A new tool for URDAD to Java EE EJB Transformations

Craig Edwards / Stefan Gruner
Department of Computer Science
University of Pretoria
South Africa
craig@tuple.co.za / sg@cs.up.ac.za

ABSTRACT
Following the Object Management Group’s (OMG) Model-Driven Architecture (MDA) approach, the semi-formal, service-orientated “Use Case, Responsibility Driven Analysis and Design” (URDAD) method is used by requirements engineers to specify a software system’s functional properties in a Platform Independent Model (PIM). PIMs are represented using the URDAD Domain Specific Language (DSL), and thus conform to the URDAD MOF meta model. As a result, they can be transformed into Platform-Specific Models (PSM) for frameworks such as Java Platform Enterprise Edition (JEE) Enterprise Java Beans (EJB). This paper describes the semi-automatic transformation of a URDAD PIM into a EJB PSM, which is the basis for the further generation of EJB program code. For this purpose, a new prototype CASE tool was implemented to facilitate such transformations. The tool was evaluated using a non-trivial example project, with results indicating that it produces the PSM and template code that constitutes the static Java EE EJB structural representation of the example PIM.

Categories and Subject Descriptors
I.6.5 [Simulation and Modelling]: Model Development

Keywords
URDAD, MDA, EJB, Model transformation, CASE tool, MOF, JaMoPP, QVT

1. INTRODUCTION
High software failure rates are still prevalent. Various software development approaches still struggle with the intricacies of requirements management [19], platform interoperability, and system evolution [20]. Abstraction is a technique used to simplify software design, implementation, and visualisation, assuming the form of models that provide schematic descriptions of system characteristics.

Model-Driven Engineering (MDE) relies on models to drive the software development process, aiming to increase productivity, to improve the prospect of automation, and to simplify development by raising the level of abstraction [12]. MDE is used to develop artefacts, such as business process designs and the software architectural characteristics. However, with the typical 20 years from theoretical research to industrial production, MDE is still in an early stage of industrial adoption [13]. Particular challenges are: assessment of model quality, insufficient representation of run-time properties, continuing deficiencies in requirements capturing, standardisation and benchmarking for CASE tool support, as well as shortage of useful and precise languages for domain-specific modelling.

To standardise the use of MDE, the Object Management Group (OMG) recommends the Model-Driven Architecture (MDA) approach [16]. MDA focuses on the definition and transformation of Platform-Independent Models (PIM) into Platform-Specific Models (PSM) and eventually into concrete software artefacts such as executable code. MDA aims to separate a system’s functional requirements, defined using PIMs, from its other (quality) requirements [20]. PSMs are used to model the characteristics of the platforms selected to implement the system. The MDA standards that are relevant to this paper include:

Meta-Object Facility (MOF): a 4-layered hierarchal architecture used to describe and construct meta-models. The top ‘M3’ layer represents the model that defines the MOF language itself. The ‘M2’ layer is used to build meta models that ‘M1’ layer models must adhere to. The ‘M0’ layer is used to represent instances of ‘M1’ models.

XML Metadata Interchange (XMI): an XML-based format which is used to represent MOF models.

Object Constraint Language (OCL): a constraint and query expression language that can be used on any MOF meta model or model.

Query/View/Transformation-Relations (QVT-R): a declarative language to express Model-to-Model (M2M) transformations.

Unified Modelling Language (UML): the de-facto standard modelling language that provides diagram types for describing object-oriented systems.
While becoming a popular development approach [20], MDA is also still immature. Adoption rates of many OMG standards are only slow, and tool support is still inadequate. UML’s vagueness impedes consistent model definition, and competig model transformation approaches are difficult to compare. Economically MDA requires a long-term perspective on return on investment. There is also no precise definition of what actually constitutes an MDA PIM [23]. A systematic analysis and design method, which can be used to capture a system’s functional requirements as a PIM, is still lacking, too.

URDAD, the semi-formal, service-orientated “Use Case and Responsibility-Driven Analysis and Design” method [10] [23] [24] [22] [21] attempts to leverage the benefits associated with MDA, whilst seeking to address some of MDA’s above-mentioned issues, particularly to improve the manner in which requirements are captured. URDAD is already used in the regional software industry, while URDAD’s formalisation is still the topic of ongoing research. URDAD prescribes a systematic step-by-step recipe to be carried out in an “engineering manner" in order to arrive at a system’s PIM design.

