JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Transformation of Web Service Specification Languages into UML Activity Diagrams

Magisterarbeit zur Erlangung des akademischen Grades

Diplom-Ingenieur (Dipl.-Ing.)

In der Studienrichtung *Informatik*

Angefertigt am *Institut für Bioinformatik,*
&
School of Computer and Information Science,
University of South Australia

Eingereicht von:

Thomas Josef Reiter

Betreuung:

Univ.-Prof. Mag. Dr. Werner Retschitzegger

Univ.-Prof. Dipl.-Ing. Dr. Markus Stumptner

Linz, *März 2005*

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, 8.3.2005 Thomas Reiter

Acknowledgements

First of all I want to thank my supervisor Prof. Werner Retschitzegger for linking me to the University of South Australia, which turned out to be an invaluable experience. I am very thankful for all the advice and support he offered concerning this work.
Special thanks go to Prof. Markus Stumptner who supervised me during my research visit at the UNISA, where a substantial part of this diploma thesis had been conceived.
A huge “Thank you” goes to my girlfriend Rachel, who kept me motivated all the way, and to my mother Maria, for her support throughout my studies.

Last but not least, I want to thank all the anonymous developers out there, who in a self-less effort, contribute to the numerous open-source projects we all benefit from!

Thanks to all of you,

Thomas

Abstract

The *Business Process Execution Language* has found wide acceptance as a means to describe executable business processes. However, BPEL is an XML-based language and thus suffers from poor readability. Therefore, the need for abstraction in the form of a more intuitive, higher-level notation arises. In turn, fragmentation into several proprietary, mutually incompatible realizations ought to be avoided. This clearly favours the use of a wide-spread standard, such as the *Unified Modeling Language*. A final goal is to implement a mapping able to perform the round-trip from BPEL to UML and vice versa.

This work deals with the automated transformation of BPEL documents into UML Activity Diagrams. To carry out the mapping, a working prototype of a generic model transformation engine named *Marius* has been developed. The mapping from BPEL to UML is specified in transformation definitions written in *Marius*' transformation language. Apart from the mapping's implementation, this diploma thesis explores the practical issues of design, development and application of a framework for generic model transformations in the context of the *Model Driven Architecture*.

Linz, 8.3.2005 Thomas Reiter

Contents

Chapter 1

Introduction

Several standards and technologies for describing and automating business processes have evolved in the recent years. The *Business Process Execution Language*, BPEL [BPEL03] for short, is one of them. BPEL is a language designed to describe executable business processes based on web services. An engine may execute such processes by orchestrating the control flow between the Web Services involved. In that manner, organizational boundaries between companies or departments can be overcome, and several web services can be composed into a loosely coupled business flow. From a technical point of view, BPEL is an XML based language and is built upon SOAP [SOAP03], WSDL [WSDL01] and XML Schema [XSCH04]. BPEL is an XML-based language and therefore not practical for use in business modelling and not suitable for users lacking a certain level of technical background on that matter. Hence, the need to provide more user-friendly modelling capabilities than those XML editors offer, is given.

Since UML is the „lingua franca“ in modelling and considering its extensibility and support in tools on the market, the decision to map BPEL to UML (strictly speaking a UML profile for business modeling) comes easy. Business processes can intuitively be understood as workflows, and therefore mapping the dynamic part of a business process to UML Activity Diagrams is at hand.

The goal of this diploma thesis is to map BPEL documents to UML Activity Diagrams. Another, previously started diploma thesis [RINN03] works in the opposite direction and produces BPEL documents from UML models. The idea behind these two projects was to provide means to create UML from BPEL to analyse and model in a UML environment and then finish the roundtrip to BPEL again.

Working with a focus on models in mind fits exactly into the paradigm of MDA - Model Driven Architecture [MDA02]. This is why the implementation executing the above-mentioned requirements works after this very principle. How the architecture is exactly laid out and how the MDA paradigm is followed, will be elaborated on in the following chapters in more detail.

1.1. Scope of this Diploma thesis

The mapping employed in this work is largely inspired by a proposal made by [IBM03] in the way BPEL constructs are mapped to UML model elements. Providing an implementation that facilitates a mapping of all possible BPEL constructs to UML is out of scope of this work. Nevertheless, the implementation provides a modifiable mechanism to create UML from BPEL. This flexibility allows users to extend and alter the mapping in a way they see fit.
A decision was made to follow the MDA paradigm and develop a simple framework for model transformation, which as a proof of concept, transforms BPEL to UML. The implementation of a new transformation definition language and engine was sparked by the fact that at the time of writing, no standardized tool for generic model transformation has yet evolved. For an up-to-date overview of related projects concerned with model transformation refer to the “Related Work” chapter.
The transformation language used in this work is limited in its expressiveness, but was nonetheless developed with a focus on openness and extensibility. As already stated, providing a full-blown generic model transformation tool is out of scope of this work. The implementation should be understood as a prototype - to demonstrate as well the usage and the development of a model transformation capable system. This diploma thesis focuses on the practical issues of model transformation in the context of transforming business processes defined in BPEL into UML Activity Diagrams.

1.2. Structure of this Document

This document is split into eight chapters, each one dedicated to an aspect of the concepts relevant to this diploma thesis.

Following this introduction, chapter two gives an overview of the technologies and standards employed in this work.

Chapter 3 elaborates on the notion of model transformation in general, and discusses various technologies and methods applied in this respect.

Chapter 4 introduces the Marius Transformation Tool and gives a detailed description in terms of architecture and implementation.

Chapter 5 explains how the BPEL constructs were mapped to equivalent UML model elements. Furthermore, an example transformation is carried out that shows the execution of the transformation process in a step-by-step manner.

Chapter 6 gives an overview of related work in the model transformation domain.

Chapter 7 tries to give an outlook into the future development of technologies and standards relevant for this work. Additionally, ideas on how to improve and extend the Marius tool are discussed.

Chapter 8, the appendix, is mostly comprised of the transformation definitions' source code facilitating the BPEL to UML mapping. Also contained are the BPEL metamodel used in the mapping, further example transformations, and a number of other documents enhancing the understanding and completeness of this work.

Chapter 2

Background Technologies

The approach to model transformation taken in this work adheres to the paradigm of MDA. To make full use of this paradigm various technologies and standards come into play. This chapter provides a quick breakdown of each of these technologies and elaborates on its role within MDA.

The model transformation tool implemented in this project makes use of all the below mentioned technologies, including a prototype transformation engine to complete the framework proposed by MDA.

2.1. Model Driven Architecture

In many other engineering disciplines, model building is a common part of the development process. The shipbuilding or aerospace industry for instance, build different models for various kinds of purposes prior to the production process. An example would be a scaled replica (analysis model) of an airplane for wind tunnel experiments. MDA tries to aid the software engineering development process by integrating and standardizing means for model building, such as model interchange and model transformation.

MDA tries to shift the software engineering focus from code-centric to model-centric development. There are many software modelling standards and tools around, but those mainly act as means for describing and analysing a data model, which then is

implemented manually. The code generation wizards in CASE tools work on a model-to-text base only, not on a real model-to-model base. Furthermore, most CASE tools tend to have proprietary code generation systems with a fixed metamodel and domain transformations already in place. (e.g.: code generation from UML to Java) MDA aims to relieve these interoperability problems by keeping the modeling and code generation systems accessible to the developers.

2.1.1. Model Transformations in MDA

As the OMG's new flagship MDA is supposed to raise the efficiency of software engineering through use of modelling, model transformation and code generation. Therefore, one of MDA's main goals is to add a new level of abstraction through model layers. A distinction between platform-independent models (PIM) and platform-specific models (PSM) can be made. The use of a model transformation language enables programming at an abstract level and should keep technical and domain specific issues separated. Thus, through separation of concerns and code reuse, manageability and maintenance efficiency should be increased and development time lowered.

There can as well be different model domains and levels of refinement between these models, down to source code. The models used in MDA have to be formal models, meaning they have to adhere to a certain metamodel or „model language". Otherwise, there would not be a way to process these models automatically. A typical example would be the "refinement" of a PIM into one or more PSM. These "refinements" are actually transformations of models from a source domain to a target domain. Code generation can be interpreted as the lowest tier in model transformation. Therefore, a transformation is an operation that is carried out on a source model producing a target model. This transformation is described in a transformation definition, which again has to be defined formally in a transformation definition language for automation purposes.

Executable transformations are generated by an engine, which takes in transformation definitions and produces executable code, which then is run on a source model to produce a target model. The models used may be UML models or models of a specific UML profile. But also any other kind of formal models may be used in MDA, like the BPEL models that are mapped to UML models in this work.

2.2. Unified Modeling Language

The Unified Modeling Language is the quasi-standard modelling and specification language for software systems and provides different types of diagrams to enable a developer to describe the static, dynamic and functional behaviours of a system. UML offers extensibility through a profiling mechanism, which means the basic UML constructs can be semantically enriched to allow customization for different usage scenarios. (e.g.: UML profile for Business Modeling)

2.2.1. UML Metamodel

The UML metamodel is a description that defines the structure of UML models. Instances of the UML metamodel are UML models.

The MDA paradigm requires formalized models to be able to automate the model transformation and model interchange process. A metamodel (like the UML metamodel) for a given model facilitates this kind of formalization. Practically, the UML metamodel is a blueprint for its models, as it defines of which meta-objects and meta-attributes the different diagram types consist of.

Figure 1 shows the top-level package structure of the UML metamodel. At the time of writing the current version of UML was 1.5 [UML03] (which basically is UML 1.4 including Action Semantics), although version 2.0 should soon be adopted. However, the implementation belonging to this work is based on UML version 1.4 plus the UML 2.0 Diagram Interchange as an extension to the standard specification.

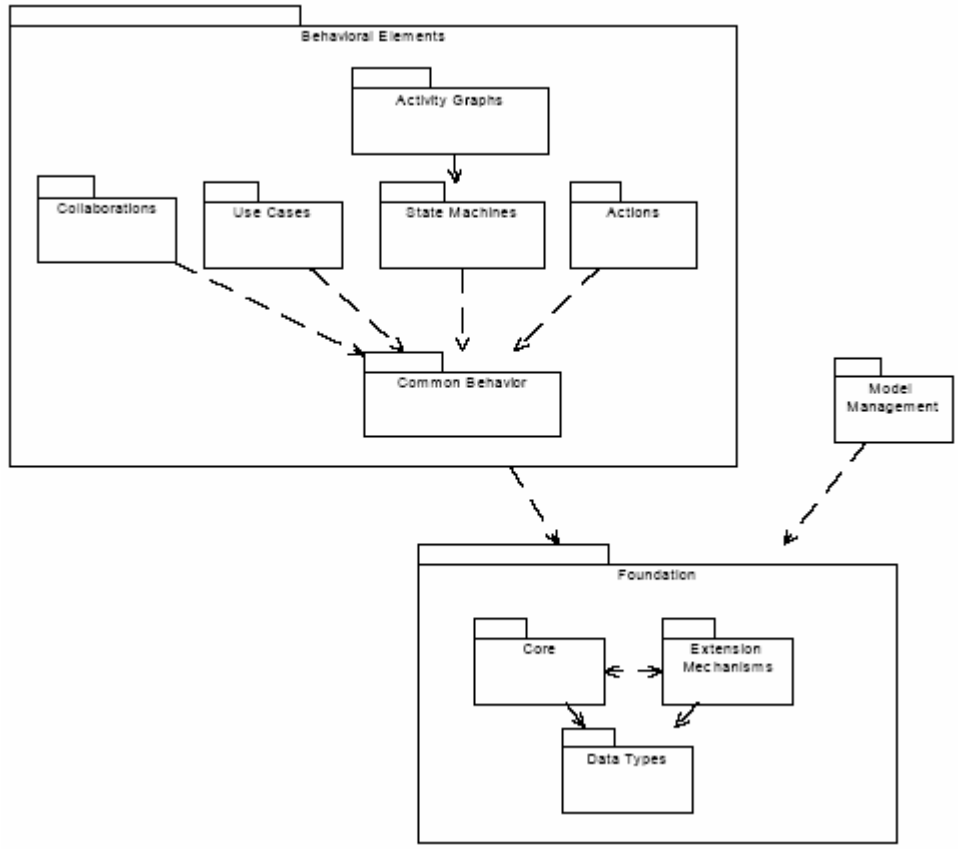

Figure 1 Class Diagram Showing UML Package Structure

2.2.2. UML Activity Diagrams

An *Activity Diagram* displays an *Activity Graph* as defined in the Activity Graphs Package in the UML metamodel. An Activity Graph is a state transition system and an extension of a *State Machine*. The primary focus is set on modelling the sequence and conditions in which actions take place concerning a certain identifier. The states in these graphs represent actions, which are connected by transitions, which are triggered by events.

The very nature of Activity Graphs makes them especially suitable for describing workflows. The UML profile for Automated Business Processes as proposed by IBM, [IBM03] makes use of UML Activity Diagrams to describe the dynamic aspect of a Business Process.

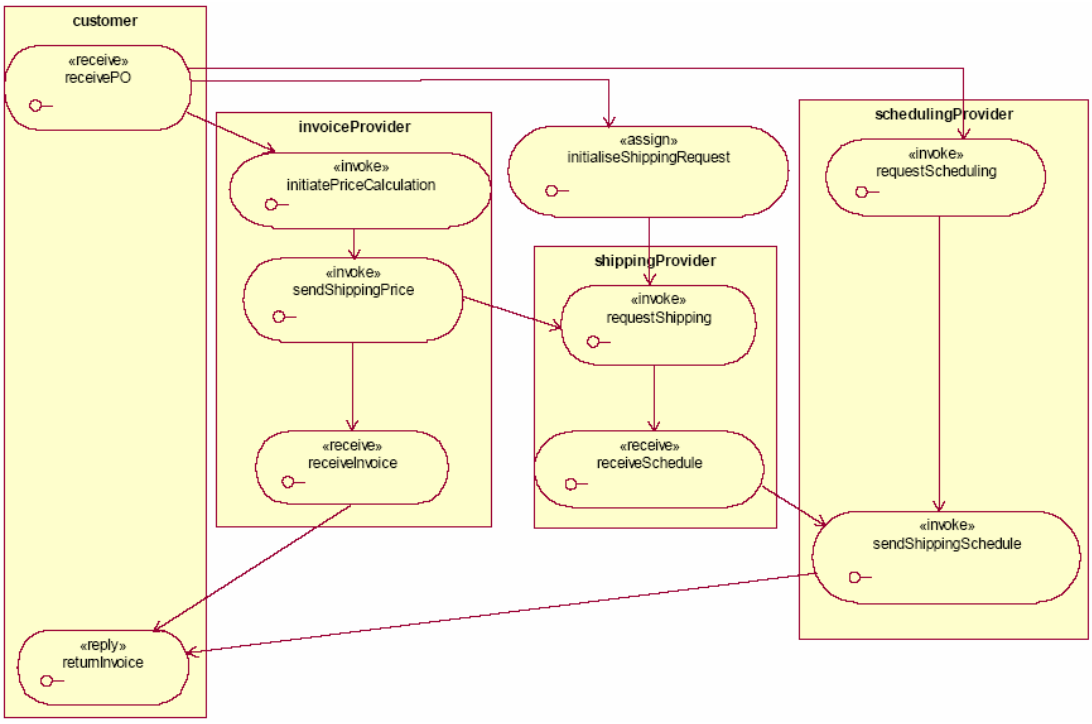

Figure 2 [IBM03] "Purchase Order" Business Process as Activity Graph

Figure 2 depicts the "Purchase Order" business process according to IBM's UML Profile for Business Modeling [IBM03]. The process begins by receiving a purchase order request from a customer. Subsequently activities from other business partners are triggered, which produce a response that is finally returned to the customer.

Though for simplicity reasons some details on message flows are omitted, the structure of the process and the sequence in which the activities are executed is clearly visible. Note that the way any UML Diagram is displayed, is not defined in UML 1.x. Hence, the look or "style" – but not the semantics - of Activity Diagrams may vary.

2.2.3. UML 2.0 Diagram Interchange

"Diagram Interchange" [DIA03] is a package of UML 2.0 and is a separate metamodel designed to standardize the exchange of diagram display information between CASE tools. UML 1.4 however does not include this package. But through merging the Diagram Interchange *DI* metamodel with the Standard UML 1.4 *UML* metamodel a new, extended *UML+DI* metamodel is created. Such a merged metamodel is provided by Gentleware and finds application in their *Poseidon* [GENT] modelling tool. Using this extended metamodel not only model information but also display information, meaning the way the model should be displayed in a CASE tool, can be captured.

Because previously modelling tools' file formats have been largely proprietary, the above outlined approach is a major step towards interoperability between tools. For tools not supporting Diagram Interchange *DI*, but SVG [SVG03] (Scaled Vector Graphics) as means of storing display information, an XSLT sheet [XSLT99] can be applied to an XMI [XMI03] document containing a *UML+DI* model, and as a result produce an SVG document.

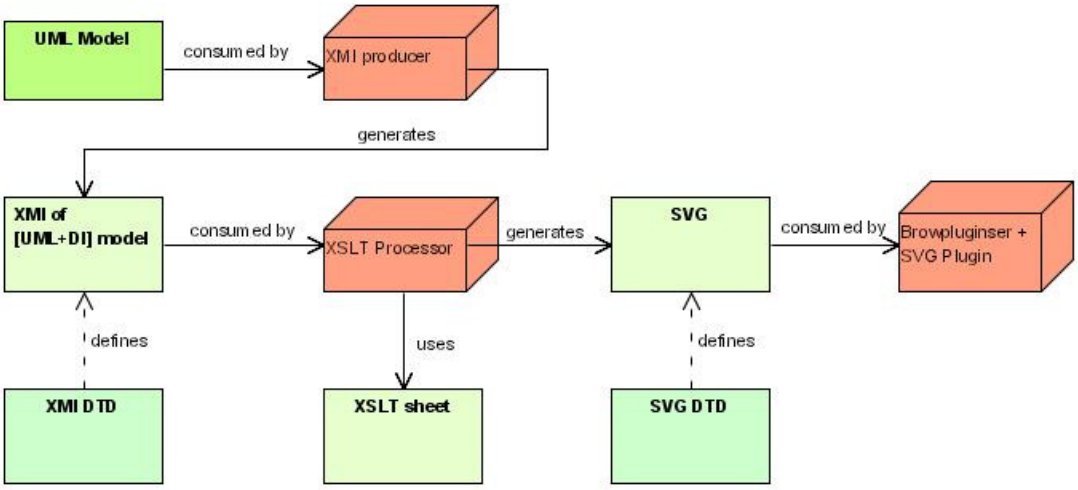

Figure 3 XMI UML+DI Application Scenario

Figure 3 displays an example application scenario for XMI and the Diagram Interchange extension for UML: First, an XMI document is generated by exporting a UML model from a CASE tool. Now the XMI document (containing diagram display information) could be imported in another modelling tool. However, the above scenario feeds the XMI document into an XSLT processor and produces an SVG document.

