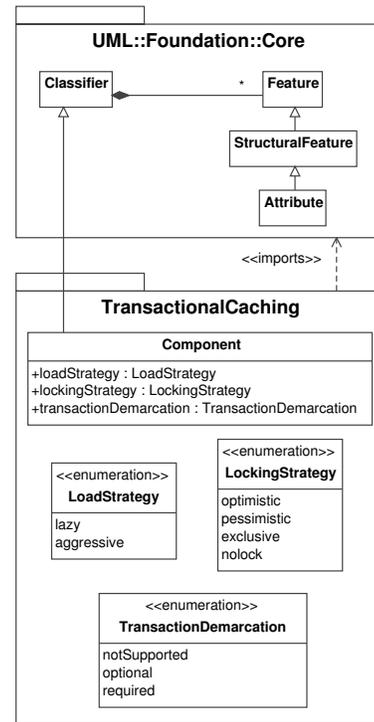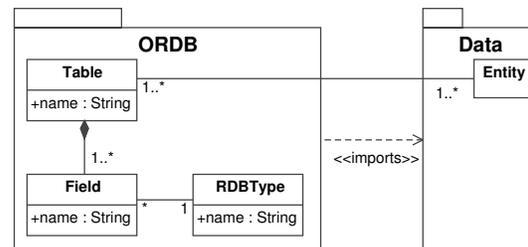

Figure 4. The TransactionalCaching metamodel.

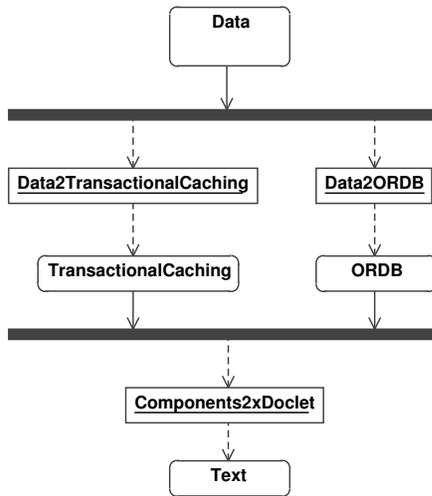**Figure 5. The ORDB metamodel.**

Figure 6. The overall refinement process for the "Write Once, Deploy N" pattern.

lustrate that model transformations can be introduced incrementally. On the one hand, we already benefit from the ability to reuse, specialize, and separate concerns in the upper part of our code generator. On the other hand, we can start testing the complete code generation process rapidly by relying on the robust code generation library of xDoclet. Still, requirements 4 to 6 from Section 3.2 could only be supported by replacing xDoclet by a code generator that generates intermediate models representing the standard EJB artifacts as well. The problem with xDoclet is that it does not expose standard information on traceability and the relationship between models and code.

Figure 6 displays a control flow diagram of the proposed refinement from pattern and vendor independent models to concrete component files. The notation is based on UML activity diagrams. Models are transformed from one metamodel to the other. Metamodels are displayed as regular states (with rounded rectangles) and transformations are displayed as object flow states (with regular rectangles). Using synchronization bars, one can visualize the independent aspects of the refinement process.

4.2 Transformations as Domain Specific Classes

Transformations can have more than one input model. Consider for example the Components2xDoclet transformation on Figure 6. Generally, transformations that join (or "weave") different aspects require one input model per aspect (or "view"). It is also possible that a transformation has multiple output models. Although not strictly required

to solve the case study, it may be desirable to output an automated text description as documentation on a model refinement. In such scenario, users can zoom in to the next level of models or zoom to the text description. If a transformation takes two models conforming to the same metamodel as input, one should still be able to distinguish them. Hence we cannot simply refer to "the" input or output models by mentioning their metamodel. Finally, to realize traceability, a transformation should be able to have a (persistent) state. Therefore, one can apply the principle of object orientation to transformations: a transformation can be represented as a class with a set of typed attributes for storing in- and output models. Although primitive transformation rules can map from one input model to one output model, this does not hold for larger transformation components that maintain the consistency within one or between two or more models. Our proposed transformation language does not distinguish between transformation attributes holding in- or output models since the role of these models may vary in different parts of the synchronization process.