However, transformation tool support for URDAD has been lacking. Such lack has impeded its further industrial adoption and motivated the work described in this paper. Once URDAD PIMs are defined by requirements engineers, these PIMs have to be manually converted into program code, thus negating the benefits of automation and repeatable engineering processes, which are associated with the use of MDA. To this end, a prototype tool was successfully developed [4] to facilitate the semi-automatic transformation of URDAD-constructed PIMs into Java Enterprise Edition (EE) Enterprise Java Bean (EJB) PSMs, which may then be used to construct Java EE EJB code. This tool is the subject of our paper.

Each resultant PSM contains the model elements that are required to produce Java code representing the static structures of an EJB encoding of an URDAD-compliant system design. The tool is also capable of generating Java code from these PSMs. The EJB structures that are produced by the tool include full Java class hierarchies representing all the data structures defined in a PIM. Stateless EJB Session Beans are generated to depict each URDAD namespace, including method definitions on these beans that represent the services that are associated with a namespace. Tests with a non-trivial example use case showed that our prototype can carry out the desired model transformations as expected.

2. RELATED WORK

In spite of their slow acceptance in the software industry, MDE, MDA, and model transformations are popular topics of research in academia. For this reason, and because of lack of space in this paper, we cannot discuss all related work comprehensively here. Nevertheless, the following references are particularly relevant in the context of our work.

Already 20 years ago, Börstler and Janning [2] emphasised the importance of defining relationships between the model elements involved in a transformation. In those days, the term ‘document transformation’ was common. The project of [2] focused on the transformation of requirements specification diagrams into a system’s design or architecture diagrams. Entity Relationship Diagrams (ERDs) and Data Flow Diagrams (DFDs) were used for that purpose. Initially the documents did not explicitly conform to meta models. In those days, meta models had to be constructed ‘ad hoc’ and were deduced intuitively by the comparison of the structural features of the diagrams themselves. Transformation traceability was a concern, and an additional mapping diagram had to be used to track the correspondences between source and target models. The transformation described in that paper was not automatic and depended on occasional user intervention. The work of [2] can be regarded as pioneering and laid the conceptual foundations for many later document or model transformation projects [14] [5].

Transformations of documents from the Business Process Model and Notation (BPMN) format into XML are described in [8]. The BPMN is an OMG standard that provides a graphical notation that can be used by requirements engineers to define business process models. It is similar to URDAD in that it can be used to specify business processes in a platform-independent manner. However unlike URDAD, BPMN does not prescribe a service-orientated system design. The authors describe their BPMN models as Computation-Independent Models (CIMs) which, in the context of MDA, are models that are used to specify system requirements at an even higher level of abstraction than PIMs. Unlike in our work, each CIM is transformed into an XML-based PIM in [8]. Both types of models conform to their own MOP meta models. For transformations the QVT-like ATLAS Transformation Language (ATL), facilitated by an Eclipse-based ATL plug-in, was chosen.

The project described in [18] introduces a method for reverse-engineering UML use case diagrams from Java programs. A Java PSM, based on their own Java MOF meta model, is generated from Java code, which is then transformed into a PIM that represents the use case diagrams. The PIM is based on the official OMG UML MOF meta model. Compiler techniques are used to convert the Java code into an abstract syntax that conforms to the Java meta model. The rules that dictate how the PSM elements are mapped to the PIM elements are defined as OCL constraints.

A similar project proposes the use of MDA methods to automatically produce optimised business process specifications from business rules [25]. Business rules are defined using the OMG’s Semantics of Business Vocabulary and Business Rules (SBVR), and are represented using a PIM that conforms to a SBVR MOF meta model. A textual syntax based on the SBVR Structured English (SBVRSE) notation is used to define the rules and to produce the PIM. The optimised business processes are specified with BPMN. The final result of such transformation is a PIM that conforms to the BPMN MOF meta model. The business rules are optimised during the transformation process, which is horizontal and exogenous, and tool-supported by an Eclipse-based ‘XTend’ plug-in.

Other authors [11] have studied the transformation of an Ecore-based meta model into an Eclipse Modelling Framework (EMF) Generator Model (GenModel). They focused on a CASE tool’s ability to support transformation change propagation and traceability, which is also an important property for our work. Eclipse provides a function that produces GenModels from Ecore meta models. This requires input from the tool’s user. However, when these meta models are changed existing GenModels may become inconsistent with their corresponding Ecore meta models. A solution is proposed that entails the use of annotations on the Ecore meta model to capture user requirements.
GenModels can then be automatically produced by transformation. The use of the Component Object-based Software Architecture (COSA) to raise the level of abstraction while developing or maintaining a system, is proposed in [1]. COSA’s purpose is to describe systems as components and their dependencies. Such specifications are then automatically converted to EJB 2.0 implementations using MDA transformation techniques. The COSA and EJB 2.0 meta models are defined with UML profiles. These profiles are created using UML stereotypes and OCL constraints. The element mappings between the two meta models are described with ATL, and the Eclipse-based ATL tool is used to do the transformations.