This example should illustrate that XMI need not only be used to interchange models, but also prove valuable in other application scenarios, such as XML and XSL processing, which can be considered as low-level model transformations, too.

2.3. Business Process Execution Language for Web Services

Web Services inaugurated a completely new way in which systems could interact with each other on the Internet. They can play a major role in providing a flexible way to offer services to business partners when conducting e-business. However, this openness and flexibility also causes interoperability problems when integrating distributed systems on the internet. These interoperability problems stem from

differing protocols that can be agreed on to drive communication among web services. To successfully integrate web services into a cohesive system, the use of standards is at hand. To address this problem several technologies have been developed – BPEL4WS is one of them. For an overview of related approaches refer to [BKKR03].

BPEL4WS (or BPEL for short) is a fusion of IBM's WSFL [WSFL01] and Microsoft's XLANG [XLAN01] designed to enable the automatic execution of business processes based on web services. BPEL is based on XML (data model and vocabulary), SOAP [SOAP03] (message exchange) and WSDL [WSDL01] (web service definitions).

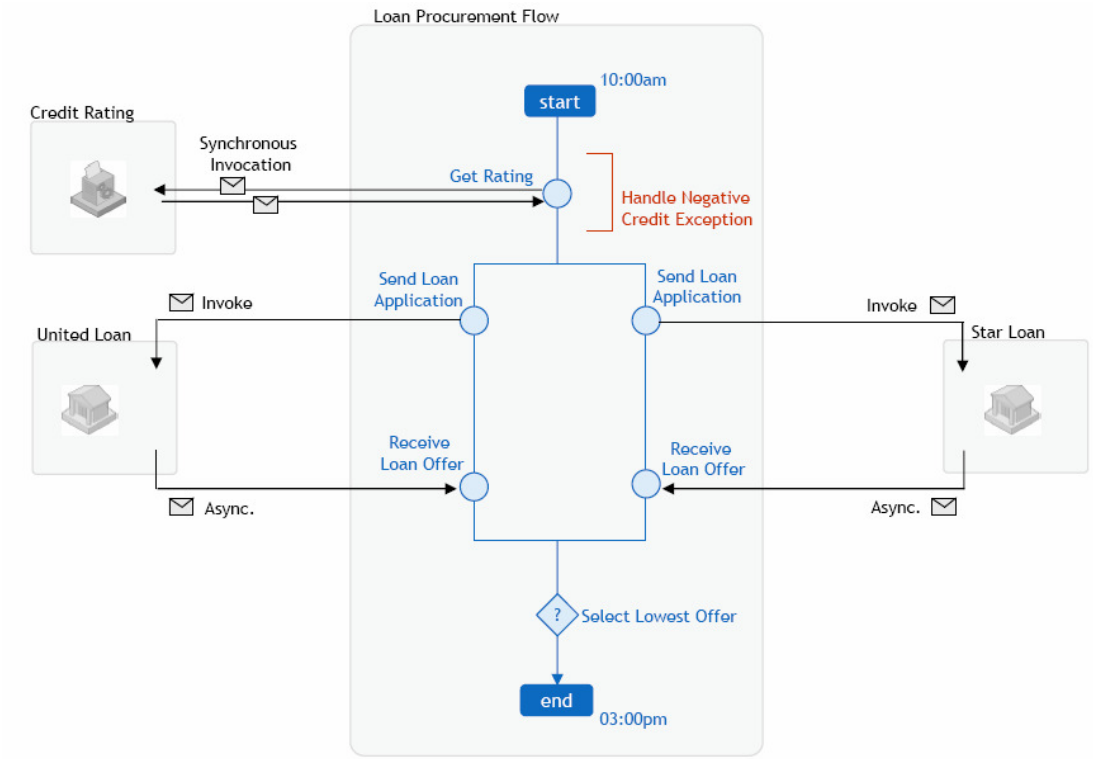

Figure 4 [COLL03] Informal Description of Business Process

Figure 4 shows an informal description of a business process, a loan procurement to be exact. It is taken from Collaxa's tutorial on BPEL [COLL03] and deals with the automation of the example process on a BPEL platform.

2.3.1. Scope of BPEL

BPEL has two usage scenarios. The first one is be to describe abstract business processes. An abstract business process is not executable and shows a process' business protocol view only. The other, for us more interesting scenario, is using BPEL to describe executable business processes. Executable business processes are

workflows and can be run in a BPEL engine. Such an engine is essentially orchestrating the web services involved in the business process as described in a BPEL document. Orchestrating means providing statefulness, invoking other business partners' Web Services, and passing messages among these according to the control flow laid out in the underlying BPEL document.

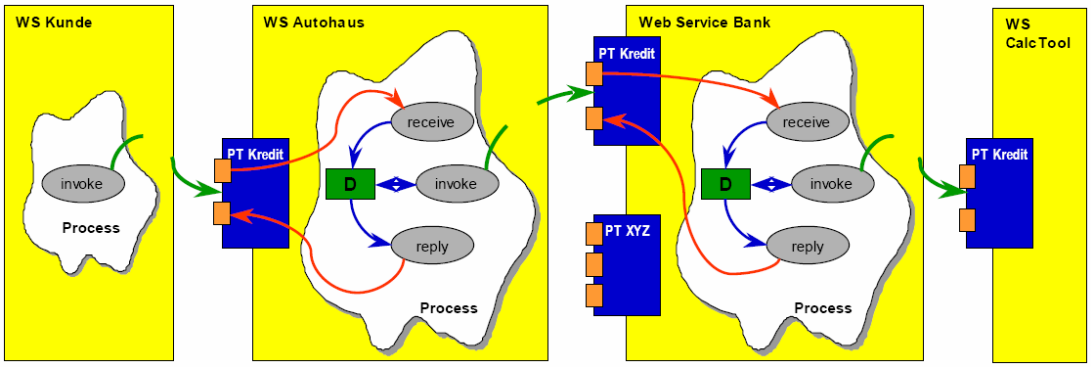

Figure 5 [MART02] BPEL Execution Example

The example depicted in Figure 5 deals with a customer wanting to obtain a loan for a car purchase. It shows how web services belonging to different business partners, such as customer, car dealer, bank and an independent calculation tool are invoked, their responses processed and a result being replied to the customer.

2.3.2. The "MyBPEL" Metamodel

BPEL is specified as an XML Schema [BPEL03]. Until completion of this work no standardized, XMI serialized, Meta Object Facility [MOF02] compatible metamodel for BPEL existed. The BPEL metamodel used in this work is a subset of BPEL version 1.1. The developed BPEL metamodel, from now on quoted as *MyBPEL*, omits some language constructs like compensation, fault and event handlers for reasons of simplicity.

The *MyBPEL* metamodel was created in the Poseidon UML tool and can be found in the appendix. The serialized version of the model was converted from *UML-XMI* to *MOF-XMI* and used as the source metamodel in the transformation framework. For an exact description of how to generate a MOF-compatible metamodel, please refer to 3.3.1, "Defining and Hosting Metamodels".

2.4. MOF – Meta Object Facility

The Meta Object Facility [MOF02] is a three layered, conceptual framework for describing metadata and enabling model driven systems. The main goal of the MOF

is to capture the semantics of a system in a language and technology independent manner.

At the top of this architecture stands the MOF model. The MOF model is an "abstraction language" to describe other metamodels. Thus, it is a metamodel for describing metamodels. This is why it is also referred to as a "meta-metamodel" or simply M3, because it is one abstraction level above M2 metamodels, like UML for example.

2.4.1. Metamodel Hierarchy

As described above, the MOF Model is placed in level three of the OMG's metamodelling hierarchy, as depicted in Figure 6. Theoretically there could also be a fourth layer above, describing the MOF Model, and a fifth layer above, describing the description of the MOF model and so forth. But due to the fact that the MOF model is actually able to describe itself, the hierarchy ends at the third tier and introducing any further levels would generally not prove useful. The MOF model can be seen as a "Great Unifier" being able to describe all kinds of different metamodels, which are instances of M3 and found one level lower in M2.

The second layer, M2, is the metamodel layer. Here we find metamodels such as UML, Common Warehouse Metamodel [CWM03] and the like. These models are metamodels defining the "language" of underlying models. Every UML model consists of constructs that are defined in the UML metamodel.

The first layer is where we find models, instances of their specific metamodels. These are UML models, CWM models and so on. The M1 layer is the abstraction level usually worked on by users when creating models in CASE tools. Modelling in a UML CASE tool, where the metamodel is set to UML produces UML models. Modelling involving layers M2 and M3 are considered metamodelling activities producing metamodels.

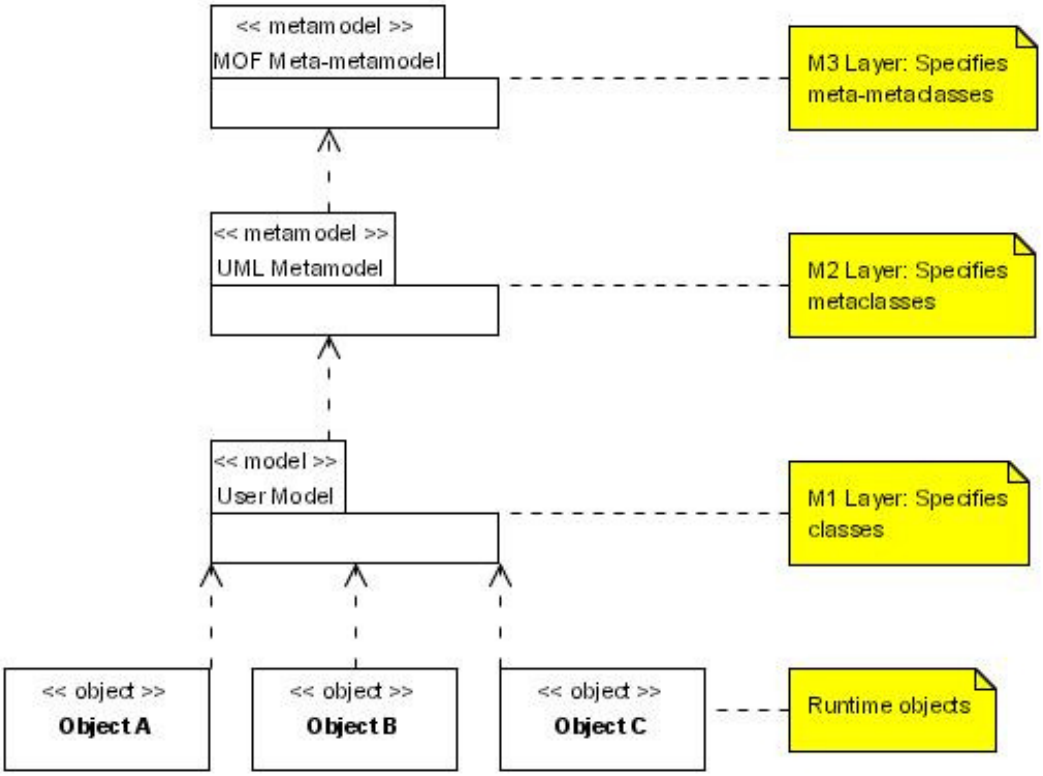

Figure 6 The four-layered architecture of the MOF

The lowest layer in the hierarchy is M0. This layer is reserved for instances of models, for objects. These objects are entities existing at runtime, like Java objects or database entries, and comprise the implementation of a modelled system.

2.4.2. MOF within a Metadata Repository

As the name indicates, a metadata repository is a system designed to store meta-information. This means such a repository can store metamodels (M2) and models (M1), their instances. A metadata repository is built upon a core meta-metamodel, such as the MOF metamodel. Therefore, a MOF-based repository can store any MOF compliant metamodel, such as UML, CWM or a custom made metamodel like MyBPEL used in this work, as well as instances of them.

Due to the self-descriptiveness property of the MOF metamodel, a MOF based repository can even contain the MOF metamodel itself (!).

To implement such a system and provide a standardized programming interface for it, a way to map MOF semantics to a programming language is required. With JMI, this technology exists and will be discussed in the following section.

2.5. JMI – Java Metadata Interface

The Java Metadata Interface is the Java rendition of the MOF. It provides a common Java programming interface for accessing metadata. In this work, the JMI programming interfaces are used to programmatically instantiate metamodels and manipulate their instances within a metadata repository.

2.5.1. MOF Mapping to Java Interfaces

Any MOF compliant metamodel can be used to produce JMI interfaces. For every metadata element, certain JMI interfaces are generated. Basically, there are four different kinds of JMI interface types: *RefPackage*, *RefClass*, *RefObject* and *RefAssociation*. The metamodel depicted in Figure 7 is taken from the JMI specification [JMI02]. Below is an example showing each kind of JMI interface generated from a specific model element within the "XMLModel" metamodel.

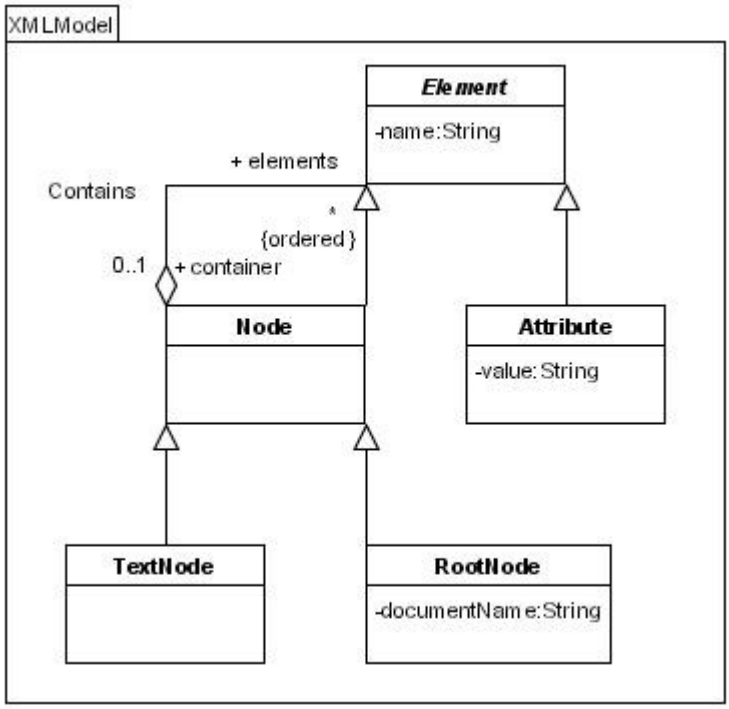

Figure 7 Simple XML Metamodel

- *Package Objects* correspond to MOF packages and are little more than a compilation of operations that provide access to metaobjects stored in this very package. Metaobject packages can contain class proxies, instance objects, associations and nested packages. The code produced for the package XMLModel that contains all other metaobjects is the following:

```
public interface XMLModelPackage extends javax.jmi.reflect.RefPackage {
public NodeClass getNode();
public AttributeClass getAttribute();
public ElementClass getElement();
public RootNodeClass getRootNode();
public Contains getContains();
}
```

- *Class Proxy Objects* are basically factories that are used to create and hold instance objects once existing. They also hold their state and provide functionality to manipulate classifier-scoped attributes. The code generated for the Class Proxy Attribute is the following:

```
public interface AttributeClass extends javax.jmi.reflect.RefClass {
public Attribute createAttribute();
public Attribute createAttribute(String name, String value);
}
```

- *Instance Objects* correspond to MOF classifiers and are tied to Class Proxy Objects which produce and contain them. Instance objects provide functionality to manipulate instance-scoped attributes as well as accessing and updating referenced associations. The code generated for the instance Element is the following:

```
public interface Element extends javax.jmi.reflect.RefObject {
public String getName();
public void setName(String newValue);
public Node getContainer();
public void setContainer(Node newValue);
}
```

- *Association Objects* correspond to associations defined in a metamodel. They contain a collection of links, which are instances of Association Objects, and refer to two Instance Objects. Association Objects provide functionality to query, add, modify and remove links from the link set. The code generated for the Association Object Contains is the following:

```
public interface Contains extends javax.jmi.reflect.RefAssociation {
public boolean exists(Element element, Node container);
public java.util.List getElements(Node container);
public Node getContainer(Element element);
public boolean add(Element element, Node container);
public boolean remove(Element element, Node container);
}
```

The following code piece shows how an Attribute Instance Object can be generated and manipulated, assuming that service is a reference to the outermost package proxy:

```
 Attribute attr = service.getAttribute().createAttribute(<name>,
<value>);
```

```
attr.setContainer(<parentNode>);
```

For the complete example and more detail on the MOF to JMI mapping please refer to the JMI specification. [JMI02]

2.5.2. JMI Reflective Package

The JMI Reflective Package is a part of the metamodel API and allows a program to use objects without prior knowledge of the objects interfaces. This functionality enables a program to discover the semantics of any object and manipulate it just as it could with the metamodel-specific, "tailored" interfaces. The Reflective Package contains eight interfaces that all generated, metamodel-specific interfaces - like those in the example above - extend. These interfaces contain common operations for the type of metaobjects they represent. "RefClass" for instance provides operations for instantiating metaobjects, whereas "RefAssociation" deals with managing the links belonging to an association.