4.3 Rules maintaining Consistency Invariants

If transformations are represented as objects with the in- and output models stored in their attributes, we can extend the notion of an operation to represent behavior. Based on requirement 4 that states that *a refinement language should support the resolution of evolution conflicts where possible*, we propose to integrate consistency checking with model transformation and code generation. This can be done by assuming a one-to-one correspondence between consistency invariants and the postconditions of transformation rules. As soon as a consistency invariant fails, a particular transformation rule should be executed to re-establish consistency between in- and output models. Code generation is the initial case of the top-down version of this inductive algorithm where only high-level models exist and the low-level models need to be generated in order to establish the consistency invariants. Parsing is the initial case in the opposite direction.

Figure 7 illustrates how specific OCL constraints can be used to specify the relationship between the abstract and concrete metamodels in a declarative manner. The rule states that for all entities in the abstract domain (Data), there should be a corresponding read-only component in the concrete domain (TransactionalCaching). The rules for the read-write and batch-update components follow the same approach, yet set other values for the deployment attributes. The rule from the Data2ORDB transformation can be specified declaratively as well: for each entity in the Data domain, there should be an entity linked to a table with the appropriate fields in the ORDB domain.

```
1  Transformation Data2TransactionalCaching {
2   // models as structural features (attributes)
3   Data d; // metamodel as static type of model
4   TransactionalCaching tc;
5   // rules as behavioral features (operations)
6   Rule Entity2RO_Component () {
7    postcondition: // a specific invariant is established
8     d.Entity.allInstances->forAll(e |
9      tc.Component.allInstances->exists(c |
10      e.Classifier = c.Classifier and
11      c.lockingStrategy = LockingStrategy::noLock and
12      c.transactionDemarcation =
13           tc.TransactionDemarcation::optional
14     )
15    )
16   }
17   ...
18 }
```

Figure 7. The Entity2RO_Component rule from the Data2TransactionalCaching transformation. By using specific OCL constructs, such as exists, one can rely on the transformation engine to automatically satisfy the constraint.

The invariants can be monitored by the transformation engine and trigger a semi-automatic resolution process for failed assertions. The degree of automation depends on the phase in the software lifecycle. If the *exists* predicate of the constraint fails in the first iteration of the development cycle, the engine can simply instantiate an object that satisfies all the equations.

Line 9 of the transformation fragment in Figure 7 shows a syntactic shorthand for stating that all the fields of a common superclass should be equal. A transformation engine can either store these fields twice and register an observer [12] on both to propagate their values to one another or make them explicitly point to a shared value. The former enables transformation users to confirm or reject the actions by which the engine proposes to rectify occurring inconsistencies. The latter strategy is appropriate when automatic bidirectional consistency maintenance is desirable.

Since the OCL constraints play both the role of transformation invariants and rule postconditions, one could argue for replacing the *postcondition* keyword by something like "*maintains*". However, this paper focusses on highlighting relevant semantical concepts for existing QVT submissions instead of propsosing yet another concrete syntax.

4.4 Traceability Models

A traceability model consists of a set of tuples that maintain the relationship between abstract and concrete model elements. These tuples allow users to browse from abstract to concrete concepts and vice versa after the transformations have been applied. These tuples should also be acces-

sible by the transformation developer through specific language constructs: if the transformation engine cannot automatically derive corrective logic for all inconsistency scenario's of a transformation, then the transformation writer should be able to add reconciliation code manually. In such reconciliation code, one should be able to access properties from all participants in the transformation.

Czarnecki and Helsen [9] observed that among the languages with dedicated tracing support, some languages require the transformation developer to create traceability links manually whereas other languages create them automatically. Considering our declarative transformation rules, model elements created by a transformation engine reacting to a failing *exists* predicate can be automatically associated to a traceability model element. For example, when the rule on Figure 7 would fail, an engine would automatically create corresponding Component instances for all Entity instances without a previously created link. Additionally, the traceability link would be established. For all component instances that already have such a traceability link, all equations of the consistency constraint need to be carefully examined in order to determine what fields of the two related model elements need to be corrected. A custom reconciliation observer, written by the transformation developer, may be appropriate for such cases.