Last but not least we mention a project in which MDA techniques are used to generate distributed program code from sequential programs [9]. Distributed systems rely on runtime support systems such as CORBA, COM or RMI. Consequently, mapping problems arise which can be solved with model transformation techniques. Like in our project, model transformation and code generation was done with a propriety tool. The tool converts the program code into a PIM, defined using a meta modeling language provided by the Generic Modelling Environment (GME), which is then used to define the distributed representation. Once a specific platform has been selected, the tool transforms the PIM into a PSM, and then uses the PSM to generate the distributed program code.

3. TOOL REQUIREMENTS

The purpose of our tool is to support the transformation of URDAD PIMs into Java EE EJB PSMs, which are the basis for the further production of EJB code. There are prima facie many approaches that could be followed to achieve this goal. We now outline some of the requirements and constraints that influenced our tool’s design and construction. Many details, described in [4], have been omitted due to paper length constraints.

In MDA, reference is made to the use of models to create lexical information (text) such as program code, or documentation. However, direct PIM-to-text transformation lacks flexibility because it is tightly bound to the particularities of the resultant text format. The use of an intermediate PSM is preferred, and is comparable to the intermediate representations of code in the field of compiler construction. For this reason our CASE tool relies on for PIM-to-PSM transformation, with an additional PSM-to-text transformation as a second phase.

Following the MDA approach, a unidirectional transformation strategy was chosen. Unlike with a compiler, user interaction is required to provide platform-specific information about the target system (PSM) that cannot be inferred from the functional requirements at the higher level of abstraction (PIM).

The OMG maintains several standards that serve as a methodical basis for MDA. Given URDAD’s industry-orientation and close association with MDA, preference was chosen to be given to languages and formats such as MOF, XMI, OCL and QVT, for which software development environments are widely available.

The PIMs that are the source (input) of our transformation tool originate from an URDAD-guided requirements engineering phase, and are constructed by following the stepwise URDAD engineering recipe. URDAD not only prescribes the steps to be carried during in this phase, but it also stipulates the type of model elements that may be used. The main building blocks in URDAD are: name spaces (“domains of responsibility”), services at various levels of abstraction, contracts (pre- and post-conditions) which services must fulfill, as well as further auxiliary modelling elements to denote data types, etc. URDAD PIMs must conform to the URDAD DSL meta model [21], and our tool must produce a PSM representation for all applicable URDAD PIM model elements.

The target model for the transformation needs to be a PSM representing the structure and semantics of a Java EE EJB application. Therefore, an Ecore meta model supporting the latest version of the Java Language Specification was required. For this purpose we re-used the Java Ecore meta model that is part of the Java Model Parser and Printer (JaMoPP) project [7]. It can be used to abstract Ecore models from Java source code, or to produce Java source from Ecore models. JaMoPP was also chosen because of its association with EMFText which was also used for the URDAD DSL.

Five key relationships were identified between the source and target models. They cater for an almost complete transformation of all URDAD elements, excluding the algorithmic details of the URDAD service elements, for which only the signatures of Java methods are generated. These methods have to be manually implemented. There are order-of-creation dependencies between some of the relationships. For example, target ‘Session Bean’ methods cannot be produced until all source data structures used for method parameters, return types and exceptions, have been transformed. As described in [4], all in all we mapped: URDAD data structures to Java Beans, URDAD features to Java Bean fields or to Java Bean accessor and mutator methods, URDAD responsibility domains to Session Beans, and URDAD service contracts to Session Bean method signatures.

Figure 1 illustrates the conceptual mapping of a data structure to a Java Bean. There are three types of Java Bean classes that are used to represent data structures. ‘Exception’ data structures are created from service contract preconditions, while persistent data structures are mapped to EJB Entity Beans. If both an exception and an Entity Bean are inappropriate for representing a data structure, then they are mapped to a ‘standard’ Java Bean. If a data structure has a super type, if it is abstract, or if it references other classes, then these characteristics will be represented in the resultant Java Bean class, too.
The process is initiated upon receipt of an invoice, and the system ensures that the student complies with course prerequisites. The service is responsible for the creation of an invoice, and the successful termination of the service.