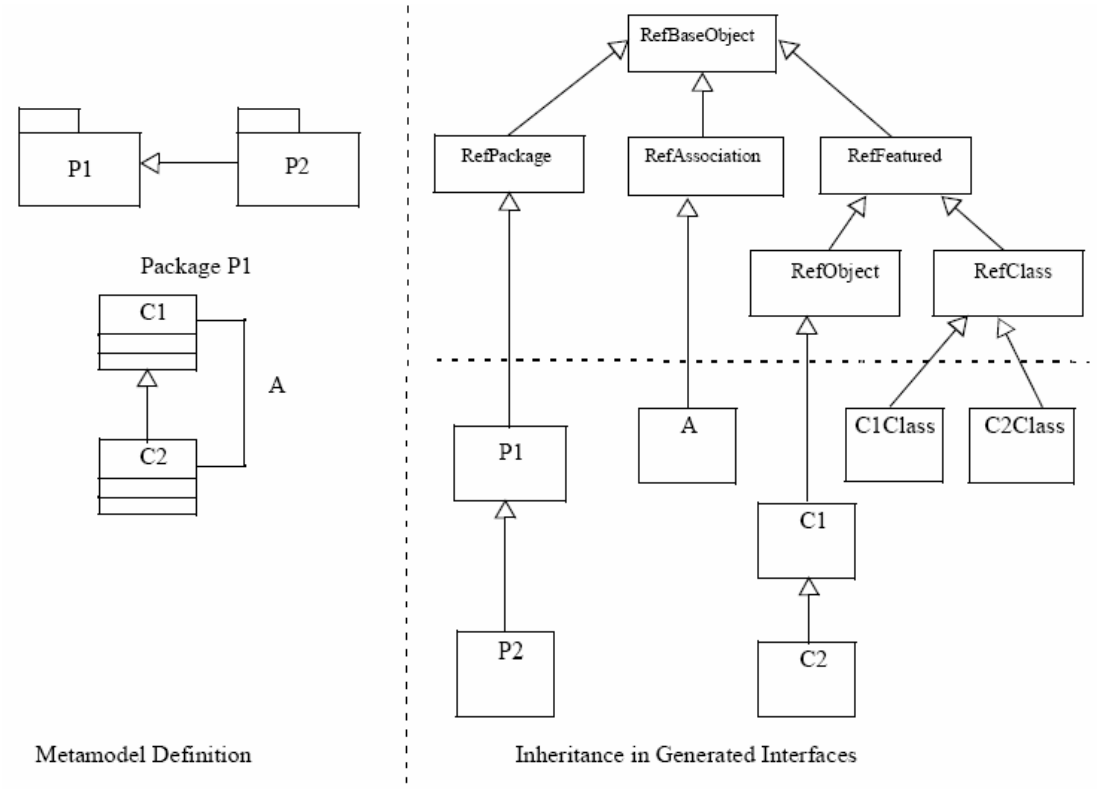

Figure 8 [JMI02] Generated Inheritance Patterns

Figure 8 above shows an example metamodel on the left, and the generated, "tailored" interfaces on the lower right side, with the inheritance hierarchy of the six most important reflective classes on top of it.

The JMI Reflective Package is especially useful if a program, such as a generic model transformation engine, has to deal with unknown metamodels. The implementation of the *Marius* tool (see Chapter 4) complementing this work is solely based on the reflective functionalities provided by JMI. The below code sample is taken from a transformation implementation generated by this engine. It shows how to use reflection to instantiate a new CallAction object located in the Common_Behavior package, assuming targetPackage corresponds to an instance of a UML root package.

```
javax.jmi.reflect.RefObject CallAction_ =
(javax.jmi.reflect.RefObject)targetPackage.refPackage("Common_Behavior")
.refClass("CallAction").refCreateInstance(null);
```

The second code piece shows how a link between the objects source and CallAction_ is added to an A_state_entry association within the State_Machines package.

```
targetPackage.refPackage("State_Machines").refAssociation("A_state_entry")
.refAddLink((RefObject)source , (RefObject)CallAction_);
```

2.6. XMI – XML Metamodel Interchange

XMI is an XML based language by the OMG for the exchange of metadata. It provides a mapping from MOF to XML and thus standardizes the way any kind of metadata, based on the MOF meta-metamodel, can be interchanged between different modelling tools. For validating UML-XMI and MOF-XMI files, mappings are available from the respective metamodels to DTDs, and with version 2.0 as well to XML Schemata [XMI03].

The code below shows the XMI 1.2 serialization of a single UML class. It also contains a header and versioning information, which is not directly part of the exported model.

```
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp =
'Tue Aug 10 20:48:13 CET 2004'>
        <XMI.header>
                <XMI.documentation>
                        <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
                        <XMI.exporterVersion>1.0</XMI.exporterVersion>
                </XMI.documentation>
        </XMI.header>
        <XMI.content>
            <UML:Class xmi.id = 'sm$59a735:100d31e7a35:-7ff6' name = 'Class_1'
             visibility = 'public' />
        </XMI.content>
</XMI>
```

Chapter 3

Model Transformation Concepts

In 2002 the OMG launched the Query/Views/Transformations-RFP [QVTR02] in an effort to standardize a transformation language for MOF-based metadata. This chapter explains the general concept behind model transformation and compares different transformation approaches. These approaches all rely on different kinds of software infrastructures, like metadata repositories, XSLT processors or most commonly a programming language such as JAVA. The following sections discuss these technologies and their relevance to the various transformation approaches introduced.

The model transformation implementation complementing this work takes a generic, transformation language driven approach, supported by the *Marius* transformation engine. Henceforth special emphasis is put on this topic, describing the necessary software infrastructure and the role played by the transformation engine.

3.1. Different Approaches to Model Transformation

Model transformation comes in many different colours and flavours. Since it is a technology finding application in different domains, often the same terms are used with different meanings in different approaches. Nevertheless all these approaches do have several characteristics in common. Following [CZAR03], the following

paragraphs describe these characteristics and categorize different approaches thereafter.

- *Source/Target:* A transformation uses a source model and transforms it into a resulting target model. Source and target are often quoted as “left-hand side” and “right-hand side”.
- *Selection Patterns:* A pattern is a model fragment consisting of one or more model elements. Patterns are typically used to “select” constructs from source and target domains to carry out transformation steps on them. An XSLT rule applying to a number of model elements is a fitting example.
- *Transformation Logic:* All means to express constraints or computations in a transformation fall into this category. An example would be an OCL [OCL20] constraint on a model element or a JAVA language construct instantiating a new model element on the target.

Other not necessarily mandatory but widespread characteristics include *Variables*, *Scoping*, *Traceability* and the ability to *fine-tune* transformations. Variables are actually metavariables, used for storing intermediate results during a transformation’s execution. Scopes applied to a model limit the range in which transformations apply and parameterization can tune a transformation execution to special needs. Traceability (keeping track of links between source and target models and their transformations) can either support the transformation definition process itself or support impact analysis, model-based debugging and synchronization of models.

Furthermore, a general distinction between *model-to-code* and *model-to-model* transformations can be made, whereas the former can be considered as a model-to-text transformation producing textual artefacts instead of a target model.

One way to drive a model-to-code transformation is a rather direct approach making use of the *Visitor Design Pattern*. The internal representation of the source model is traversed and specific code for each node is produced. Another way of code generation is a *template-based approach*. A template consists of target-text with interwoven bits of metacode, which access the left-hand side to retrieve modeldata and control code selection and iterative expansion. [CLE01] The advantage of template-based code generation is that the development of templates can be aided by existing examples of the intended target. Translation – the actual model-to-text transformation - of the templates is taken care of by a template engine, such as Jakarta’s Velocity [VELO04] or EMF’s *Java Emitter Templates* [JET04]. The author of this work advocates the opinion that the overall better readability and maintainability favours a template-based approach. An example for such an approach

is the translation of *Marius* transformation definitions into JAVA source code, driven by *JET* templates, as discussed in section 4.4.4.

Model-to-model transformations play a vital role in the Model Driven Architecture. To gain several layers of abstraction ("zoom-in"/"zoom-out"), transformations between models have to be possible and the rules for these transformations should be kept separate from the models. These transformations can drive a separate mechanism that automatically generates a target model from a source model. Since model-to-model transformation is a more complex and challenging field, the following sections are dedicated to describe different approaches to it. The approaches are categorized after the way the transformations are defined, ranging from the immediate implementation in a programming language to a high-level, generic definition language.

3.1.1. Direct Approach

The only infrastructure a so-called "direct approach" requires, are in-memory data structures representing source and target models, and an API, such as JMI for JAVA, to query and manipulate them. The transformations have to be implemented manually and there is no support for any kind of automation or organization of transformations. A direct approach is easy to set up, but does not prove feasible for more complex challenges, nor does it prove helpful in terms of abstraction of layers and maintainability of transformations.

3.1.2. XMI/XSLT Based Approach

With XMI being an XML based language, and XSLT being a standardized technology for processing XML documents, the idea to implement model transformations using XSLT technology seems promising. The problem however is that developing and maintaining transformations in the form of XSLT style sheets is quite cumbersome due to the verbose nature and poor readability of XML and/or XSLT. However, there are approaches to generate XSLT from more high-level transformation definition constructs to overcome these issues [PGB01]. Nevertheless, this approach's major drawback is its reliance on previously serialized XMI documents. For large models these files become huge and are the source of performance problems. A UML model consisting of a class diagram with ten classes and ten associations, and an activity diagram with twelve states and eleven transitions, sums up to a total of 5255 lines or a file size of 258 KB (XMI version 1.2, Netbeans XMI Writer 1.0, Diagram Interchange Package included).

3.1.3. Metadata Repository-based Approach

This approach requires a metadata repository capable of hosting source and target metamodels and their instances, and a programming API, JMI for instance, to manipulate the repository contents. The transformations between source and target elements are meta-modeled in a CASE tool and exported to XMI. From here, the JMI interfaces are generated and the transformation logic, meaning the methods defined in the transformation model, are manually implemented. This approach has the advantage of using meta-modeled transformations, of which the overall transformation framework can be generated from. Still there are limitations to the expressiveness in which transformations can be described in a CASE tool. The relationships between source, target and transformation model element are mere associations. Even though constraints on the model (OCL) can be enforced, the transformation logic for complex mapping rules has to be hand-coded. The use of a metadata repository providing an in-memory representation of the transformations artefacts makes this approach less likely to fall victim to poor performance, as compared to the XMI/XSLT based method above.

An example for this approach, mapping UML to BPEL4WS, is described in [GAR03B] and [IYEN03]. Included in the ETTK 2.1 [ETTK04] is a demo implementation ("UML 2 BPEL Demo") built on this idea.

3.1.4. Generic Model Transformation Approach

This approach is similar to the above one, in the sense, that it uses a metadata repository infrastructure for hosting source and target metamodels and their instances. But instead of requiring to manually complete the transformation logic, this approach automatically generates executable code from a transformation definition language. Such a language is specifically designed to capture the semantics needed to describe mappings between any kind of metamodels, which - when executed - transform their respective instance models.

The *Marius* transformation engine is built on the notion of generic model transformations. It translates transformation definition files into JAVA source files. After compiling, these classes make up the executable transformations that carry out the source/target mapping within the metadata repository, by utilizing the JMI Reflective API.

Other very interesting projects utilizing this kind of approach are those at [INR04], [ASTT03] and [DIC03]. They are all designed to answer OMG's QVT-RFP

[QVTR02]. Another quite promising submission to the QVT-RFP, which also provides a small Eclipse plug-in demo is [QSUB03] from [QPAR04].

3.1.5. Discussion

Of the previously mentioned approaches, the one employing Generic Model Transformation is of course the most desirable. Due to the expressiveness and separation of concerns between transformation and modeling, this approach will presumably yield the best results in terms of useability and acceptance in the community. Still the model transformation field is in its early days, and much more work in terms of exploring different methods and eliciting requirements has to be done. Although existing MDA tools, for instance [ANDR04], [OPTI04] and [ARC04], can perform very well in developing large scale applications, they are often built to support a certain target domain (EJB, Web Services, etc.) and are not based on a solid theoretical foundation. But in my opinion this shows even more the power that lies within the MDA paradigm, and should be a motivation to further pursue the standardization and development of generic model transformation - still the missing link in MDA.

3.2. Metadata Repository

To implement the MDA doctrine it is vital to have infrastructural support for storing and retrieving metadata. MDRs are designed to offer this kind of functionality. The foundation for a repository is some M3-level meta-metamodel, such as the MOF. An even more fundamental requirement is an existing mapping of such a meta-metamodel to a programming language API. Otherwise, there would be no standardized means to access the repository or even develop it in the first place. Metadata repositories fulfilling this criteria, can host any custom made metamodels and models, as long as all are instances of the top-level meta-metamodel the repository is built on.

Functionalities a repository offers include the import and export of models and metamodels, and programmatic access to manipulate its contents. However, further features like GUI support and code generation mechanisms do a great job at enhancing useability and making an MDR the backbone of any MDA-driven approach.

Below, two metadata repository technologies are introduced and their features are described briefly. Both are successfully applied in the community and provide similar

functionalities. However, each of them has been developed with a different rationale in mind, which certain distinct characteristics give away.

3.2.1. The NetBeans Metadata Repository

The Netbeans MDR [MDR03] is a metadata repository built on OMG's MOF and can be integrated into the Netbeans Tool Platform or used as a stand-alone application. With JMI being a mapping from MOF, it constitutes the standard programming API of the MDR. The generated JMI-compliant interfaces for any metamodel do not have to be implemented manually. MDR provides a default runtime implementation, which can be overridden with a custom implementation if the need arises. One of its main aspects is its persistence mechanism, which actually allows different implementations to store the metadata, and abstract the way this happens to the client. For instance, a B-Tree [BTREE] is used to persist metadata on a hard disk's file system. An in-memory storage mechanism for transient metadata is also available. Other methods like a JDBC-based approach are not contained by default, but can easily be plugged into the existing MDR architecture.

Figure 9 depicts the layered architecture of the MDR. The top-tier represents client programs using functionality in the form of the MDR-API and JMI, which abstract the underlying repository management mechanisms. The lowest tier shows the actual storage layer, which can be realized with different implementations, abstracted by a persistence interface.

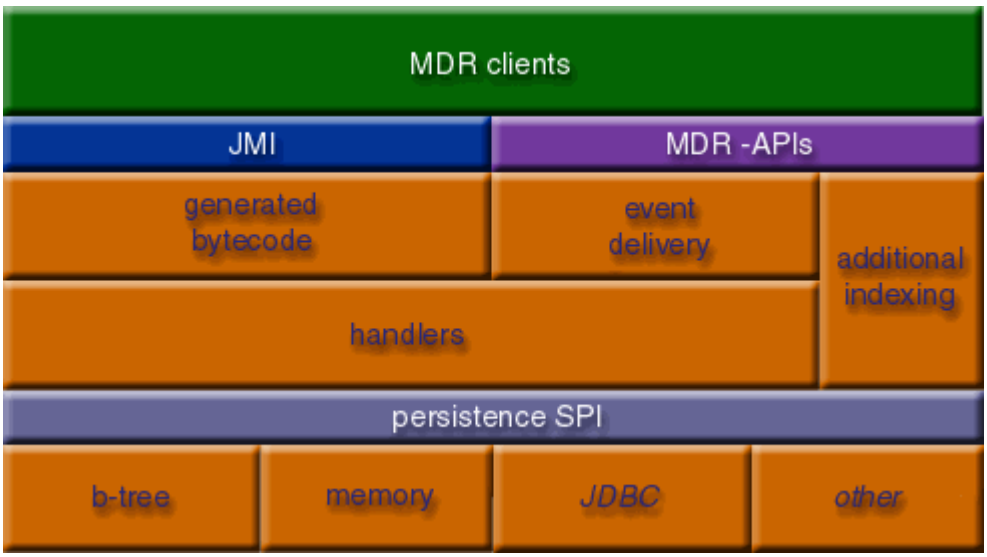

Figure 9 [MDR03] NetBeans MDR Architecture

As mentioned above, MDR offers functionality to integrate the repository into the Netbeans IDE. The NetBeans MDR Explorer is a product of this integrative capabilities. This GUI tool allows browsing and editing a repository's metadata-content represented as a tree structure within the IDE. The MDR project site also hosts a UML-to-MOF command line tool, to generate MOF-compatible metamodels from UML models designed in a standard CASE tool, as demonstrated in 4.4.2. Other key features of the MDR include:

- Import of XMI 1.1/1.2 for MOF 1.3 and MOF 1.4. with NetBeans XMI Reader.
- Export of XMI 1.2 with NetBeans XMI Writer.
- Generation of JAVA interfaces (JMI-compliant) for any hosted metamodel.
- Instantiation of any MOF-compliant metamodel.
- Automatic implementation of generated interfaces at runtime.
- Shell access for stand-alone usage case.

Whereas EMF as an example, is designed to integrate well with other tools and support model-driven development in an IDE, the NetBeans MDR is especially well suited for storage and archive purposes where possibly thousands of instances have to be persisted. This makes it a worthwhile background infrastructure for several professional software projects, the Poseidon UML CASE tool being one of them.

3.2.2. The Eclipse Modeling Framework

The EMF is, as the name gives away, an Eclipse Tools project targeted at enabling MDA-style development. It is a modelling framework based on Java/XML for building applications from data model. EMF can map such a model to JAVA implementation classes, as well as generate adapter classes for viewing and editing purposes. New models can be specified through either writing JAVA code annotated with comments containing instructions for the EMF generator, loading XMI serialized EMF models, importing a Rational Rose ".mdl" file, or generating the model from an XML schema.

The EMF is founded on the "Ecore" meta-metamodel, which is related but different from OMG's Meta Object Facility. MOF 1.4 and Ecore have been developed in parallel, and Ecore is essentially a subset of MOF 1.4 modelling concepts. It is streamlined to support the integration of modelling and code generation tools, whereas MOF's focus lies on building repositories to persist and manage metadata. The relationship between MOF and Ecore, is similar to the one between JMI and the

EMF Java mappings. Those have been optimized towards in-memory tool integration, while JMI is favourable for metadata repository usage scenarios.