Access to the in- and output models is provided through the instance variables of the transformation. Access to model elements that the engine created to establish an "exists" predicate can be provided through the names of the OCL iterator variable.

4.5 Integrated Code Templates

Model (to model) transformation is currently one of the most active points of research in the MDA space. It will improve the power of existing MDA tools substantially [24]. Due to the lack of proper model refinement techniques, early MDA tools are forced to use template languages to transform too abstract models directly into code. This leads to too much complexity in the templates, as illustrated by Figure 8.

The template is responsible for generating J2EE entity beans, which fits well within the WODN case study. The fragment illustrates the complexity resulting from a too abstract input metamodel. In this case, one used the general purpose UML metamodel as template input. Note that in the emerging third generation of AndroMDA, one is trying to tackle this problem by decorating metaclasses with platform specific properties. One uses Java to implement the accessors of so-called *metamodel façades* (MMF [4]). A MMF encapsulates platform details that belong to one model element of the PIM (which is a UML profile equivalent to the Data metamodel presented in Section 4.1). In

```
1  #foreach ( $op in $class.operations )
2  #if ($transform.getStereotype($op) == "FinderMethod")
3   * @ejb.finder signature="${transform.
4     findFullyQualifiedName($op.getType())} ${transform.
5     getOperationSignature($op)}" #set($viewtype = "")
6  #set($viewtype = $transform.findTagValue(
7                         $op.taggedValues,
8                         "@andromda.ejb.viewType"))
9  #if($viewtype == "local" || $viewtype == "remote"
10                        || $viewtype == "both")
11  *      view-type="$viewtype"
12 #end
13 #set($querystring = "")
14 #set($querystring = $transform.findTagValue(
15            $op.taggedValues, "@andromda.ejb.query"))
16 #if($querystring == "")
17 #set($querystring = "SELECT DISTINCT OBJECT(c) FROM
18                                 $class.name AS c")
19 #if($op.parameters.size() >0 )
20 #set($querystring = "${querystring} WHERE")
21 #foreach($prm in $op.parameters)
22 #set($querystring="${querystring} c.$prm.name =
23                                 ?$velocityCount")
24 #if($velocityCount != $op.parameters.size())
25 #set($querystring = "${querystring} AND")
26 #end
27 #end
28 #end
29 #end
30  *      query="$querystring"
31 #end##if op.stereotype = "FinderMethod"
32 #end##foreach operation
```

Figure 8. Fragment from EntityBean.vsl, a code template from the AndroMDA code generator [5]. The complexity of the control flow should be moved from this code template to a model transformation.

```
1  abstract Transformation Components2xDoclet {
2   TransactionalCaching tc;
3   ORDB o;
4   Text code;
5   Rule Generate() {
6    postcondition:
7     tc::Component.allInstances->forAll(
8      c | o::Entity.allInstances->forAll(
9       e | (c.Classifier = e.Classifier)
10         implies (
11          code::File.allInstances->exists(
12            // ignore directories in this example
13            f | f.name= (c.name + ".java") and
14              f.content= Join2JavaFile(c, e)
15          )
16         )
17      )
18    )
19  }
20
21  Rule Join2JavaFile (TransactionalCaching::Component c,
22                ORDB::Entity e): Text::String {
23   // code template fragment for Java imports
24   // code template fragment for conventional Javadoc
25   #call Join2ClassTags(c,e);
26   ...
27   // code template fragment for iterating over methods
28   ...
29    #call Join2MethodTags(c,e);
30   ...
31   ...
32  }
33
34  abstract Rule Join2ClassTags (
35                TransactionalCaching::Component c,
36                ORDB::Entity e): Text::String {
37   // code template for vendor independent xDoclet class
38   // tags like @ejb.finder, @ejb.home, @ejb.interface,
39   // ...
40  }
41  ...
42 }
```

Figure 9. Fragment from the Components2xDoclet transformation.

our solution to the WODN case however, we decided to decompose the refinement process by using several domain specific metamodels instead of using one large decoration of the UML metamodel.