Three URDAD views are used to illustrate the specification of the enrollForPresentation service. Figure 3 shows the service’s contract and the static structures of the its request and result objects. The EnrollForPresentationRequest object contains three identifiers identifying the Presentation and the Client who is responsible for payment. The EnrollForPresentationResult contains an Invoice, a ProofOfEnrollment and a StudyGuide. Note the preconditions and postconditions that are associated with each service.

Figure 4 illustrates the three lower-level services that are used by the enrollForPresentation service, as well as the service providers [3] that offer each of these services. The checkStudentSatisfiesEnrollmentPrerequisites service ensures that the student complies with course prerequisites, the issueInvoice service is responsible for the creation of an invoice, and the performEnrollment service enrolls the student for the presentation.

Figure 5 depicts the enrollForPresentation service’s full process specification, including the use of the lower-level services. The process is initiated upon receipt of an enroll-

---

**Figure 2:** Persistent data structure to an Entity Bean (element mapping)

Figure 2 shows the mapping of a persistent DataStructure element of the PIM to the PSM elements that are required to represent an Entity Bean class. This results in a single CompilationUnit element and an associated Class classifier element. Note the naming of these elements and their namespace, as well as the references to the Public modifier and the EJB Entity AnnotationInstance elements. A ClassifierReference element has been used to indicate the Entity’s extension of the Classifier element that represents the base abstract Entity super class that all Entity Beans extend. The complex structural characteristics of the four remaining relationships are similarly defined in [4].

**4. SCENARIO**

We briefly present the enrollForPresentation scenario [21], a high-level business process use case, to illustrate the kind of transformation tasks that motivated the development of our tool. This scenario is often used as a basis in the URDAD-related literature and also guided our tool’s implementation and validation. Using that PIM, we handcrafted a PSM and the corresponding Java program code, which could be compared with our tool’s generated output.

Three URDAD views are used to illustrate the specification of the enrollForPresentation service. Figure 3 shows the service’s contract and the static structures of the its request and result objects. The EnrollForPresentationRequest object contains three identifiers identifying the Presentation and the Client who is responsible for payment. The EnrollForPresentationResult contains an Invoice, a ProofOfEnrollment and a StudyGuide. Note the preconditions and postconditions that are associated with each service.

Figure 4 illustrates the three lower-level services that are used by the enrollForPresentation service, as well as the service providers [3] that offer each of these services. The checkStudentSatisfiesEnrollmentPrerequisites service ensures that the student complies with course prerequisites, the issueInvoice service is responsible for the creation of an invoice, and the performEnrollment service enrolls the student for the presentation.

Figure 5 depicts the enrollForPresentation service’s full process specification, including the use of the lower-level services. The process is initiated upon receipt of an enroll-

---

**Figure 3: enrollForPresentation Service Contract [21]**

**Figure 4: enrollForPresentation Responsibility Allocation [21]**

ForPresentationRequest object. An enrollmentPrerequisitesNotSatisfied exception is raised if enrollment prerequisites are not satisfied. When an invoice is requested, if the student does not satisfy financial prerequisites a financialPrerequisitesNotSatisfied exception will be thrown. An enrollForPresentationRequest object is returned on successful termination of the service.

**Figure 5: enrollForPresentation Business Process [21]**

Figure 6 illustrates the workflow followed to produce the example models and code, based on the use case of above. The first step involved the use of an URDAD DSL EMF-Text editor and textual notation [21] to define the PIM. The notation simplified the task of specifying the model as an Ecore PIM, whilst the editor provided syntax checking and ensured that the PIM conformed to the URDAD Ecore meta model. Using the EMF framework, the PIM was then translated into the XMI format required as input for the model transformation. For the sake of comparison between the manually created artefacts and the output of our tool, a JaMOPP conformant PSM of our example scenario had to be constructed from the manually programmed Java code. The results of this comparison are reported in the sections that follow and are described in detail in [4].
5. STRUCTURE OF OUR CASE TOOL

Our tool consists of a combination of newly developed with already existing third-party components. It has been designed to cater for future changes to the model transformation requirements.

The StandardJavaProfileCreator component takes as input a file containing a list of fully qualified names of Java classifiers (interfaces, classes, annotations or enumerations) and produces a Java PSM representation of these classifiers. A Java EJB implementation of an URDAD PIM will not only contain the Java classifiers that are created to represent URDAD model elements, but will also contain references to the pre-existing Java classifiers that these classifiers depend on, such as the Entity, Session, Remote and Local EJB annotations. These will be referred to as ‘URDAD’ Java classifiers and ‘referred’ Java classifiers respectively. To create a Java PSM that can be used to generate EJB code, the Java PSM must not only contain model elements representing the URDAD Java classifiers, but also the model elements representing the referred Java classifiers. During transformation, it would not be feasible to redundantly create the model element representations of these referenced Java classifiers, considering that a consistent set of reference classifiers are used during each transformation process, regardless of URDAD PIM content variation. To simplify the transformation, a separate Java PSM called the ‘Java Standard Profile’ was created that only contains the model element representations of the referenced Java classifiers. When a Java PSM is created from a URDAD PIM, the model elements in the Java PSM now refer to the model elements upon which they depend that are contained externally in the Java Standard Profile.