The Eclipse Modeling Framework consists of three parts, at its heart being the *EMF Core.* It provides the above mentioned Ecore metamodel, change notification for metaobjects, a persistence mechanism based on XMI serialization and a reflective API (similar in concept to the JMI Reflective API) to generically manipulate models. Then there is the *EMF.Edit* and the *EMF.Codegen* framework that provide generic classes and code generation facilities to build working editors for EMF models. When changing the underlying model and regenerating the model code, the previously customized code stays unaltered and will be merged with the new version. The code generation EMF offers, is layered in three tiers:

- *Model code:* Interfaces and implementation classes corresponding to the model, as well as a factory to produce instances of the implementation classes.
- *Adapter code:* Implementation classes that adapt the model classes for viewing and editing purposes.
- *Editor code:* Constitutes a basic Eclipse-GUI style editor which serves as a starting point for further customization.

The EMF is an excellent framework for developing model-based applications within Eclipse, though stand-alone applications built on an EMF model can also be run. A developer can get from designing an application's business logic to running a rudimentary version of it, without writing a single line of code. Especially with Eclipse's 3.x new support for Rich Client Platforms, EMF's model-driven capabilities become even more appealing when developing a GUI-based application.

Since the time this diploma thesis was started the EMF has evolved tremendously, and a number of handy extensions and plug-ins have appeared on the map. In terms of model transformations, among others mentioned in chapter 6 "Related Works", there is [TEFKAT], the MTL Transformation engine at [INR04], and most notably IBM's just recently released Model Transformation Framework [MTF04].

One can only welcome the fact that there is a lot of work going on in that respect, but the fact that EMF is not compatible to MOF and JMI, does not favour interoperability – an MDA key goal - in model-driven systems development. Even though EMF/XMI and MOF/XMI are not interchangeable, there has been some research done by [DGR03] and [AGKR] on how to cope with that problem. The main problem obviously is, that a transformation from MOF to EMF is "lossy", because EMF is a subset of MOF and therefore less expressive.

3.3. Transformation Framework

The following sections describe the various parts that make up a suitable framework capable of a transformation language based approach to model transformation. Such a framework utilizes a metadata repository for hosting metadata and a transformation engine. Depending on the form of the transformation definition language, the transformation definitions may have to be translated into a language able to manipulate the repository. Typically, this is the case when the repository is accessible with a programming language, such as JAVA for MDR. The framework provides access to the metadata repository and controls the translation described above, as well as the subsequent transformation execution.

3.3.1. Defining and Hosting Metamodels

Every metamodel to be stored in a metadata repository has to adhere to the very meta-metamodel the repository is based on. Thus, metamodels (e.g.: UML, CWM) used as source and target domains in model transformation have to be instances of a common meta-metamodel, typically MOF or Ecore. Creating custom metamodels is an activity aimed towards creating an instance of a meta-metamodel and loading it into a metadata repository. The modelling part preferably happens in a graphical editor, just like the definition of models in a CASE tool. Exporting the metamodel and loading it into the repository makes use of XMI, the standard metadata interchange format. Even though modelling tools are traditionally based on UML and not MOF, that does not necessarily render a standard CASE tool unsuitable for metamodelling. Netbeans provides the *UML2MOF* [U2MOF] command-line tool to convert UML-XMI into MOF-XMI, which is then compatible with Netbeans MDR. The EMF also offers to generate Ecore models from annotated JAVA and XML Schema. A rather cumbersome and impractical approach at defining metamodels would be to write an XMI file manually.

Once a metamodel has been created, its XMI file can be imported into a metadata repository. From this point onwards, all the functionality offered by the particular repository can be applied to the metamodel. Most importantly, this includes generation of programming interfaces, namely JMI and the EMF Java mappings respectively.

3.3.2. Transformation Definitions

The transformation definitions describe how a source metamodel relates to a target metamodel. Thus, they specify how model elements or sets thereof correspond to each other in either domain. A transformation definition language is a metalanguage

defining the syntax in which these definitions are written. Generally, a transformation language has to offer semantics to capture specific needs to describe relationships between metamodels. Useability and expressiveness as well as preferences of the intended user groups, nevertheless influence the actual way in which a transformation language manifests.

To date several transformation languages exist and are subject to further research and development, as no standard has emerged yet. These languages are mostly textual notations based on declarative and/or imperative constructs, as it is the case with the *Transformation Rule Language* [ASTT03], *ATL* and *MTL* from [INR04]. However, there are also approaches relying on the theoretical work on graph transformations like *GreAT* [AGRA04], [AGRA03] and *VIATRA* [VAPA02]. A graphical transformation language specialized for business process models is proposed by [MMGK04].

Whatever the specific syntax of a transformation language might be, this work advocates the opinion that a transformation definition has to expose a few common key characteristics:

- *Source/Target language references:* References as to what the source and target metamodels are. Depending on the transformation framework, these references could for instance point to metamodels hosted in a repository.

- *Source and target:* The source and target are metamodel elements or sets thereof which are related through transformation definitions.

- *Transformation parameters:* To enable fine-tuning, parameters can be passed that affect a transformation's execution. An example thereof would be a parameter to distinguish between "create" and "update" behaviour when executing a transformation.

- *Source and Target pre/post conditions:* Conditions that have to be met prior to the execution of a transformation.

- *Mapping rules:* A transformation definition is made up of mapping rules, where each rule maps source model elements to target model elements. Querying an attribute value of a source model element and updating an attribute in a corresponding target model element poses an example for this.

The *Marius* transformation language encompasses the above mentioned characteristics. As a textual, hybrid transformation language it allows a mix of

declarative and imperative constructs. In model transformation mapping rules can be expressed using declarative constructs carrying an implicit meaning, since the mapping's semantics are intuitively understandable. An example would be the aforementioned "query-and-update" mapping. However, for more complex types of mapping rules it may be more applicable to resort to imperative, "programming language-like" constructs. Encapsulating too complicated semantics might waiver the advantage of the rather concise, declarative constructs over the generally more verbose imperative constructs, as much thought has to be put in understanding their implicit meaning. Finding a good balance between the declarative and imperative nature of a hybrid approach is deemed important for its usability, as further elaborated on in chapter 4.

3.3.3. The Transformation Engine

At the heart of a transformation framework lays a transformation engine, which executes transformation definitions and produces a target model from a source model. This implies that the transformation definitions have to be in an executable form and that an instance of the source metamodel (for an update operation or a source/target conformance check a target model will also be required) is available. During the translation step, it is recommendable for a transformation engine to perform compiler-typical error checking on the transformation definitions, such as syntax and type checks, variable declarations and so forth.

At execution time, the engine has to manage the transformation process' control flow. Depending on the type of transformation language used, the requirements for a transformation engine may differ. For instance, an engine for a language not enforcing explicit rule ordering, will have to determine a correct and conflict-free execution order. Other tasks transformation engines will typically perform, involve resolving conditions applying to transformations, calling sub-transformations and taking care of traceability concerns.

In case of the *Marius* transformation engine, the transformation definitions are translated into JAVA source files, which after compilation constitute the executable transformations. Refer to the next chapter for a more detailed description of *Marius'* features and its architecture.

Chapter 4

Marius Implementation Details

This chapter describes the *Marius* transformation tool and explains how the components it consists of interact. *Marius* is a prototype implementation developed for gaining hands-on experience in generic model transformation and is therefore not meant to be a response to [QVTR02]. Refer to "Future Work" section 7.5 for a breakdown of the features that still need to be implemented to satisfy the QVT-RFP.

The name "Marius" stems from Gaius Marius, a roman consul and general, best known for initiating a series of reforms in 107 BC, completely restructuring the organisation, equipment and tactics of the roman army. These "Marian Reforms" significantly changed the face of the roman legions and transformed them into a political factor sustaining the power of the late republic and the rise of the Caesars.

4.1. Marius Architecture

As to be seen in Figure 10, three major components make up the Marius transformation tool. Those are the Netbeans MDR, the Marius transformation engine and the Marius transformation framework.

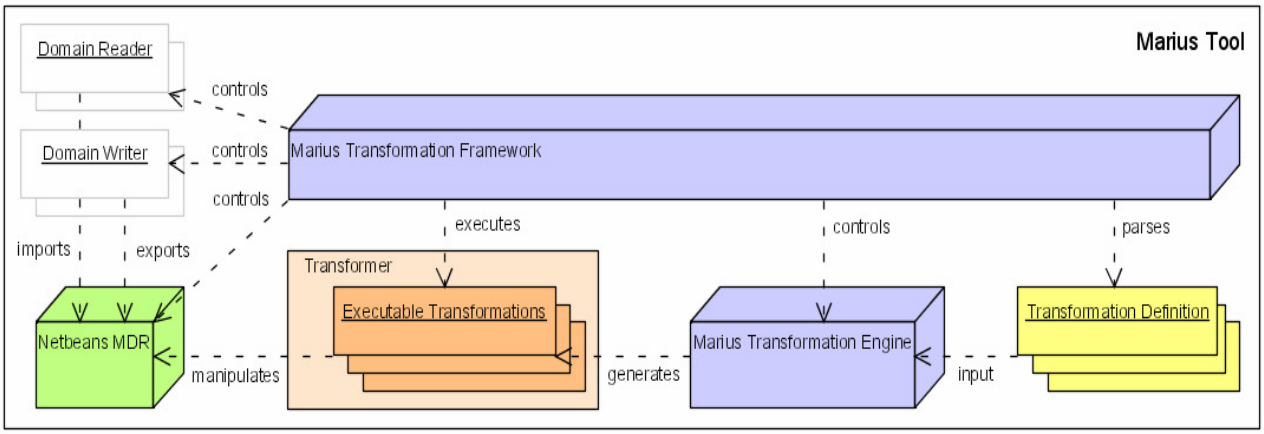

Figure 10 Components of the Marius Tool

Other components that do not directly belong into the immediate setup of *Marius*, are readers and writers for source and target documents. An example thereof are the readers and writers for BPEL documents described in 4.3.1. and 4.3.2.

4.1.1. Marius' Metadata Repository: Netbeans MDR

The Netbeans MDR [MDR03] is incorporated in Marius as a stand-alone version, independent of the Netbeans IDE [NBIDE]. Favouring MDR over the EMF as Marius' metadata storage was a decision mainly based on the fact that MDR is compatible with MOF and JMI. Both carried by standardization groups enjoying broad acceptance in the community, whereas the EMF relies on the Ecore meta-metamodel and its own JAVA mappings. The EMF also offers code-generation mechanisms for the development of model-based applications, overall streamlining it for functionalities not necessarily being a priority for Marius.

Prior to the actual transformation execution, the transformation framework imports source and target *metamodels* in the form of XMI and creates instances in the repository. Then, the source *model* is either imported from metamodel specific XMI, or a domain specific reader (see 4.3.1) creates the model from an M0-level model instance. After this procedure (metamodel and model instantiation), the repository is ready for transformation execution, which finally generates the target model. Netbeans MDR provides XMI readers and writers for the purpose of model and metamodel import and export. However, to produce domain specific M0-level artefacts, a specialized writer for codegeneration (see 4.3.2) is needed.

4.1.2. Marius Transformation Engine

Doing the chores at *Marius*' core is the transformation engine. During the translation step, as can be seen in Figure 11 below, the transformation definitions are input to a code-generation mechanism based on *Java Emitter Templates* [JET04]. Via such a JET template, the JAVA classes representing the executable transformations are generated. The engine reads its input - the *Marius* transformation definitions - with a *SableCC*-generated [SABL] parser, which is automatically built according to the *Marius* transformation language grammar. This includes a check for syntactical correctness of the transformation definitions, a semantical analysis however is not performed. The use of a non-declared identifier for instance will not result in a translation error, but leads to an unsuccessful compilation of the JAVA transformation files. For a more detailed description of the technologies and activities involved in the translation step, see section 4.4 below.

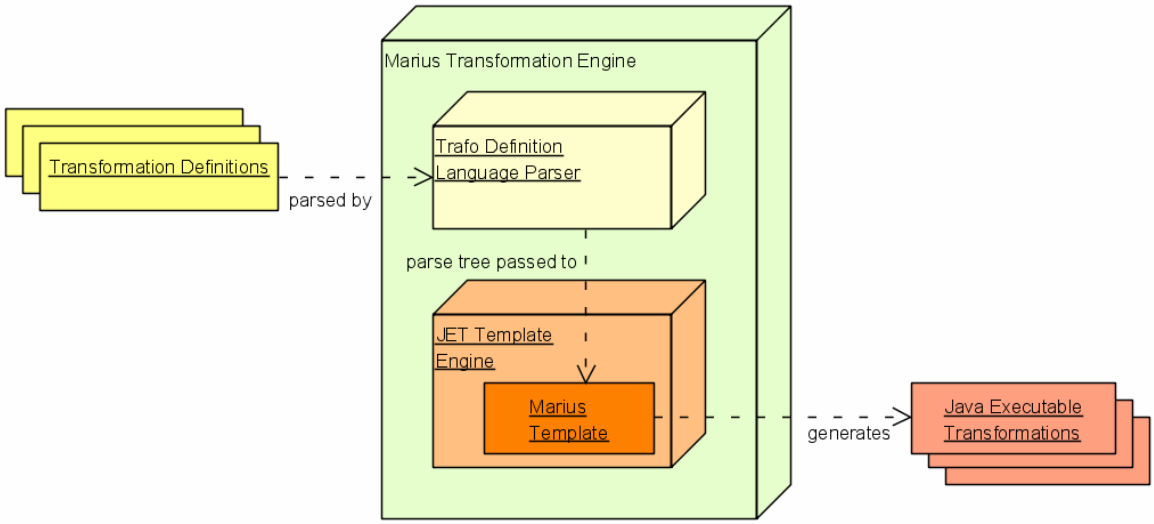

Figure 11 The Marius Transformation Engine

As previously mentioned, the *Marius* transformation engine translates the transformation definitions into a set of JAVA classes. Control flow managing functionalities that a transformation engine exerts during runtime, as mentioned in section 3.3.3, are also encapsulated in these executable transformations. This makes the set of executable transformations independent of any entity controlling their execution. Thus, they constitute a domain specific transformation engine themselves, executable as a stand-alone component, as it happens within *Marius'* transformation framework.

4.1.3. Marius Transformation Framework

The Marius transformation framework holds the MDR, the transformation engine and the domain specific readers and writers together. It is implemented in JAVA and drives the model transformation process, as it offers means for MDR management, transformation engine control and the use of domain specific readers and writers. Below is a compilation of key functionalities, which are further elaborated on in following sections in this and the following chapter.

- Instantiation of MOF-XMI compatible metamodels. (see 4.2.3.)
- Instantiation of a model from XML documents using XSLT. (see 4.3.1 and 4.3.2)
- Import and export of domain specific XMI-based models. (see 4.3.3 and 4.3.4)
- Generation of JAVA transformation classes. (see 4.4.4)
- Execution of transformation classes. (see chapter 5)

The fact that the repository access logic is contained within the JAVA transformation classes hard-wires the Marius transformation engine specifically to the Netbeans MDR. If the transformation engine were to be used with different repositories or means of metadata storage, adapters within the framework were needed to abstract repository access from the transformation engine.

4.2. How to generate metamodels

Model transformations are defined on a metamodel level, between a source and a target metamodel. MDA relies heavily on the use of different kinds of metamodels to support representation and abstraction levels of different systems. To define a custom metamodel one can resort to UML's extension mechanisms. These allow defining and making use of new modelling constructs not native to UML, thus enabling to create a new UML-based language. Such a UML "dialect", officially called a UML profile, is specified by the set of extensions that apply to it. UML offers three different kinds of extensions for profiling:

- *Stereotypes:* A model element can become a stereotype essentially by adding a text string enclosed between '«' and '»' to its traditional representation. The information content of the model element stays the same, but the stereotype indicates a specialized meaning or behaviour.

- *Tagged Values:* A tagged value is a pair of strings that can be applied to any UML model element describing an additional property. One string holds the

property's name, whereas the other contains its value. Tagged values prove useful in enriching models with information required for subsequent processing purposes, such as code generation, project management and of course MDA.

- *Constraints:* Restrictions and relationships beyond the notation of UML can be expressed with constraints attached to model elements. A standardized textual language used for this is the *Object Constraint Language (OCL).*

Profiling however, is not the only way to create custom metamodels. Just like UML, metamodels can of course also be defined using MOF. This approach can be considered as a heavy weight approach, yielding a metamodel definition based upon the full semantic depth of MOF, but also requiring a MOF-compatible modelling tool. A light-weight profiling approach on the other hand, although it suffers from less expressive power, has an advantage in its applicability with a generic UML tool.

The following sections (from 4.2.1 to 4.2.4) describe the use of metamodelling with MOF in more detail, and explain its application practically, as the *MyBPEL* metamodel used in this work has been created accordingly.

4.2.1. Using a CASE tool for metamodelling

Most modelling tools will not allow to work with MOF, but with UML. Nevertheless, a standard CASE tool can become a MOF-compatible metamodelling tool. The notation of UML 1.x and MOF is very similar, and by obeying a few rules, a metamodel can be edited in a UML environment and later on be translated into MOF. These rules are based on UML 1.4 and make up the *UML-Profile for MOF* [UPMF04], which enables a modeller to represent MOF model elements that do not have a straight-forward mapping from UML. Although the specific rules and guidelines on how to specify a MOF model in UML can be found in the profile, as a rule of thumb it is obvious to avoid UML features not available in MOF. This includes refraining from the use of "n-ary" associations, association classes, as well as dependencies and qualifiers. Additional to these concerns, there are other metamodelling virtues to be taken into account. Following [FRAN03], these mainly deal with optimizing a metamodel for subsequent generation of a compilable model (e.g.: JMI) in a programming language:

- *Define important operations only:* Accessors and mutators for properties, as well as factory operations for instantiation will be generated automatically. Concentrate on defining "interesting" operations that provide functionality for the target application and are not implicit in the structure of the model.