An important observation from the AndroMDA tool is that code template languages still provide a very intuitive means to generate code from models at a low level of abstraction.

Therefore, we propose to integrate model transformation languages with code template languages, as in Figure 9 and 10 which are explained in the next section. A code template can be considered as an imperative rule: instead of specifying a match pattern, it is explicitly called by other rules. Therefore, code templates can only access model elements through their formal parameter list. Code template rules do not have a postcondition but do require a return type. It may be desirable to standardize the Text metamodel as a basic type of the QVT/M2T language.

```
1  abstract Transformation Components2WL
2   inherits Components2xDoclet {
3
4   // Join2JavaFile inherited, not overriden
5
6   Rule Join2ClassTags (TransactionalCaching::Component c,
7                ORDB::Entity e): Text::String {
8    #call super.Join2ClassTags(c,e);
9    // code template for WebLogic specific xDoclet class
10    // tags like @weblogic.persistence, @weblogic.cache,
11    // @weblogic.invalidation-target, ...
12   }
13   ...
14 }
```

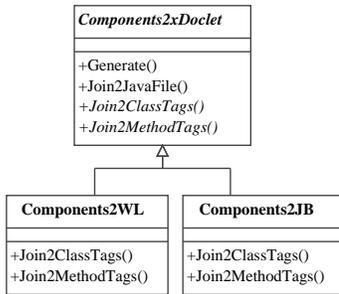**Figure 10. Fragment from the Components2WL transformation.**

Figure 11. Inheritance hierarchy for the Components2xDoclet transformation.

4.6 Platform Specialization

In the last step of the transformation process, `Components2xDoclet` maps information from the `TransactionalCaching` and `ORDB` metamodels to an input file for the xDoclet code generator. The idea behind xDoclet is that deployment attributes are specified in special documentation tags within a source file, instead of in separate XML deployment descriptors.

The vendor lock-in problem with xDoclet is that the deployment attributes are still completely vendor specific and there is no dedicated language or framework to refine user-defined properties to these low-level properties. We will illustrate how this problem can be overcome by using a transformation language supporting polymorphism.

In our case study, we need to map the vendor neutral properties from the enumerations in the `TransactionalCaching` domain to concrete properties for our target application servers, say JBoss [15] and BEA WebLogic [1].

Figure 11 displays how the abstract `Components2xDoclet` transformation is specialized by a concrete transformation for each target application server. These vendor specific transformations implement the `Join2ClassTags` and `Join2MethodTags` rules by outputting the xDoclet tags that realize the properties from the transactional caching and persistence aspects on WebLogic and JBoss respectively. Compared to conventional inheritance, transformations play the role of classes while rules play the role of methods. Figures 9 and 10 are intended to give an idea of the conceptual structure of the abstract transformation and the concrete transformation for WebLogic and JBoss.

The Generate rule on Figure 9 weaves the two independently refined models into text by stating that for all combinations of Component and Entity that are derived from the same Classifier, an input file for xDoclet should be generated. The imperative rule Join2JavaFile defines the overall file structure whereas the polymorphical calls to Join2ClassTags and Join2MethodTags realize platform specialization.

The call to the parent rule on line 8 of Figure 10 illustrates how code templates for specific platforms like WebLogic can reuse fragments from platform independent code generation templates. Lines 9 through 11 indicate that the rest of the code template are WebLogic specific.

5 Conclusions and Future Work

Within the MDA, model transformations should be treated as first class citizens. Therefore, it is important to use a simple yet powerful language to design and implement such transformations. In this paper, we presented a complex middleware pattern as a realistic case study for model-driven development. From this case study we derived concepts of a transformation language that would allow us to generate such patterns towards different application servers.

The language is complementary to mainstream model management frameworks like JMI and supports the specialization of common transformations, the separation of concerns, and evolution conflict resolution. For the latter feature we applied the familiar concept of design by contract. Following this principle, code generation can be integrated with architectural consistency checking: consistency constraints can be considered as invariants that need to be maintained by transformation rules. Failed invariants trigger a reconciliation unit whose postcondition is equal to the invariant.