Figure 7 illustrates how the inclusion of a predefined Java Standard Profile simplifies the transformation process. The JavaStandardProfileCreator, is a custom-made Java program that produces the Java Standard Profile model in XMI format. Its design and architecture are described in detail in [4]. The profile is created by using functionality inherited from the JaMoPPC application, and functionality provided by classes from the EMF and JaMoPP APIs. The application uses a JavaResource, which is able to produce Java Ecore classes representing Java code, or vice versa, and a XMIResource, which creates Java Ecore classes representing XMI elements, or vice versa. A temporary Java class is defined that contains import statements that represent each classifier contained in the ‘classifiers’ file. Using a JavaResource, this Java class is converted into Java (JaMoPP) Ecore classes that represent the Java class and all the classes referred to by its import statements. A XMIResource is used to convert the Java Ecore classes into XMI elements, which are then written to a XMI file named ‘javastandardprofile.xmi’. The Java Standard Profile model conforms to the JaMoPP Java Ecore meta model.

The tool’s primary component is the QVT-R script, sketched in Figure 8, executed with the Medini QVT model transformation IDE, delineated architecturally in Figure 9. The use of QVT-R, as opposed to other transformation techniques such as triple graph grammars [14] [5], is justified in [4].

The Medini QVT QVTProcessor component is responsible for model transformations. It is given a reference to the QVT-R script file, an indication of the model transformation direction, and a collection of Java Ecore objects representing the contents of models participating in the transformation. Functionality provided by EMF is used to produce the Java object representation of the elements belonging to each model. It uses the QVTParser class to retrieve the contents of the script and produce a Concrete Syntax Tree (CST) based on the QVT-R formal grammar. The QVTSemanticAnalyzer class then converts the CST into a more simplified Abstract Syntax Tree (AST), which is used by the QVTEvaluator class to evaluate the QVT expressions and produce a collection of mapping objects called ‘traces’.

Medini QVT uses a trace model to facilitate model transformations and conformance checking. A hybrid trace Ecore meta model is generated, which is based on the meta models of all the models involved in the transformation. It contains model element structures that represent the model element mappings defined in each QVT-R relationship. A trace Ecore model that conforms to the meta model, and contains the state of each trace object, is created during the transformation. When the QVTEvaluator interprets a QVT-R expression, it determines which elements from each model will be affected by the expression. It uses the QVTTraceAdapter class to check for a
pre-existing trace model and load any trace objects that relate to the current expression. A new Java trace object is then constructed that describes the effect the expression will have on the model elements.

The QVTProcessor passes the trace objects to the QVTModelManipulationAdapter class, which ensures that the model elements within the target model are altered to reflect the model element characteristics altered by the trace objects.

A basic understanding of the QVT-R language is needed to interpret QVT-R scripts [17] [15]. Our script was structured into five top-level relationship definitions for the URDAD-to-EJB transformations.

The DataStructureToClassCompilationUnit top relation is used to manage the mapping of a data structure to a JavaBean, a data structure feature to a JavaBean field, and a data structure feature to JavaBean accessor and mutator methods. The mapping of a responsibility domain to a stateless Session Bean and a service contract to a Session Bean method is managed using the ResponsibilityDomainToCommonSessionBeanInterfaceCompilationUnit, ResponsibilityDomainToLocalSessionBeanInterfaceCompilationUnit, ResponsibilityDomainToRemoteSessionBeanInterfaceCompilationUnit, ResponsibilityDomainToSessionBeanClassCompilationUnit for the top-level relations. As mentioned before, there are order-of-application dependencies between some of these relationships. Every top relationship depends on a number of lower-level relationships and ‘helper’ queries to fulfill its element mapping [4].