- *Proper use of association end navigability:* Only make association ends navigable when needed so. Otherwise unused accessors, mutators and properties will clutter the target code.

- *Avoid name clashes:* A navigable association end becomes a property in the opposite class. Therefore association ends opposite to the same class cannot have the same name.

- *Carefully specify multivalued properties:* MOF and UML allow to impose "ordering" and "uniqueness" on multivalued properties. In JMI the ordered tag will result in a java.util.List, rather than a java.util.Collection. The isUnique tag enforces set semantics and does not allow duplicates in a multivalued property. Modelling tools usually provide default values for ordered and isUnique, so care has to be taken not to overlook an undesired setting.

- *Use of Abstraction:* Abstract classes cannot be instantiated, hence no factory operations will be generated which results in smaller, "cleaner" APIs.

- *Syntactic and semantic completeness:* MDA generators rely on fully defined models, so ensure to specify all types of operations and attributes, names of associations and multiplicities of association ends. Always reflect on whether the models produced by this metamodel expose the intended behaviour.

An example for a metamodel defined in Poseidon is the *MyBPEL* metamodel, which can be found in the appendix. To get started with metamodelling, the Netbeans MDR project page offers templates for the modelling tools Poseidon and MagicDraw. These templates serve as empty project files with the standard stereotypes and tags according to the *UML Profile for MOF* already in place.

4.2.2. Converting UML to MOF

After a metamodel has been defined in a UML tool (e.g.: Poseidon), it still has to be translated into MOF. To do this, the model has to be serialized into XMI first, implying that the modelling tool supports this export mechanism. The resulting XMI file describing a UML model, now has to be converted into an XMI file describing an equivalent MOF model.

The Netbeans MDR project provides the previously mentioned *UML2MOF* command line tool that facilitates the above described translation from UML-XMI to MOF-XMI. It reads the UML-XMI input file using Netbeans XMI reader, uses JMI to

implement the mapping to MOF programmatically, and finally exports the resulting MOF-XMI file with Netbeans XMI writer. The tool is founded on OMG'S *UML-Profile for MOF,* but differs in minor ways [NBUM].

4.2.3. Instantiating a metamodel in the Netbeans MDR

Any MOF-compatible metamodel can be instantiated in the MDR. After setting up a repository, a MOF model package needs to be instantiated, which will then hold the metamodel. The example code below shows how to programmatically prepare a repository, create an extent (MOF model package) and load a metamodel in XMI form ("metamodel1_mof.xmi") into it.

```
repository = org.netbeans.api.mdr.MDRManager.getDefault().getDefaultRepository();

ModelPackage metamodel1 =
(ModelPackage)repository.createExtent("METAMODEL1");

reader = org.netbeans.api.xmi.XMIReaderFactory.getDefault().createXMIReader();
File f = new File("metamodel1/resources/metamodel1_mof.xmi");
reader.read(f.toURI().toString(), metamodel1);
```

Now, that a metamodel exists in the repository, instances of this metamodel can be generated. To do this, the metamodel's root package has to be found. Assuming the above loaded metamodel, now nested in the MOF model package "METAMODEL1", contains several packages with "MetaModel1" being the root package, a reference "root" can be obtained like this:

```
MofPackage root = null;
for (Iterator it = metamodel1.getMofPackage().refAllOfClass().iterator(); it.hasNext();) {
        MofPackage pkg = (MofPackage) it.next();
        if (pkg.getContainer() == null && "MetaModel1".equals(pkg.getName())) {
           root = pkg;
        }
}
```

The following piece of code finally creates a MOF model package holding an instance of the located "MetaModel1" package belonging to the previously loaded metamodel.

```
metamodel1_instance =
(MetaModel1Package)repository.createExtent("METAMODEL1_INSTANCE", root);
```

From thereon the model can be populated either programmatically using the JMI interfaces or by de-serializing an XMI model.

4.2.4. Building JMI Interfaces

JMI is a specification enabling a mapping from MOF to JAVA. The produced interfaces serve in accessing, querying and manipulating metadata. However, to work with JMI, one is not required to generate domain specific interfaces. The JMI reflective package allows to build fully generic applications. *Marius* for instance, uses these reflective capabilities and is able to work with any MOF compatible metamodel.

The generation of JMI interfaces for an extent in the MDR can happen either programmatically or by using the MDR Explorer, a GUI application providing repository management functions.

4.3. Reading and writing source and target documents

In MDA, transformations are typically defined between metamodels on level M2, and are executed on instances in the M1 model layer, according to the common four-layered metamodelling paradigm. The QVT initiative aims at that scenario, and is not concerned about transformations on lower levels or in between levels. Transformations involving the M0 data level are not possible due to the fact that the "instanceOf"-relationship between level M0 and M1 is not specified in MOF, as this is obviously an implementation specific issue. It can be argued that there are only three metamodelling layers instead of four [JBRL97], because only M2 and M1 entities are "real" instances of layer M3 [IKKB04].

To bridge the gap between model layer M1 and data level M0, domain specific readers and writers have to be employed. Such a reader's and writer's behaviour implicitly defines the "instanceOf"-relationship between M1-model and M0-instance. Below are descriptions of the readers and writers used to transform MyBPEL to UML. In the course of this transformation, the Marius Transformation Framework uses the Netbeans XMIReader [MDR03] to instantiate metamodels in the MDR, the MyBPELReader to parse BPEL source documents and the Netbeans XMIWriter [MDR03] to serialize the generated UML model. Not involved in this scenario is the MyBPELWriter, which would only be of use in a transformation running the opposite direction.

4.3.1. MyBPELReader

Starting point of the BPEL2UML transformation process is a BPEL document. MyBPELReader is a JAVA implementation using Xalan [XALA04] to parse this

document and create a model instance within the MDR. With BPEL being an XML dialect, the parsing can greatly be enhanced by using XSLT. To do that, an XSLT sheet matching the various BPEL constructs is needed. Within the specific templates, calls to the MyBPEL JMI interfaces populate the MDR accordingly. In this usage case, the XSLT engine does not produce textual output as it usually does when transforming an XML file for instance. The output is a MyBPEL model instance located in the MDR, representing the parsed BPEL document.

Below is an example of an XSLT template matching BPEL's partners construct and making a JAVA call to instantiate the according Partners class. Then, for each contained partner element, a Partner instance is created and the setName method is called with name as a parameter.

```
<xsl:template match="bpws:partners">
   <xsl:variable name="partnersClass" select="java:getPartners($meta1)" />
   <xsl:variable name="partners" select="java:createPartners($partnersClass)" />
   <xsl:for-each select="bpws:partner">
      <xsl:variable name="partnerClass" select="java:getPartner($meta1)" />
      <xsl:variable name="partner" select="java:createPartner($partnerClass)" />
      <xsl:if test="@name">
         <xsl:if test="java:setName($partner, @name)" />
      </xsl:if>
    </xsl:for-each>
</xsl:template>
```

The advantage of the XSLT based parsing approach lies within the way XSLT allows a developer to describe operations carried out on XML data. An equivalent JAVA implementation based on DOM or SAX for instance, might prove harder to read and maintain. Furthermore, XSLT sheets are more flexible concerning changes, as no further compilation is required. In case various versions of an input language exist, which are similar enough to be represented by the same metamodel, different XSLT style sheets can be utilized to easily "switch" between language versions. In the course of this work, this proved practical when supporting BPEL versions 1.0 and 1.1. Mostly differing in names for elements and attributes, but not in their overall structure, these two versions can be represented by the same MyBPEL metamodel. The apparent disadvantage of an XSLT-based parser is its relative poor performance. However, this circumstance can be accepted due to the fact, that neither the

implementation proposed in this work, nor the Netbeans MDR in general is a performance critical application.

4.3.2. MyBPELWriter

Although not necessarily within the scope of this work, but nevertheless useful to complete the roundtrip from documents parsed with MyBPELReader, is a domain specific writer for BPEL. MyBPELWriter serializes an instance of the MyBPEL metamodel stored in the MDR to a BPEL document. The writer is implemented as a JAVA program iterating the model stored in the repository, and according to the model element it encounters, prints the according XML constructs into a file.

This is a very simplistic, straightforward approach to build a domain specific writer. Bridging the gap between M1 model and M0 instance layers is not trivial, and mapping to the data level always involves knowledge about the specific domain language in question. However, an attempt at describing a more structured method for writer implementation, especially those based on XML dialects, is taken in section 6.6, “Domain Specific Readers and Writers”.

4.3.3. Netbeans XMIReader

The XMIReader module is used to de-serialize data stored in an XMI file and load it into the MDR. Netbeans offers two standard implementations, building on SAX and DOM respectively. Both reader implementations are compatible with MOF 1.4 and support XMI versions 1.1 as well as 1.2.

4.3.4. Netbeans XMIWriter

The XMIWriter serializes data contained in the MDR to XMI 1.2 documents. XMI documents are produced according to the object containment hierarchy of the repository instance to be exported. Therefore, outermost composites in the metamodel map to XMI’s top level, as direct subelements of *XMI.content*. All other non-outermost composites map to subelements of their respective containers. Link ends belonging to non-composite associations map to references signified by *XMI.idref*.

4.4. From transformation definitions to executable transformations

The Marius translation process involves various technologies and document types to convert transformation definitions into executable transformations. Prior to the actual

translation, a parser for the Marius Transformation Language has to be in place, which is automatically generated with SableCC from *Marius'* Transformation Language grammar. This parser reads the transformation definitions and yields a parse tree that is passed on to a JET template engine. The JAVA source files, representing executable transformations, are the result of the JET engine processing a generic template for *Marius*' executable transformations by applying the information in the parsed syntax tree to it.

Figure 12 is an illustration describing the activities occurring in the translation process beginning with the generation of a SableCC parser from the *Marius* language grammar, which then reads the transformation definitions. In the next step, the parse tree is passed on to the JET template engine, which produces the JAVA output according to the *Marius* transformation template. This template is the central part of the translation, as its structure implicitly defines the mapping between the transformation definition language and the JAVA implementations of the executable transformations.

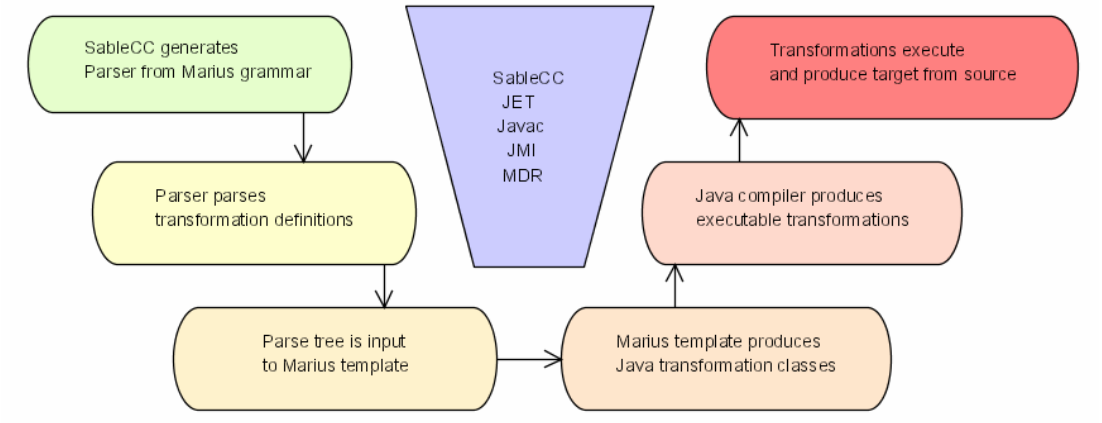

Figure 12 Generation of Executable Transformations

4.4.1. Marius Transformation Language Grammar

The lexical definitions and the grammar specifying the syntax of *Marius'* transformation language are written in EBNF and stored in a SableCC specification file. This text file adheres to a simple structure basically separating token definitions and grammar productions. The complete SableCC specification file defining the grammar for the *Marius* transformation language can be found in the appendix.

A shortened version to show the overall structure of a *Marius* transformation definition can be seen below. There are six sections, each one being introduced by a keyword accordingly. `Trafosource` and `Trafotarget` signify the declaration space for variables referring to the transformation's source and target model elements.

Auxiliary variables that do not belong to either source or target model, but which are used to store meta-information to handle the transformation execution are declared in the `meta` section. Every transformation also has a return value, which is a complex type containing source, target and the transformation's meta-information stated in `meta`. The `structs` section allows to declare data structures, mappings of MOF's StructType. Variables used to build up the target model or hold intermediary values are declared in the `variables` section. `Mapping`, the last section, contains all the transformation rules.

```
<title>

Trafosource
<variable declaration>

Trafotarget
<variable declaration>

meta
<metavariable declaration>*

structs
<structure declaration>*

variables
<variable declaration>*

mapping
<transformation rules>*
```

A *Marius* transformation definition adheres to the above depicted structure. The concrete example below shows a simple *Marius* transformation definition and explains the semantics and the usage of the language constructs in more detail:

Every transformation definition begins with a title naming the transformation, for instance the names of source and target model elements separated by '2'.

```
ActionState2GraphNode
```

The source declaration consists of three identifiers specifying a variable named `ActionState` referring to the `ActionState` model element in the `Activity_Graphs` package.

```
Trafosource
ActionState ActionState Activity_Graphs
```

Similar to `Trafosource`, the target model element is declared. If a transformation does not specifically refer to a target model element but rather initiates other sub-

transformations, a `Trafotarget` declaration can be omitted by using the '---' keyword.

```
Trafotarget
GraphNode GraphNode Diagram_Interchange
```

`Entry` and `Exit` are both declared as meta-variables. They have a collection-like behaviour and are able to store and group various model elements.

```
meta
Entry
Exit
```

Data structures are declared along a list of arguments.

```
structs
Point Point Diagram_Interchange ( Double 0 ) ( Double 0 )
```

Model elements to build up part of the target model can be declared in the variables section. Similar to the `Trafotarget` declaration, an instance of the specified model element will be produced at transformation execution time. If only an auxiliary variable is needed to reference an intermediary value, the package identifier can be left out and the model element's instantiation will be skipped.

```
variables
Uml1SemanticModelBridge Uml1SemanticModelBridge
Diagram_Interchange;
```

The final mapping section contains all transformation rules. Variables used have to be declared earlier. To understand the rules in the mapping section, refer to the section 4.4.2 for a concise description of the transformation rules employed by Marius.

```
mapping

$true =: GraphNode.isVisible;
Point =: GraphNode.position;

$"" =: Uml1SemanticModelBridge.presentation;

Uml1SemanticModelBridge <> ActionState
A_uml1SemanticModelBridge_element Diagram_Interchange;

Uml1SemanticModelBridge <> GraphNode
A_graphElement_semanticModel Diagram_Interchange;

ActivityGraph2Diagram:Diagram <> GraphNode
A_container_contained  Diagram_Interchange;

ActionState.entry -> *.GraphNode <> GraphNode
A_container_contained  Diagram_Interchange;
```

Every transformation rule can be preceded by a condition, determining whether the rule will be executed or not. The objects resolved from left- and right-hand side are compared by their equals method. Only if the comparison is successful, the subsequent transformation rule will be executed.

```
#Transition.name == Source.linkName# Transition <>
ActionState A_outgoing_source State_Machines;
```

Applying conditions to rules is often necessary to implement complex transformations that do not simply relate single source and target model elements, but build up a distinct structure in the target extent. Conditions preceding rules invoking sub-transformations act as their pre-conditions.

4.4.2. Marius Transformation Rules

Generally, all transformation rules assign values from the left-hand side to the right-hand side. Basic Expressions of the type *name.{sub-name}*, for instance `Process.name`, are used to qualify model elements and contained or associated model elements in respect to their owner. However, there are also other kinds of expressions available described later in this section, which are nevertheless similar in their concept of resolving model elements. Left-hand side and right-hand side are evaluated at runtime and in case an expression cannot be resolved, because a referred model element does not exist, an exception is thrown. The state diagram in Figure 13 shows how an expression is evaluated, and to which kind of model element the name *sub-name* can refer to, depending on the original type of *name* or the intermediary expression result respectively. Except Collections and primitive types (String, Boolean, Integer, Double), every model element referred to in an expression is extended from an interface belonging to JMI's MOF reflective package, thus the diagram is categorized according to these generic types. Meta-variables, declared in `meta`, are of Collection type. Additionally, every transformation returns a HashMap to its calling parent, which includes all meta-data, source and target model elements.

The states correspond to the type of the currently resolved expression - which initially is *name*, whereas the transitions originating from a state represent possible types *sub-name* can resolve to. A specialty is RefAssociation, which can resolve to Collection calling the `AllLinks` method (corresponding to JMI method refAllLinks()) returning all RefAssociationLinks governed by the RefAssociation in question. Similar to that, a call to the methods `FirstEnd` and `SecondEnd` (corresponding to JMI methods refFirstEnd() and refSecondEnd()) on a RefAssociationLink yields the RefObject connected by this link-end.

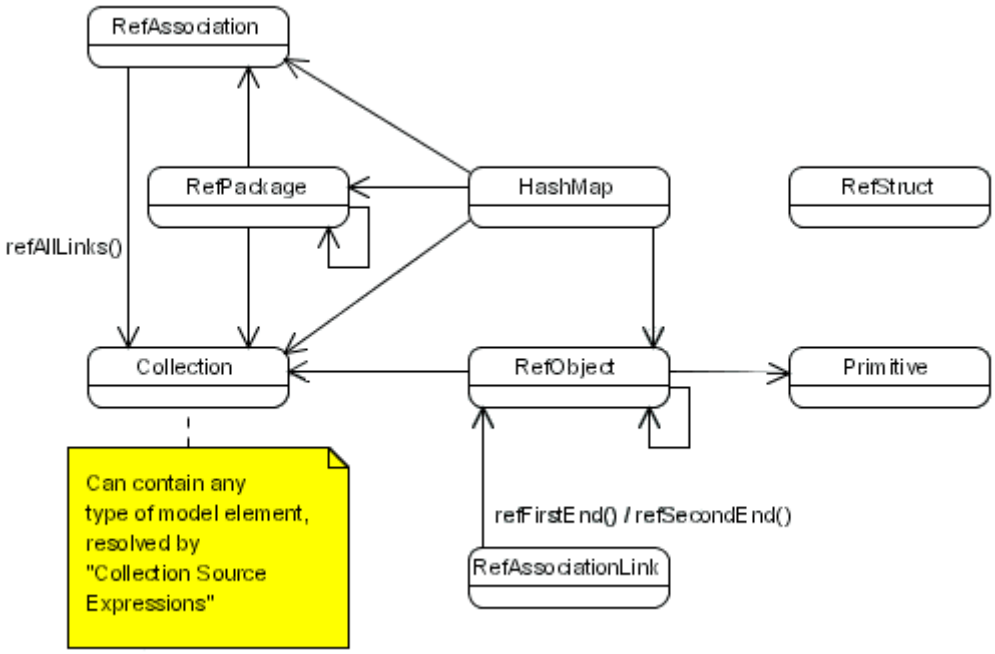

Figure 13 Rules to Resolve Marius Expressions

The example below shows how an expression could be resolved following the rules portrayed in the above state diagram. A Collection containing RefObjects, all instances of the "`ActivityGraph`" class, are the final result.