The body of reconciliation units can often be generated from the declarative OCL postconditions. For new components, reconciliation will result in code generation. Evolution conflicts in existing software require reconciliation units that correct the inconsistencies without regenerating other items. For this purpose, such units can analyze source and target models by navigating traceability models that maintain a persistent link between these two. Since conflict resolution logic cannot always be generated by the transformation engine, transformation specifiers need language support to implement it manually.

By careful inheritance from common superclasses in the design of metamodels for PIMs and PSMs, one can specify consistency constraints across abstraction layers in a very concise manner.

At a low level of abstraction, imperative code templates can be integrated with the model (to model) refinement process. Using an object-oriented approach, code templates can be overriden to support platform specialization. Note that the overriding of model to model transformations is not investigated in this initial version of the WODN case study. It would be worthwhile to investigate how one can syntac-

tically prevent (or at least statically check) that parent and child rules have conflicting postconditions.

We can already derive some preliminary conclusions on how model-driven code generators can be modeled. Activity diagrams can elegantly visualize the parallelism in the refinement process that is needed for the separation of concerns. Visualizing transformation inheritance hierarchies as class diagrams is useful to make variability in the refinement process more explicit.

We will adapt the ATLAS Transformation Language [2] with the proposed features to validate these conclusions in practice.

Acknowledgments

References

[1] BEA Systems. Weblogic server, 2004. Available online: http://www.bea.com/products/weblogic/server/.

[2] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[3] J. Bézivin and R. Heckel. Dagstuhl seminar 04101 - language engineering for model-driven software development, February 2004. Schloss Dagstuhl, Germany. Available online: http://www.dagstuhl.de/04101/.

[4] M. Bohlen. Metamodel Facades, July 2004. Available online: http://team.andromda.org/tiki/tiki-index.php?page=MetaModel+Facades.

[5] M. Bohlen et al. AndroMDA, February 2004. Available online: http://andromda.sourceforge.net/.

[6] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *Anti Patterns – Vendor Lock-In*, chapter 6, pages 167–176. Wiley Computer Publishing. John Wiley & Sons, 1998. Available online: http://www.antipatterns.com/vendorlockin.htm.

[7] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[8] Compuware. OptimalJ, February 2004. Available online: http://www.compuware.com/products/optimalj/.

[9] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[10] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, and T. Mens. A discussion of refactoring in research and practice. Technical report, University of Antwerp, January 2004.

[11] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 5, pages 293–303. Professional Computing Series. Addison-Wesley, 1995.

[13] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard, 2003.

[14] Java Community Process. Java metadata interface (JMI) specification – JSR 000040, June 2002. Available online: http://java.sun.com/products/jmi/.

[15] JBoss Group. The JBoss application server, 2004. Available online: http://www.jboss.org/.

[16] T. Jewell. Unlocking the true power of entity EJBs, 2001. Available online: http://www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html.

[17] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, 2003.

[18] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.

[19] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[20] Object Management Group. MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10, October 2002. Available online: http://www.omg.org/cgi-bin/apps/doc?ad/02-04-10.pdf.

[21] Object Management Group. Draft MOF Model to Text Transformation RFP ad/2004-01-13, January 2004. Available online: http://www.omg.org/cgi-bin/apps/doc?ad/04-01-13.pdf.

[22] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

[23] A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.

[24] S. Sendall and W. Kozaczynski. Model transformation - the heart and soul of model-driven software development. *IEEE Software, Special Issue on Model Driven Software Development*, 20(5):42–45, 2003.

[25] B. A. Tate and B. R. Flowers. *Bitter Java*, chapter 8.4.4 and 8.4.5. Manning, 2002.

[26] P. Van Gorp, N. Van Eetvelde, and D. Janssens. Implementing refactorings as graph rewrite rules on a platform independent metamodel. In H. Giese and A. Zündorf, editors, *Proceedings of the 1st International FUJABA Days*, pages 17–24. University of Kassel, Oktober 2003.

[27] E. Visser et al. Program-transformation.org – a taxonomy of program transformation, February 2004. Available online: http://www.program-transformation.org/Transform/ProgramTransformation.

[28] xDoclet Team. xDoclet, February 2004. Available online: http://xdoclet.sourceforge.net/.