The DataStructureToClassCompilationUnit top relation ensures that all URDAD DataStructure elements are mapped to Java CompilationUnit elements containing a single Class classifier element. A DataStructure’s name is assigned to a Class name, and its name space is derived from the ResponsibilityDomain hierarchy within which it is contained. Using specific OCL query methods, a CompilationUnit unique name is constructed by concatenating ‘java’ to the value of the name variable, which is in turn concatenated to the String representation of its name space. Once the relation holds true, to continue the mapping process, each of the lower-level relations named in the where clause are evaluated. These relations add additional elements to the CompilationUnit or Class elements, depending on characteristics exhibited by the particular DataStructure being mapped. For example, some of the relations apply to data structures that extend the Entity data structure, while others only apply to abstract or exception data structures.

The ResponsibilityDomainToCommonSessionBeanInterfaceCompilationUnit ensures that all URDAD ResponsibilityDomain elements are mapped to Java CompilationUnit elements containing a single Interface classifier element representing the common Session Bean interface. Each responsibility domain is represented by a stateless Session Bean, consisting of a common interface, annotated local and remote interfaces, and a class that implements these interfaces. The names of the CompilationUnit and Interface are constructed using the ResponsibilityDomain’s name.

For each URDAD service contract belonging to the responsibility domain represented by a Session Bean, both a synchronous and asynchronous method declaration is defined on the Session Bean’s common interface. There is a relation that determines if the ResponsibilityDomain contains any ServiceContract elements, and if it does, it maps the Java elements needed to import the Future Class classifier contained in the Java Standard Profile. Likewise, there is another relation that determines if a ServiceContract element specifies preconditions. The relation maps the Java elements needed to import the exception DataStructure that is associated with each PreCondition element, if the DataStructure does not belong to the same namespace. Yet another relation is used to map each ServiceContract element to synchronous and asynchronous InterfaceMethod elements. The Java Bean representing the service contract’s request data structure is used as a parameter for each method, and the Java Bean representing the service contract’s result data structure is used as a method result.

The JavaCodeCreator shown in Figure 10 is a Java application that produces Java code from the Java-oriented PSMs. Three arguments need to be specified at execution time, including the locations of the Java Standard Profile and Java PSM XMI files, and the directory where the resultant code will be generated. The JavaStandardProfileCreator architecture is documented in [4]. EMF and JaMoPP APIs are used to generate Java code from a Java PSM and the Java Standard Profile. Both the models conform to the JaMoPP Java meta model. Before
proceeding, all the relative references made in the Java PSM to elements contained in the Java Standard Profile need to be converted to absolute references. By default, Medini QVT creates relative element references such as ‘href="javastandardprofile.xmi#/35/@classifiers.0"’, which needs to be changed to an absolute reference such as ‘href="file://localhost/javasourcecodecreator/source/javastandardprofile.xmi#/35/@classifiers.0"’. The application uses an XMIR esource to convert the model elements contained in the two input models into Java (JaMoPP) Ecore classes. It then locates all the CompilationUnit Java Ecore classes and converts each of these classes into Java code, which is placed in its own Java source code (.java) file, and in a directory that corresponds with the Java package structure.

The Java EE EJB Test Application is an Eclipse-based Java project that is used to test the Java code. Its directory structure includes a ‘source’ directory where the Java code produced by the JavaCodeCreator is placed, a ‘descriptors’ directory that contains the EJB XML deployment descriptors, and a directory that contains ‘third-party’ JAR files. Apache ANT is used to manage project lifecycle activities such as compiling, packaging the code and EJB descriptors as an EJB application, and deploying the code to a JBoss EJB application server. Our application server used a MySQL database for EJB Entity Bean persistence, and Hibernate for object relational mapping.

6. PROJECT RESULTS

In this section we describe our tool’s model transformation and code generation capabilities, with regard to the above-mentioned ‘enrollForPresentation’ benchmark scenario. We compared the tool’s output with a manually implemented Java EE EJB system developed without tool support.

The JavaStandardProfileCreator tool was used to produce a Java Standard Profile XMI file named ‘javastandardprofile.xmi’, using a text file containing the fully qualified names of referenced Java classifiers. At this stage, there was no reliable means of checking whether the contents of the XMI file were valid or sufficient. This only became apparent after the profile was used by other components.

The PIM and the Java Standard Profile were transferred to the Eclipse-based Medini QVT QVT-R IDE, where the URDAD and Java JaMoPP Ecore meta models were registered. A run-configuration was created that identified the QVT-R script containing the QVT-R relations, a list of three XMI files representing the model instances involved in the transformation, as well as the transformation direction. The URDAD PIM and Java Standard Profile files were specified, as well as an empty XMI file that was created to constitute the target Java PSM, which would be filled with the model elements produced by the model transformation.

Once the transformation was initiated, the resulting model elements that were generated, were compared with the contents of the Java PSM example file, to ensure that the tool had produced a valid PSM. There were indeed noticeable differences between the contents of the two files, but the cause was easy to determine.