`UmlPackageSource` → RefPackage

`UmlPackageSource`
`.Activity_Graphs` → RefPackage

`UmlPackageSource`
`.Activity_Graphs`
`.ActivityGraph` → Collection of RefObjects

Marius transformation language expressions referring to model elements in the source repository build upon the above-described resolving mechanism. Other kinds of expressions are used to assign constants or to access collections. Below is a description of all valid left-hand side expressions referring to source model elements. Each description states the name and a shortened EBNF definition of the full production within *Marius*' transformation language grammar file. Following that is an example taken from a transformation definition and a list of possible instances the expressions can refer.

Name Source Expression

SableCC Name:`name_sourceexpr`

EBNF: `name {'.' sub_name} ['(' method ')']`

Example: `Invoke.name`

Resolves To: RefObject, RefPackage, RefAssociation, RefAssociationLink, HashMap, Collection, Primitive Types

A Name Source Expression uses *name* to specify a model element and *sub_name* to refer to its attributes or associations subsequently, as described in the resolving mechanism above. Just like the Collection Source Expression described below, a Name Source Expression may be concluded by the optional *method*, which specifies a method's name to be called on the resolved object.

Parent Source Expression

SableCC Name: `parent_sourceexpr`

EBNF: `parent ':' name`

Example: `Process2Model:Model`

Resolves To: RefObject, RefPackage, RefAssociation, RefAssociationLink, HashMap, Collection

A Parent Source Expression resolves a model element or a meta-variable scoped in a parent transformation, with *parent* referring to the transformations' name and *name* to the variable to be resolved.

Primitive Source Expression

SableCC Name: `string_sourceexpr`

EBNF: `'$' ( ('"' string '"') | bool | int | double )`

Example: `$"ActionState", $"true", $"123", $"1.23"`

Resolves To: Primitive Types

A Primitive Type Source Expression allows to define constants inline. The resolved instances are JAVA wrappers for primitive types.

Concatenated String Source Expression

SableCC Name: `concat_sourceexpr`

EBNF: `'(String)' '(' stringlet { '+' stringlet } ')'`

Example:
```
(String)(To.variable + "/" + To.part + " := " +
From.variable + "/" + From.part)
```

Resolves To: String

A Concatenated String Source Expression allows to build strings by concatenating stringlets, which are either string literals, or Name Source Expressions. In the latter case the expression will be resolved and mapped to a JAVA String by the referred object's toString() method.

Collection Source Expression

SableCC Name: `collection_sourceexpr`

EBNF:
```
collection '!' [offset] {'.' sub_name}
['(' method ')']
```

Example: `subStatesVector!0.Exit`

Resolves To: RefObject, RefPackage, RefAssociation, HashMap, Collection, Primitive Types

A Collection Source Expression is used to refer to elements inside a collection. It is similar to a Name Source Expression, but instead of name to refer to a root model element, it uses `collection` to specify the collection containing the root elements. The current root element is specified by the state of the iterator belonging to the referred collection. The optional offset can be used to refer an element relative to the iterator's current position. If offset is omitted, the last element in the collection is referred.

To finally assign resolved sources, *Marius* supports a number of right-hand side expressions referring to target model elements. The type of a target expression is determined by the transformation rule in place. Therefore, the various transformation rules supported by *Marius'* transformation engine and their respective target expressions are discussed below.

Assign Transformation Rule

SableCC Name: `string2equ, name2equ, parent2equ, concat2equ, colletion2equ`

Operator: *'=:'*

Source EBNF: *string_sourceexpr | name_sourceexpr | parent_sourceexpr | concat_sourceexpr | colletion_sourceexpr*

Target EBNF: *name {'.' sub_name}*

Example:
```
$1.0 =: Diagram.zoom;
Receive.name =: ActionState.name;
Copy2CallAction:ActionExpr =: CallAction.script;
(String)(Send.operation + "()") =: Activity.name;
 SubStateVector!0.Entry =: Entry;
```

The Assign Transformation Rule resolves a left-hand side source expression and assigns the resulting value to the right-hand side model element specified by the target expression. All possible source expressions are allowed to be used. The target expression refers to a model element by specifying a `name` referring to a variable, and optionally various `sub_name`, referring to owned elements.

Link Transformation Rule

SableCC Name: `name2normlink, parent2normlink, colletion2normlink`

Operator: *'<>'*

Source EBNF: *name_sourceexpr | parent_sourceexpr | colletion_sourceexpr*

Target EBNF: *name association package*

Example:
```
ActionState <> CallAction A_state_entry
State_Machines;
```

The Link Transformation Rule links source and target model elements via an association link. `name` specifies the model element to be linked with the source expression's resolved value. `association` is the name of the association residing in the `package` package, of which a link will connect source and target.

Sibling Transformation Rule

SableCC Name: `name2siblink, parent2siblink, colletion2siblink`

Operator: *'?>'*

Source EBNF: *name_sourceexpr | parent_sourceexpr | colletion_sourceexpr*

Target EBNF: *name association package*

Example:
```
Link.(FirstEnd) ?> GraphConnector
A_graphEdge_anchor Diagram_Interchange;
```

The Sibling Transformation Rule behaves equally to the previously mentioned Link Transformation Rule as it connects two model elements via an association link. The difference is that not the resolved left-hand side expression is linked, but its 'sibling'. A transformation definition's source model element declared in `Trafosource` has the according target model element declared in `Trafotarget` as a sibling. Therefore, the sibling look-up takes in a source model element and yields a target model element. If a resolved expression has no sibling, an exception is thrown. The above source expression `Link.(FirstEnd)` refers a source model element, of which the according sibling element will be linked with the `GraphConnector` model element.

Invoke Transformation Rule

SableCC Name: `name2arr, parent2arr, colletion2arr`

Operator: *'->'*

Source EBNF: *name_sourceexpr | parent_sourceexpr | collection_sourceexpr*

Target EBNF: *'*' { '.' sub_name} { (assign_righthandside | link_righthandside) }*

Example:
```
UmlPackage.Activity_Graphs.ActivityGraph -> *;
Links.link -> *.Transition =: Trans;
ActionState.entry -> *.GraphNode <> GraphNode
A_container_contained Diagram_Interchange;
```

Comment: The *Invoke Transformation Rule* is used to call sub-transformations. This rule's left-hand side expression is

resolved and for each instance of the referred model element found, a transformation with the according model element as a `Trafosource` will be invoked. A sub-transformation's return value (a HashMap containing variables declared in `meta`, `Trafosource` and `Trafotarget`) is represented by `'*'` and its elements can be referenced by *`sub_name`*. Variables of sub-transformations can be the left-hand side of either Assign or a Link Transformation Rules declared in-line.

Foreach Transformation Rule

SableCC Name: `foreach`

Operator: *'{ }'*

Source EBNF: *'foreach' ('array' | 'collection') source_expression collection*

Target EBNF: *'{' variable_declarations transformation_rules '}'*

Example:

```
foreach array Subs subsColl{
   variables
   mapping
   subsColl!0.Entry =: Entry;
   subsColl!.Exit =: Exit;
};


foreach collection subsVec!0.Exit exitVec {
  variables
  mapping
    foreach collection subsVec!1.Entry entryVec{
      variables
      Transition Transition State_Machines;
      mapping
      exitVec!0 <> Transition A_outgoing_source
      State_Machines;
      entryVec!0 <> Transition A_incoming_target
      State_Machines;
    };
};
```

Comment: The Foreach Transformation Rule is different from the other transformation rules as its purpose is not to directly manipulate the target model but to enclose an arbitrary number of other

transformation rules for which it provides a context to execute. This context constitutes a collection `collection` that is populated with the resolved value of a `source_expression`, which can be either of *Name- Parent-* or of *Collection Source* type. In case the *Foreach Rule* is declared with the keyword `'collection'`, the enclosed rules `transformation_rules` will be iterated over according to the collection's size. Otherwise, the keyword `'array'` prevents looping and implicitly renders all enclosed *Collection Source Expression's* `offset` to indices accessing `collection` in an array-like manner, instead of shifting the referenced collection's iterator for the specified amount.

Depending on the types of the resolved source and target model elements, a transformation rule may expose different behaviour at its execution. However, if a source and target model element are not "compatible" within the context of a given transformation rule, an exception will be thrown at execution time. Following, Figure 14 gives an overview of the transformation rules supported by the *Marius* transformation engine. Where applicable, the various semantics resulting from different combinations of source and target model elements are stated.

Rule	**LH-Side**	**RH-Side**	**Semantics**
Assign	Primitive Types	Primitive Types	LH side value is assigned to RH side
	RefObject	RefObject	
	RefAssociationLink	RefAssociationLink	
	Collection	Collection	LHS elements are added individually to the RHS collection.
	any other	Collection	The LH side object is added to the RH side collection.
Link	RefObject	RefObject	An instance of RefAssociationLink is connecting LH side and RH side objects. Only instances of RefObject can be linked with each other.
Sibling	RefObject	RefObject	

Invoke	any other	---	A transformation fitting the LH side object's type is initiated.
	Collection	---	Collections are iterated and their elements dealt with individually.
Foreach	any other	---	The LH side object is put into the specified collection.
	Collection	---	The LH side collection's elements are entered in the specified collection individually.

Figure 14 Transformation Rule Semantics

4.4.3. SableCC

SableCC is a compiler generation framework for JAVA. Supplying an EBNF grammar (although some restrictions on naming are imposed) that specifies the intended source language, SableCC generates four different packages, each one standing for a module of the compiler being built. The node package is a JAVA representation of the abstract syntax tree, the lexer and parser packages provide the actual parsing functionality to read source documents and build an in-memory parse tree, and the analysis package contains programming interfaces and default implementations of tree walkers. These walkers can be extended to allow a customized parse tree traversal. During such a traversal, depending on the type of node visited, specific action code is invoked that produces the actual compilation output. This code has to be written manually and is located in the tree walker class, as not to clutter the node package with output generation code.

In case of *Marius'* translation process, the parse tree would represent a transformation definition and the tree walker's action code producing the compilation output would generate the JAVA code for the executable transformations.

The *Marius* transformation engine however, uses a different approach to generate the executable transformations. Only the SableCC generated parser is put to use, leaving the code generation framework's tree walkers aside. Due to the fact that *Marius'* translation target is a JAVA source file adhering to a common structure, a template-based approach is taken. In this case the template-based approach has certain advantages in terms of readability and maintainability over the SableCC variant,

where the tree walker implementation becomes confusingly complex with intermixed JAVA syntax literals.

4.4.4. JET Templates

JET is a generic template engine and a sub-project of EMF. It uses syntax similar to Java Server Pages to define templates that can be used to generate any kind of code. It is important to note that the template engine does not directly generate the target code, but an intermediary JAVA implementation file, which - when executed - produces the final output. Basically, a template contains a target document's "skeleton" and processing instructions. A template is then passed an object as input argument, which is the basis for customization during the template translation phase.

The dynamic aspects in a template can be expressed with two different scripting elements: expressions and scriptlets. A scriplet is a JAVA code fragment enclosed between the symbols <% and %>, which (during template translation) is pasted from the template right into the template implementation class. If the translated scriptlets do not pose valid JAVA statements, the template implementation class cannot be compiled. When finally the template implementation class is invoked, the scriptlet code will be executed and thus affect the generation of the target document.

An expression has to be a complete, valid JAVA statement contained by <%= and %> returning a value. During template translation time, the expression will get enclosed in a piece of code in the template implementation class, which at time of invocation, evaluates the expression and prints the resulting value into the target document. This is assuming that the control flow passes over the code evaluating and printing the expression.

Below is an example of a template file illustrating basic JET concepts. The first line is a JET directive, specifying the name `JETExampleTemplate` for the generated template implementation class and the `example` package containing it. The expression in the second line resolves the value of argument object, which is a reserved identifier referencing the template's input argument. The following three lines use scriptlets replicating the behaviour of line number two. stringBuffer is a reserved identifier used by the template implementation class to store generated code pieces that finally constitute the target document.

```
<%@ jet package="example" class="JETExampleTemplate" %>

Hello, <%=argument.toString()%> !!!

<% stringBuffer.append("Hello, "); %>
<% stringBuffer.append(argument.toString()); %>
```

```
<% stringBuffer.append(" !!!"); %>
```

Assuming the above JET template is passed a java.lang.String with the value "World" as argument, the target document produced after invocation of the template implementation class would look as follows:

```
Hello, World !!!

Hello, World !!!
```

Although JET is embedded in EMF and usually used within Eclipse, a little workaround allows running it outside the IDE, like *Marius* does. To do this, the EMF project page [EMF04] provides a tutorial featuring a JAVA utility plus ANT build file to let JET run as a standalone application.

4.4.5. Java Transformation Classes

The output of the JET engine are the JAVA transformation classes. For every *Marius* transformation definition, one transformation class is being generated. The structure of these classes is similar, as determined by the template they were produced from. If necessary, the generated JAVA source files can be manually fine-tuned. To help find the code that implements a certain transformation rule, comments referring to just that rule are placed accordingly in the transformation classes. Below is an example showing the "`Receive.name =: ActionState.name;`" transformation rule and a code sample of its respective implementation. It is to be assumed, that the objects target, targetBase and source have been previously resolved and refer to `ActionState.name`, `ActionState` and `Receive.name` respectively. Note, that for reasons of clarity the shown code sample omits exception handling.

```
/*  AName2equTrafo      START      "Receive.name =: ActionState.name;"  */
//……
if(target instanceof java.util.Collection) {
        ((java.util.Collection)target).add(source);
}
else {
        if(targetBase == null) {
                //…..
                //…..
        }
        else if(targetBase instanceof javax.jmi.reflect.RefObject) {
                ((RefObject)targetBase).refSetValue("name" , source);
        }
        else {
                //…..
        }
}
//…..
/*  AName2equTrafo      END  */
```

As mentioned previously, besides the transformation logic itself, execution management functionality is implemented in these classes as well, making them a stand-alone, easily deployable, domain specific transformation engine. For instance, this includes keeping traces between source and target model elements, instantiating and invoking new transformations, resolving references to parent transformations and steering the transformation execution control flow.

Chapter 5

Transformation Execution

The transformation from a BPEL document to a UML Activity Diagram displayable in a CASE tool is determined by two distinct mappings. The first mapping is defined between the MyBPEL and the UML 1.4 metamodel. The purpose of the second mapping is to add diagram display information to the UML model resulting from the first transformation. Therefore, a mapping between UML and UML+DI (UML extended by *UML 2.0's Diagram Interchange* package) is defined. Although both mappings could be combined into one, the separation of concerns yields an intermediary UML model uncluttered by positioning information. The result of the second mapping is a more platform specific UML model targeted at CASE tools relying on the *Diagram Interchange* package to capture visualization properties.

Figure 15 illustrates the two-stage transformation process in more detail, as it shows the executable transformations (<<Transformer>>) being produced by the transformation engine according to the BPEL2UML and UML2UML+DI transformation definitions (<<Trafo Definition>>). Then, the transformers for the two mappings generate the target models (<<Model>>) from the source models entering them.

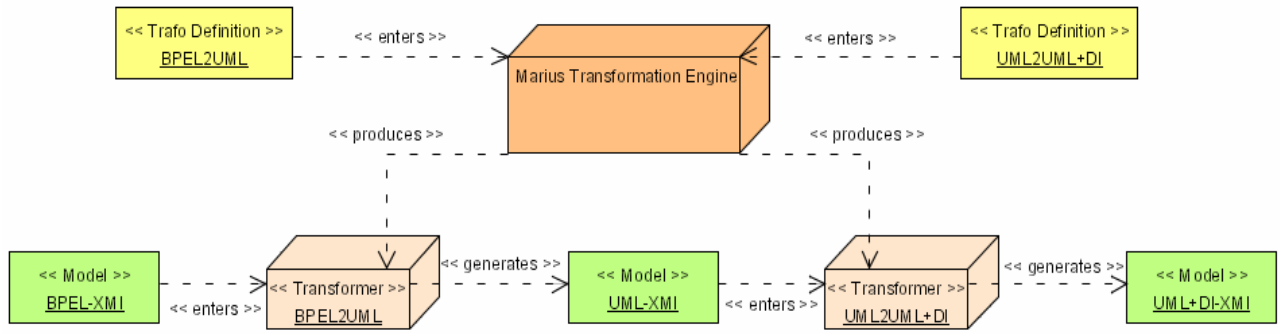

Figure 15 The two-stage Mapping from BPEL to UML+DI

The following two sections will explain the BPEL2UML and the UML2UML+DI mapping in more detail. Every mapping consists of a set of transformations relating source and target model elements. Besides a textual description of each transformation clarifying the semantic relationships between model elements, diagrams utilizing a straightforward UML-like notation are depicting the mappings in a visual way. The actual transformation definition files for both mappings can be found in the appendix.

5.1. The MyBPEL2UML Mapping

Figure 16 shows an informal UML diagram describing the transformation definitions employed in the MyBPEL2UML mapping. The classes on left-hand side represent the MyBPEL metamodel, and the right-hand side classes are a subset of the UML metamodel necessary to model an Activity Diagram. (Note, that this “transformation diagram” does not show the source and target metamodels in full detail.)

In between the source and target domain lie the transformations, which relate the various model elements. Associations targeted at the left-hand side link transformations with the model elements declared as `Trafosource` in the respective transformation definition. Analogous to that, the associations targeted at the right-hand side link to `Trafotarget`. Furthermore, there are associations linking to model elements declared in the `variables section`, which are instantiated through transformation execution. Associations between transformations express a parent/child relationship, with a parent being the caller of a child.

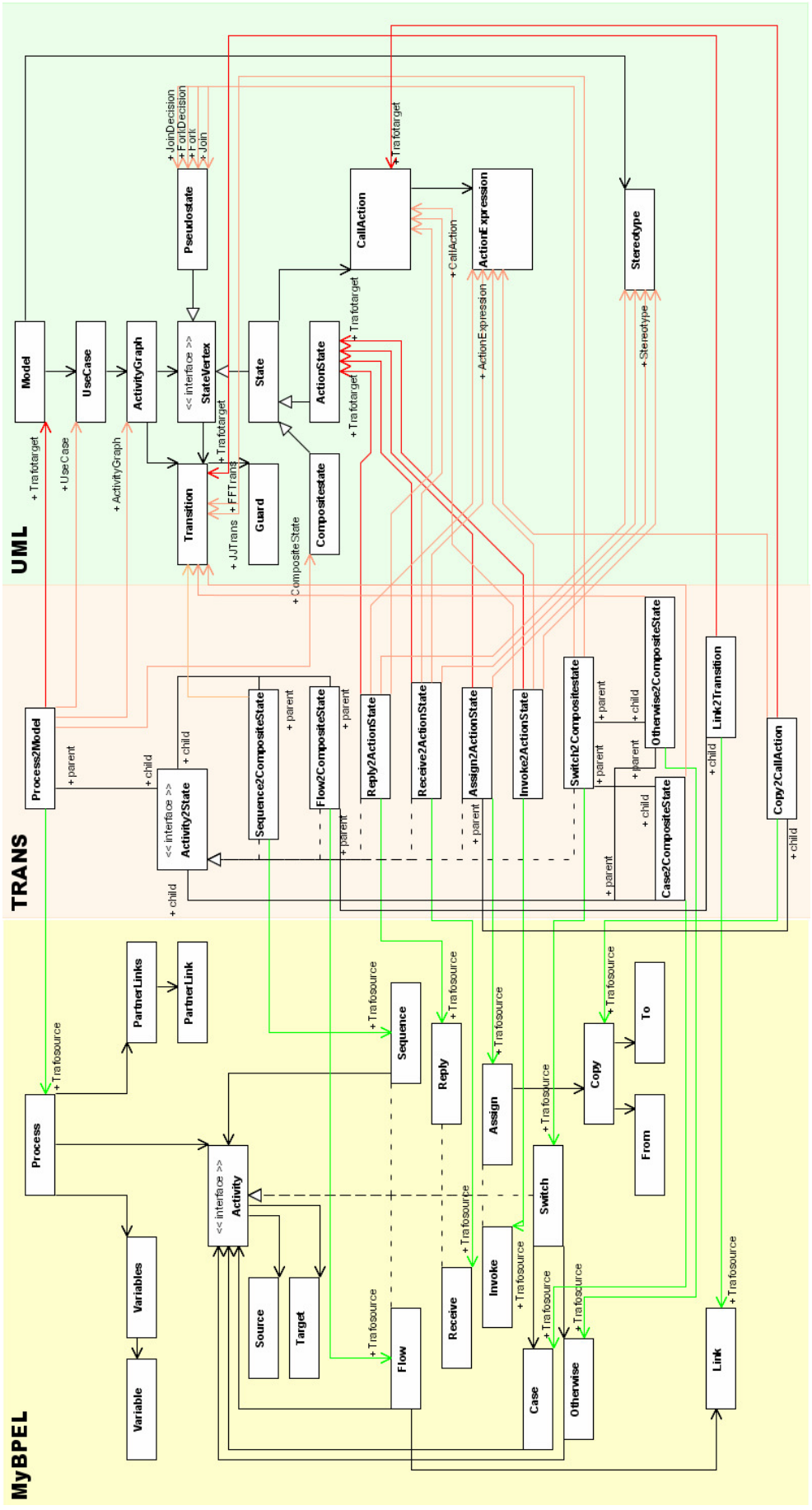

Figure 16 The MyBPEL2UML Mapping

MyBPEL2UmlPackage

The very top-most transformation is relating the source and target package, namely the MyBPEL and the UmlPackage packages. The describing picture – Figure 17 - below relates to the larger transformation diagram above such that the packages contain all the respective left-hand side and right-hand side classes, and the MyBPEL2UmlPackage is the root transformation initiating the mapping process.

Figure 17 The MyBPEL2UmlPackage Transformation

Process2Model

The root top-most element in a BPEL definition is the Process element, which relates to the Model element within UML's Model Management package. Furthermore, a UseCase element will be generated within the Model's namespace. In the context of this UseCase, an ActivityGraph element having a CompositeState element will be instantiated. Simply put, Process maps to a UML Model containing a UseCase associated with an empty ActivityGraph.

Activity2State

As marked in the diagram, this transformation has solely interface character and does not have an implementation. It is an abstract transformation linking the Activity and the State model elements, thus representing all concrete transformations relating sub-types of these model elements.

Sequence2CompositeState

The Sequence2CompositeState transformation maps all activities (sub-types of Activity) contained in a Sequence to their respective counterparts in the UML domain by initiating the appropriate sub-transformations. This results in a number of target model elements sub-type to StateVertex. Transition elements are instantiated which then consecutively connect these resulting UML model elements.

Flow2CompositeState

Similar to the Sequence2CompositeState transformation, Flow2CompositeState maps all activities contained in Flow to their UML counterparts. Furthermore, Links contained in Flow are mapped to Transition elements.

Reply2ActionState

A Reply activity relates to an ActionState having a Stereotype and containing a CallAction with an ActionExpression. If the Reply activity is part of a Flow and contains a Source or Target element, the Transition elements created by parent transformations are queried and the resulting ActionState is linked with a matching Transition element accordingly.

Receive2ActionState

The Receive2ActionState behaves analogous to the Reply2ActionState transformation.

Invoke2ActionState

The Invoke2ActionState behaves analogous to the Reply2ActionState transformation.

Assign2ActionState

The Assign2ActionState transformation is similar to Reply-, Invoke-, and Receive2ActionState, but it does not produce a CallAction and an ActionExpression, but instead initiates a sub-transformation mapping a contained Copy element.

Switch2CompositeState

The Switch2CompositeState transformation maps to a skeleton of transitions and pseudostates to model the "switch-case-otherwise" characteristics. The activities in the contained Case and Otherwise elements are inserted into that structure. Thus, a "Junction-kind" Pseudostate followed by a "Fork-kind" Pseudostate is instantiated, to join all incoming transitions before fanning out to the various optional paths. All contained Case elements and the Otherwise element are mapped by sub-transformations. Finally, all optional paths are joined in a "Join-kind" Pseudostate, which is followed by a "Junction-kind" Pseudostate that allows fanning out to subsequent states.

Case2CompositeState

A Case2CompositeState transformation maps the contained activity (sub-type of Activity) to its appropriate counterpart by calling the suitable sub-transformation. Furthermore, a Transition annotated with a Guard containing a BooleanExpression models the incoming link. Likewise, a Transition represents the outgoing link. The resulting structure of transitions and states is inserted into the "skeleton" produced by the Switch2CompositeState transformation.

Otherwise2CompositeState

The Otherwise2CompositeState transformation is equal to Case2CompositeState, but instead of an expression queried from the source model, BooleanExpression is assigned the string literal "`otherwise`".

Link2Transition

A Link2Transition transformation maps a Link to a Transition model element.

Copy2CallAction

A Copy2CallAction maps a Copy model element to a CallAction model element containing an instance of ActionExpression.

5.2. The UML2UML+DI Mapping

Analogous to the description in the previous section, the "transformation diagram" in Figure 18 illustrates the UML2UML+DI mapping. Left-hand side and right-hand side are both UML metamodels, although the right-hand side explicitly represents a subset of the classes contained in the Diagram Interchange package.

The UML2UML+DI mapping refines a UML ActivityGraph and instantiates the appropriate classes within the Diagram Interchange package for display purposes. Therefore, a conventional UML source model will be extended by diagrammatic information and result in a true UML+DI target model.

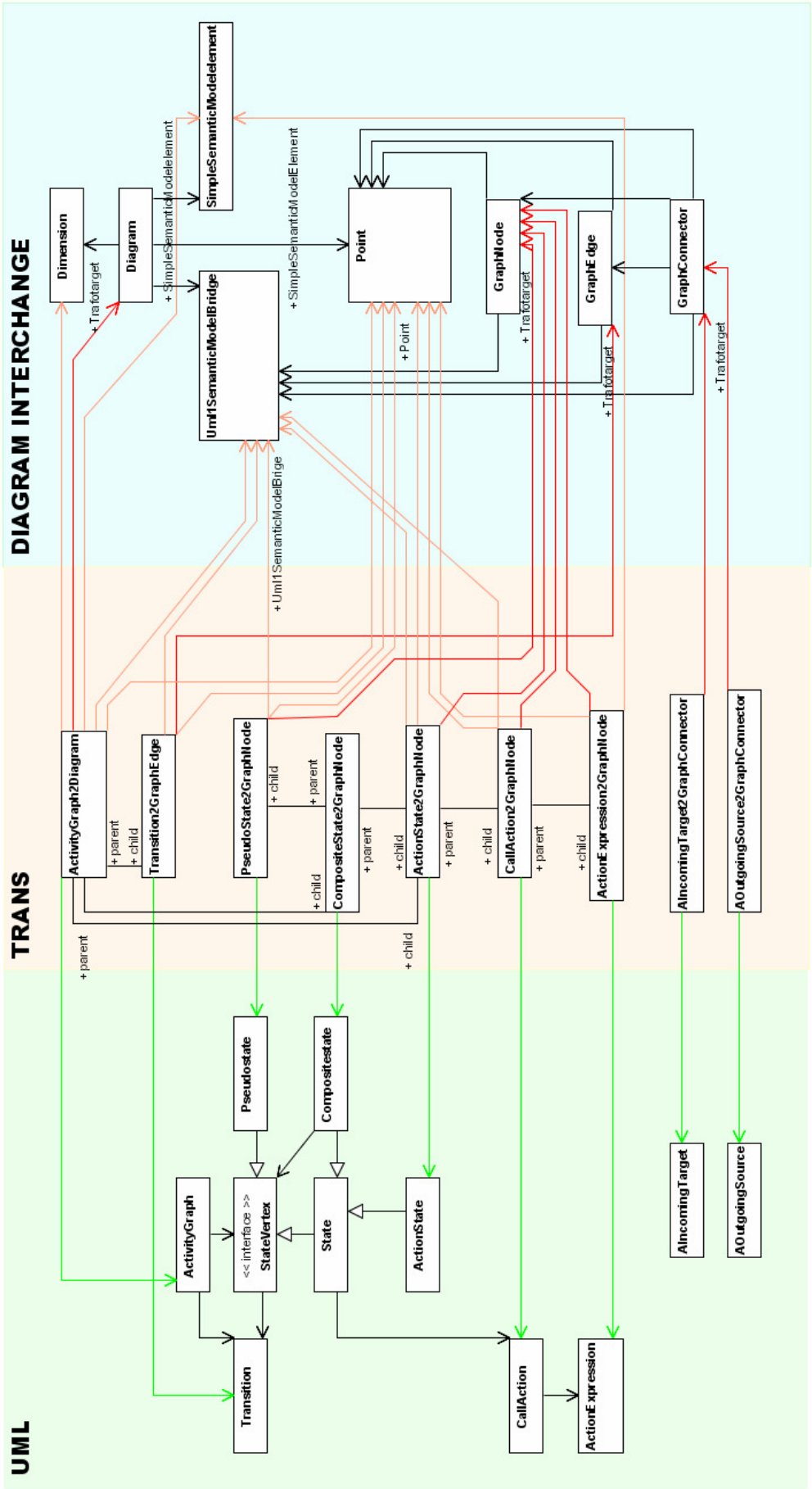

Figure 18 The UML2UML+DI Mapping

Uml2UmlDI

The UML2UML+DI mapping has a refining character, which means that it only produces diagrammatic information which it adds to the target model, but it does not clone the original source model. Hence, source and target model ought to be the same to achieve a meaningful refinement. As depicted below in Figure 19, `Trafosource` and `Trafotarget` are both referring to the same UmlPackage model element.

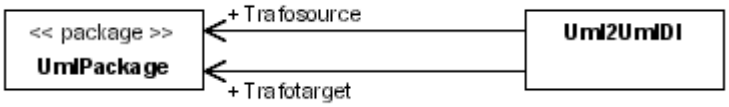

Figure 19 The Uml2UmlDI Transformation

As the root transformation initiating the mapping process, Uml2UmlDI launches the sub-transformations for UML's AIncomingTarget and the AOutgoingSource associations, as well as for the ActivityGraph model element.

ActivityGraph2Diagram

The ActivityGraph2Diagram transformation maps an ActivityGraph to a Diagram model element. Furthermore, a SimpleSemanticModelElement and a Uml1SemanticModelBridge are instantiated to semantically tie the established ActivityGraph to the newly generated Diagram. Point and Dimension structures are used to set properties of Diagram. Activity2Diagram initiates sub-transformations for all contained Transition model elements and for its top-most State.

Transition2GraphEdge

The Transition2GraphEdge transformation maps a Transition to a GraphEdge model element and semantically ties them via a Uml1SemanticModelBridge instance. A Point structure is used to set a property of GraphEdge.

PseudoState2GraphNode

The Pseudostate2GraphNode transformation maps a Pseudostate to a GraphNode model element and semantically ties them via a Uml1SemanticModelBridge instance. A Point structure is used to set a property of GraphNode.

CompositeState2GraphNode

A CompositeState2GraphNode transformation initiates sub-transformations for all contained State model elements.

ActionState2GraphNode

The ActionState2GraphNode transformation maps an ActionState to a GraphNode model element and semantically ties them via a Uml1SemanticModelBridge instance. A Point structure is used to set a property of GraphNode. ActionState2GraphNode initiates a sub-transformation to map a contained CallAction.

CallAction2GraphNode

Invoked by ActionState2GraphNode, the CallAction2GraphNode transformation maps a CallAction belonging to an ActionState to a GraphNode model element and semantically ties them via a Uml1SemanticModelBridge instance. A Point structure is used to set a property of GraphNode. CallAction2GraphNode initiates a sub-transformation to map a contained ActionExpression.

ActionExpression2GraphNode

Invoked by CallAction2GraphNode, the ActionExpression2GraphNode transformation maps an ActionExpression to a GraphNode model element, which is assigned its appropriate semantics by a SimpleSemanticModelElement instance. A Point structure is used to set a property of GraphNode.

AIncomingTarget2GraphConnector

AIncomingTarget2Graphconnector maps all instances of the AIncomingTarget association to GraphConnector model elements. Each GraphConnector is associated with those GraphNode and GraphEdge model elements, who are “siblings” of the association ends of an AIncomingTarget instance, referring to Transition and StateVertex model elements respectively. Following, Figure 20 shows a simplified subset of the UML and the Diagram Interchange metamodel to illustrate the described relationship. Furthermore, a Point structure is used to set a property of GraphConnector.

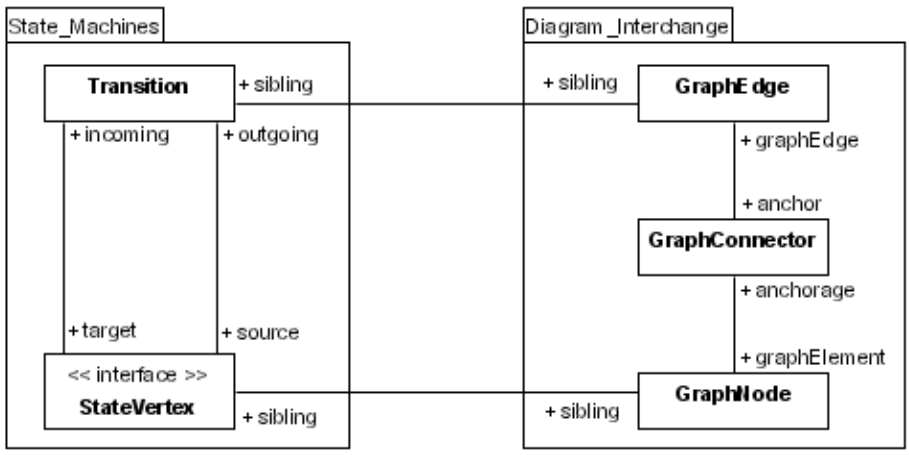

Figure 20 Relationship between subsets of 'State Machines' and 'Diagram Interchange'

AOutgoingSource2GraphConnector

The AOutgoingSource2GraphConnector transformation is equivalent to the above-mentioned AIncomingTarget2GraphConnector transformation, save for the fact that it maps AOutgoingSource associations to GraphConnector model elements.

5.3. Executing a Transformation

This section is a step-by-step guide illustrating the usage of Marius by transforming the "MyEcho" BPEL file into an activity diagram. Two more complex transformation examples can be found in the appendix.

However, first of all a parser for the Marius transformation language has to be built. In this case SableCC 2.18.2 is used, which generates a class structure from the Marius grammar file. Then, the JET template implementation class is built with the help of a JAVA utility and an ANT file provided by [EMF04]. Finally, a simple JAVA program for instance can parse all transformation definition files and feed the resulting parse trees one by one into the template engine, which generates the executable transformations in the form of JAVA source files. Figure 21 informally describes this build process, which could be bundled up in an ANT file for instance.