A notable variation was the size of the generated file being approximately 7000 lines, compared to the size of the example file, which exceeded 19800 lines. Sections within each file, representing corresponding model elements, were carefully inspected. Considering the size of the two files, only after the Java PSM was converted into more legible Java code, would it became feasible to validate by inspection the results of the transformation.

The discrepancies between the generated PSM and example PSM model elements can be summarised as follows. The PSM example contains all the model elements representing the ‘referenced’ Java classifiers, while the generated PSM model elements refer to equivalent model elements contained externally in the Java Standard Profile. Unlike within the example PSM, the names of the CompilationUnit elements in the generated PSM were fully qualified. Unique fully qualified names are required to enable QVT-R OCL ‘helper’ queries to locate compilation units during the transformation. The sequence of the CompilationUnit elements differed, as a result of the order in which they are constructed by the relationships defined within the QVT-R script. The order of the model elements, representing Java methods and fields, within each class classifier differ. The sequence of elements within the PSM example mirrors the method and field order within the Java classes from which they were derived. The order of equivalent structures in the generated Java PSM are dictated by QVT-R relationship execution sequence. There are additional ‘higher-level’ Namespace elements associated with each CompilationUnit in the PSM example, while those in the generated PSM directly represent the logical name spaces that are defined in the PIM, through the nesting of named ResponsibilityDomain elements. Each PSM was intended to have a slightly different namespace in order to differentiate them from one another.

Java code was produced from the generated Java PSM and the Java Standard Profile using the JavaCodeCreator. A ‘.java’ file was constructed for each CompilationUnit element and placed within a folder structure corresponding with the CompilationUnit name space structure. The generated Java code was then compared with the manually implemented Java code and was found to be functionally equivalent, although minor discrepancies were identified. The layout of the generated code was not aesthetically pleasing, although this could be rectified by using a code formatting tool. Synchronous Session Bean method bodies in the generated code contained no program instructions, while the corresponding asynchronous versions of these methods were fully implemented, as they simply always invoke their synchronous counterparts in an asynchronous context. The sequential order of the Member elements contained within each JavaBean CompilationUnit element in the generated Java PSM had an effect on the sequential order of the equivalent structures within the Java Bean classes. The generated Java code was man-
usually transferred to the Java EE EJB Test Application’s ‘source’ directory for the insertion of missing Session Bean method functionality, code formatting, compilation, packaging and deployment to a JBoss application server for further testing. The code was made accessible from within the Eclipse IDE by creating a new Eclipse project. The synchronous Session Bean method body program instructions were then copied from the manually implemented Java code to the equivalent method bodies within the generated Java code.

The ‘enrollForPresentation’ method of our example relies on further methods contained in other Session Beans. Those Session Beans were defined as fields within the Enrollments Session Bean and are ‘injected’ into the Session Bean instance at runtime by the EJB application server. Being able to define the implementation of the method by simply copying code from the example, further validated the accuracy of the artefacts produced by the transformation. The duplicated code refers to and uses large portions of Java code that was tool-generated. The Java program was then compiled, packaged as an EJB application and deployed to the JBoss application server. The successful compilation of the application further corroborated the validity of the generated output, and deploying it to JBoss, confirmed that the generated code conformed to the EJB standard. JBoss performs static analysis on the EJB code during deployment. Another critical part of the validation of the EJB code was the accurate generation, by the application server, of the database tables needed to manage the persistence of Entity Bean states. The generated tables were compared with the tables originally produced for the Entity Beans belonging to the example application, and were found to be identical.

Analysing the code produced by the tool we found that it can generate the majority of the structures that constitute the EJB implementation of the URDAD PIM example. Being able to compile, package and deploy the EJB application to an application server helped to increase our confidence in the validity of the artefacts produced by our tool.

As shown in Figure 11, the quality of our tool is not only dependant on the quality of the input URDAD PIMs and the PSMs and Java code it produces, it also dependant on the quality of the ‘architecture’ of these artefacts. The architecture of the PIMs and the PSMs is determined by the structure of their meta models and the EJB specification imposes a specific architecture on all EJB applications.

In summary, our new transformation tool is indeed capable of producing the static EJB structures that are required to implement the example URDAD PIM. Stateless Session Beans and method declarations on these Session Beans accurately represent the responsibility domains and service contracts. Consequently, Java Bean classes are successfully produced that represent the data structures, including data structure class hierarchies and data structure characteristics, such as attributes and relationships with other data structures. A screen capture of our tool’s user interface is shown in Figure 12.

7. SUMMARY AND OUTLOOK

Motivated by the lack of URDAD transformation-tool support, the aim of our project was to semi-automatically transform a URDAD PIM into a Java PSM. For this purpose a CASE tool was developed to support the model-to-model transformations and to generate Java code from the resulting PSM. The aim of our project was by-and-large achieved.

The ‘enrollForPresentation’ use case was used as a benchmark example throughout our project. Though our tool successfully coped with this example, further experiments with other use cases are required to validate the tool, because the tool’s correctness was never formally (mathematically) verified. Although full verification seems infeasible due to tool’s size, it would be desirable to verify at least its most critical parts. Also the quality of the tool’s Java output code was not formally analysed, although this could be done (at least in principle) with the help of existing analysis tools, such as JavaPathFinder. Such measurements are very important to increase the user’s trust in the practical usability of the tool.

Moreover, tool components produce files (XMI or ‘.java’) that are required by other components. For example, the Java Standard Profile file that is generated by the Java StandardProfileCreator is required by the Medini QVT-based QVT-R script and the JavaCodeCreator. Currently, these files must be manually transferred between those components, which is tedious and error-prone. Future versions of the tool should provide more convenience in this regard.

In some instances, before a relationship between PIM and PSM model elements may be established, other relationships must already exist. For example, the generation of the elements representing a Java Bean class depends on the creation of the elements representing its fields, and
if any field refers to a Java Bean class, the elements representing the class must also be present. While we have seen that QVT-R is capable of resolving these types of dependencies, we still need to confirm that it can cope with cyclic dependencies between model elements (structural recursion).

Java import statements are used to include Java classifiers, such as interfaces, classes and annotations, which are referred to by a classifier, but do not reside in the same name space. Test results indicate that our tool is capable of differentiating between classifiers that exist in the same name space and those that do not. Due to project time constraints it has not yet been sufficiently tested whether our tool will produce duplicate import statements if the same classifier is referred to more than once.

To date, our tool accepts an input PIM 'as is' and attempts to transform it regardless of its internal consistency. By analogy, this is comparable to a compiler that does not do type checking, or report type errors during a compilation from source to target code. Future versions of our tool should be enhanced to enable it to validate the structure and contents of an input PIM before transformation. This requires a formal representation of the very notion of 'PIM consistency', which is not yet supported, and may eventually even lead to a critique (and enhancement) of the URDAD meta model (URDAD-DSL) itself.

At PIM level, it is possible that a service contract may be fulfilled by more than one service. As per URDAD, a service does also not need to reside in the same ‘responsibility domain’ as its corresponding service contract. The tool currently only supports the transformation of services that exist in the same ‘responsibility domain’ as their matching service contract, and service contracts may each only be fulfilled by a single service. Currently, our model transformation does not take the algorithmic details (internal control- and data-flow) of services into consideration. Accordingly, we only generate their corresponding Java method signatures and leave their bodies empty for manual implementation. We hope to support the full implementation of method process logic in a future version of our tool.

One of the limitations associated with QVT-R is the difficulty of specifying complex queries using OCL [6]. It is also not a trivial exercise to declaratively define the application order of dependencies between QVT-R transformation relationships. If the perceived complexity associated with the transformation of model elements that represent URDAD services is taken into consideration, would it remain feasible to continue using QVT-R? It may be easier to define the complex transformation-order dependencies using the imperative QVT-O language instead, although any increase in the complexity of the transformation ‘engine’ is obviously in conflict with the above-mentioned desire to formally verify the correctness of the ‘engine’.

As mentioned above there are numerous ways of representing URDAD PIM in EJB. The version that is currently produced by the tool reflects just one possible interpretation. Stateless Session Beans are used to represent service contracts. An alternative approach would be to implement service contracts and services using message-driven Beans that respond to messages sent to the application through Java Message Service (JMS) queues. In such a way one could map an URDAD PIM onto a fully distributed system across a computer network. Future versions of the tool should allow users to select between alternative transformation strategies.

In ongoing work in our research group, a ‘competitor’ tool that is similar to the tool described in this paper, is currently under development. This new tool aims to transform URDAD specifications into Microsoft’s .NET framework, instead of Java EE EJB. As soon as the tool is operational, further lessons shall be learned by comparing the two tools and their generated output.

Acknowledgments

Thanks to Fritz Solms, the father of URDAD, for fruitful conversations in the context of this paper. Thanks also to Manfred Nagl for many valuable remarks on this project during his visit to Pretoria in October 2010.
8. REFERENCES