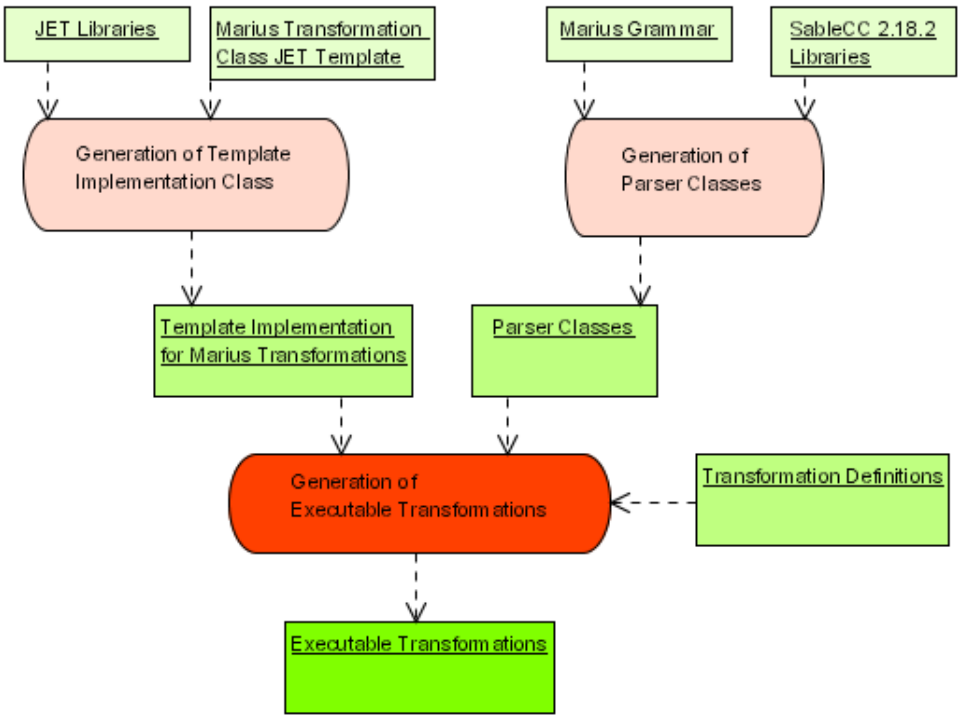

Figure 21 Build process for Marius Executable Transformations

After compilation of these transformation classes, a simple application utilizing functionality provided by Marius' transformation framework initiates the transformation execution. Assuming the below listed "myecho.bpel" file is subject to the previously explained MyBPEL2UML and UML2UML+DI mapping, an XMI-encoded UML model including diagram information will be the result.

```
<process name="echoString"
        targetNamespace="urn:echo:echoService"
       xmlns:tns="urn:echo:echoService"
       xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <variables>
    <variable name="request" messageType="tns:StringMessageType"/>
  </variables>

  <partnerLinks>
    <partnerLink name="caller" partnerLinkType="tns:echoSLT"/>
  </partnerLinks>

  <sequence name="EchoSequence">
    <receive partnerLink="caller" portType="tns:echoPT"
            operation="echo" variable="request"
            createInstance="yes" name="EchoReceive"/>
    <reply partnerLink="caller" portType="tns:echoPT"
          operation="echo" variable="request" name="EchoReply"/>
  </sequence>

</process>
```

Loaded into the Poseidon modelling tool, the UML model's Activity Diagram will be rendered as follows:

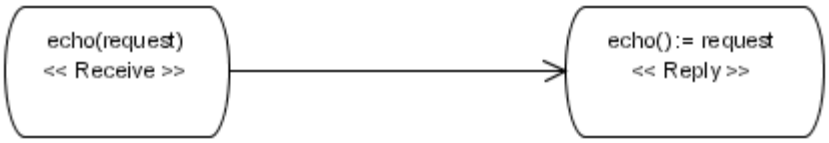

Figure 22 The 'MyEcho' BPEL Transformed into an Activity Diagram

The Activity Diagram in Figure 22 shows two states, each one stereotyped as the kind of activity it represents. First, an echo operation is called with a request parameter, which is then simply returned. Note, that Poseidon 2.5.1 CE neither displays names given to states nor is it able to handle "swimlanes", which therefore do not appear in the diagram.

Chapter 6

Related Work

This chapter tries to take a glance at some other works related to this diploma thesis, either in its applied methods or technologies. Most importantly, this includes the subject of generic model transformation. Besides a short overview of interesting topics in this respect, a focus is put on supporting software tools and frameworks. Besides MDA influenced issues, recent developments and software systems concerning BPEL are looked into.

6.1. QVT Responses

With Query / Views / Transformation, the OMG started to create a standard for a generic model transformation language. The initial QVT-RFP received eight submissions. [GGKH] provides a good overview summarizing and comparing these proposals' characteristics.

6.2. Model transformation tools & frameworks

Existing MDA tools, both commercial and open source ones, are aimed at building applications on the notion of MDD, *Model Driven Development*. Their support for model-transformation usually comes in the form of pre-defined mappings to a certain target environment. For example, on the open-source sector there is AndroMDA targeting the J2EE environment and OpenMDX [OPEN] also supporting DOTNET.

In an effort to establish languages and tools for viable, generic model transformation a number of tools have evolved. A short overview of some transformation engines and supporting utilities is given below.

Model Transformation Framework (MTF) [MTF04] by IBM is a framework integrating with Eclipse's EMF and offers model transformation functionality for EMF models.

ATL [ATL04] is a transformation engine developed by the INRIA Atlas team and is based on the QVT-RFP. The *ATL* engine shall provide the core functionality within Eclipse's *Generative Model Transformer (GMT)* [GMT04] project. The goal of *GMT* is to provide an environment for the generation, execution and debugging of model transformations.

MTL [MTL04] developed by the INRIA Triskell team is another QVT-like transformation language. The MTL model compiler can be integrated with EMF.

The *Kent Modelling Framework (KMF)* [KENT] is enabling the generation of model transformation tools from language definitions. Included are JAVA libraries for the dynamic evaluation of OCL constraints.

UMT [UMT04] is a UML transformation tool distributed with a number of generators for target domains like WSDL, XML Schema, JAVA, and others. Custom transformations can be plugged in.

The *Bidirectional Object-oriented Transformation Language (BOTL)* [BOTL] integrates with ArgoUML and allows to specify transformations in a graphical notation.

Uml2Svg [USVG] is an XSLT based project aimed at generating an SVG file from a UML model annotated with *Diagram Interchange* information. An online version as well as a downloadable one is available.

Working on a higher abstraction level than the above-mentioned "object-to-object" techniques is the concept of Generic Model Management [BERN03]. The artefacts involved are not mere model elements and transformation rules, but entire models and mappings between models. Model Management offers high-level operators to manipulate these. *Rondo* [MRB03] is a tool implementing the notion of Model Management.

6.3. From UML to BPEL

The inverse direction of the BPEL2UML mapping subject to this work is dealt with in [GAR03A]. The proposed UML2BPEL mapping is founded on the *UML Profile for Automated Business Processes* [IBM03], which is also the base for the mapping applied in this work. A proof of concept demonstrator, the "UML 2 BPEL Mapping Demo", is installable in Eclipse and bundled in the *Emerging Technologies Toolkit* [ETTK04].

6.3. BPELJ: BPEL for Java

A joint whitepaper [BPELJ] by BEA and IBM specifies BPELJ, which proposes mixing traditional BPEL with JAVA snippets. One of the key motivations was to enable interaction with other than XML-based interfaces. Even though any kind of object can somehow be wrapped up in a web service, this poses a considerable marshalling/unmarshalling overhead. Business functions, in this case especially in the form of JAVA APIs, should effortlessly integrate in the business process logic implemented in BPEL.

BPELJ has sparked a lot of controversy, mainly as it waivers language-neutrality and allows the implementation of business logic at a higher-level layer, which is supposed to define the business process only. BEA however backs BPELJ strongly and implementations of the specification are to be expected.

6.4. BPEL Process Engines

A Business Process Engine is service-oriented runtime environment for the execution of business processes. As BPEL has become a well-established standard for describing business processes from a technical point of view, various commercial and open tools have evolved supporting it. Commercial applications like *IBM's WBISF* and *Oracle's BPEL Process Manager*, typically provide an execution environment, an editor for designing and deploying business processes and means for debugging and load testing. *ActiveBPEL* [ACTI05] is an open-source project developing a BPEL engine, whose technology finds application in the products of *Active Endpoints*.

Chapter 7

Future Work

Existing MDA tools and frameworks have proven the capabilities of a model-centric approach in software development. However, turning models into working applications is still not a trivial process and there is plenty of room for improvement. More so, it is still to be seen how MDA can influence different software engineering scenarios and lifecycle phases within the development process. How, for instance, does MDA manifest in agile methods or how can model-driven testing be carried out?

Key aspects needed to fully facilitate the MDA paradigm are standards for generic model transformations. The approaches made, like the QVT responses, are still subject to ongoing research and existing applications have to prove their applicability.

This chapter discusses a few aspects of the above mentioned issues and tries to estimate their potential impact on the evolution of the model centric paradigm. Because MDA is not a single standard, but rather a family thereof, a brief outlook on what can be expected of upcoming MOF and UML versions is given.

Concerning the practical aspects of this work, the transformation from BPEL to UML, a number of possible extensions to the *Marius* tool are pointed out. The future development of BPEL and UML and its influence on the BPEL2UML mapping are also looked into.

7.1. MDA's impact on the software development process

The rise of MDA is significantly changing the way in which software is being engineered. Building systems from models requires new tools and techniques to support code generation, model transformation, model-based testing and the like. However, apart from the technical aspects, the whole software development process itself becomes affected.

Due to the fact that MDA code generators are capable of creating a considerable amount of code from models, the traditional "coding-phase" becomes significantly shortened or even obsolete. Be it a heavy-weight development process or a flexible, agile one: On the background of model-driven development and architecture, software process models shall adapt to these new realities.

Thoughts about how to combine two seemingly contradictory concepts like modelling and agility into *"Agile MDA"* are expressed in [AGMO05] and [MELL05].

7.2. Improvements to the Marius Transformation Tool

At present, the Marius Transformation Tools is an experimental work in progress and is not meant to be a full-scale model transformer. Therefore, in terms of developing the tools' architecture and its model transformation capabilities, plenty of room for improvement is left.

One immediate improvement to make full use of the MOF metamodel would be to include support for all of JMI's not yet attended Reflective Interfaces for Data Types (e.g.: `RefEnum`). Other, more far-reaching enhancements could deal with a more sophisticated mechanism to control transformation execution. Specifically traceability between transformations and model elements, as well as exception handling come to mind. Changes concerning the transformation definition language, of which some are proposed in section 6.5, will have to be reflected in *Marius'* transformation engine.

Another goal is to form a metamodel describing the transformations used in Marius in a MOF compatible way. The notion "Transformations are Models" is properly applied in that way and the implementation of a transformation engine is put on a uniform base, as all artefacts involved are rooted in MOF. In the light of that, the adoption of OCL as a query language would be a practical enhancement. Models could thus be queried and checked for consistency in a standardized, MOF 2.0

compatible manner. Generally, *Marius* ought to migrate towards that standard once it is finalized. That is, assuming a standardized JAVA mapping (JMI) for MOF 2.0 is available. Another interesting path the development of *Marius* could take is the integration into the Eclipse EMF environment, whose metamodel "Ecore" is arguably closer to MOF 2.0 than MOF 1.x is.

7.3. Improvements to the MyBPEL2UML mapping

The MyBPEL2UML mapping used throughout this work is not complete, as certain BPEL aspects are neglected for reasons of simplicity. Therefore, assuming the MyBPEL metamodel is extended in such a way, a concise mapping would require the omitted constructs to be included in the mapping. Furthermore desirable is to enable the transformation from UML back to BPEL. This would require an inverse mapping (UML2MyBPEL) preserving semantic information throughout the whole round-trip.

The mapping from UML to UML+DI, produces a UML model containing model elements relevant for diagram display. However, the mapping does not encompass automatic diagram element routing or layouting. Hence, attributes specifying location or appearance of diagram elements are set to default values, unless explicitly hard-coded in the transformation definition. If a resulting model is imported in Poseidon for instance, the diagram elements have to be arranged manually. To overcome this problem a separate routing step would have to take place. Possibly a program manipulating the model instance in the repository directly, or an XMI-based XSLT transformation would suffice. However, the layout and auto-routing is a bit out of model transformation's scope and should rather concern visual tools importing models.

A considerable drawback of using Poseidon for business modelling lies within the inability to model swimlanes (partitions) in an Activity Diagram. Gentleware states that the realization of swimlanes is planned. It is to be seen whether the migration to UML 2.0 planned for Q2 2005 will see them implemented. However, either the use of tagged values or stereotypes to "simulate" swimlanes can be an acceptable workaround to compensate for this weakness. An exported model tagged this way could undergo a simple transformation to generate the missing model elements.

7.4. Upcoming standards UML 2.0 & MOF 2.0

UML 2.0 and MOF 2.0 have evolved along-side each other and share a common core. The main advantage of these new specifications is that their metamodels have been thoroughly reworked.

Drawbacks in UML 1.x were for instance overlapping concepts (state diagram and activity diagram) and unclear semantics (composition vs. aggregation) having left room for varying interpretations. Due to the overhaul in the new version, UML models become more platform-independent and unambiguous. Activity Diagrams for example have undergone a considerable evolutionary step, finally emancipating themselves from State Diagrams. Furthermore, the new Activity Diagrams semantics' is very similar to those of Petri Nets. This should make up for the somewhat limited Business Process Modelling capabilities of the previous UML versions. Another novelty in UML 2.0 is a metamodel for OCL, as is the Diagram Interchange package already used as an extension to UML 1.4 in this work.

Overall, UML 2.0 greatly supports MDA, as it leads to an easier construction of executable models and forms a solid foundation for model transformation by seamlessly integrating with its sister specification MOF 2.0. The QVT process currently underway to create a transformation language specification is also based on MOF 2.0. Meaning, that once a final QVT version is adopted and a mapping from MOF 2.0 to a programming language, such as JMI and EMF's Java mapping exists, standardized model transformation - MDA's missing link - can be implemented.

As mentioned earlier, UML 2.0 offers many new concepts for behavioural modelling assisting the notion of Business Process Modelling. This is especially true for Activity Diagrams, which (among other innovations) now support Exception Handling and special nodes to model iterative behaviour. In the context of the MyBPEL2UML mapping this could manifest in more concise modelling approaches to various BPEL concepts.

7.5. Improvements to the Marius Transformation Language

To make the *Marius'* language more applicable for defining model transformations a number of new ideas and grammar changes have to be introduced. Keeping the second-system effect [BROO75] in mind, one can advocate to redesign and rethink the language's core concepts to address challenges in the current system. Simply adding more features and language constructs will only bloat the grammar and not necessarily improve the language. As mentioned above, utilizing OCL as a query

language serves as a solid foundation for realizing conditional rules and retrieving information from models. To improve the handling of strings and arithmetic expressions and primitive types in general, it would be convenient to offer more utility functions, as abstractions of underlying JAVA methods. Another possibility would be to allow stating JAVA code directly in the transformation definition. This is however not recommendable, as it would doubtlessly clutter the code and tempt to use JAVA not just for primitive type handling, but also for expressing the transformation logic itself. Overall, this would break the concept of an abstract model transformation language in the first place.

Finding a visual syntax to describe transformation definitions is not an immediate necessity. Although such a syntax is proposed in [QVTM04], it is to see what the upcoming QVT standard will bring. However, with UML 2.0 looming on the horizon, it could be interesting to either find a metamodel extension or a profile for model transformations.

A final goal would be to trim Marius to adhere to the QVT standard. This would mean to fulfil all of the QVT-RFPs mandatory requirements. *Marius* has a pure transformational character and supports neither the creation of views nor the relational checking of models for consistency. Therefore, a necessary improvement is to bolster the expressiveness of *Marius'* transformation definition language in this respect. Furthermore, the abstract syntax for transformation definitions would have to be represented as a MOF 2.0 metamodel.

7.6. Domain Specific Readers and Writers

Apart from the immediate development of model transformation engines, there is a need for the realization of supporting utilities. In the case of Marius, a way to easily generate domain specific readers and writers would be a great enhancement. The BPEL reader used in this work is based on an XSLT style sheet entirely hand-coded. However, in case of XML input data it should be possible to at least partly automate the generation of that style sheet. A requirement therefore would be a specification of the input data in question, typically in the form of an XML schema definition. Then, a tool mapping XML schema to XMI automatically generates a metamodel definition. A guide on how to implement such a utility creating a MOF compatible metamodel from XML Schema can be found in [XMI03]. EMF for instance offers this kind of functionality for its "Ecore" metamodel. Following the metamodel generation, another tool produces an XSLT style sheet that is finally able to parse input documents and instantiate models accordingly. The tools carrying out the mappings could be XSLT transformations themselves. Alternatively, to an XSLT

based parsing approach, the implementation could be realized with a higher-level XML handling API [JAXB] as well.

An approach to create XML data from a model instance in a repository could start out with a mapping between the metamodel in question – let's assume BPEL - and an XML metamodel. Then, a BPEL model instance can be transformed into an XML model instance. Finally, a program could traverse the XML model and serialize it into its textual form, yielding a BPEL document.

To serialize an arbitrary model contained in the repository into its M0-level representation is however not trivial. As already mentioned, the MOF metamodelling hierarchy does not define the "instanceOf"-relationship between layers M1 and M0. Domain specific readers and writers as described above implicitly carry the semantics of this relationship in their behaviour. This means a way to express the characteristics of the "instanceOf"-relationship separating meta-levels has to be found. [IKKB04] proposes a way to unify the metamodelling layers into a coherent modelling space and thus enable transformation across these boundaries.

Chapter 8

Appendix

8.1. MyBPEL Metamodel

The following class diagram in Figure 23 represents the MyBPEL metamodel, which models only a subset of all of BPEL's aspects. Classifier names that would be JAVA keywords and therefore cause compilation problems were added 'KEY' to their name. The model was created in Poseidon 2.5.1. CE.

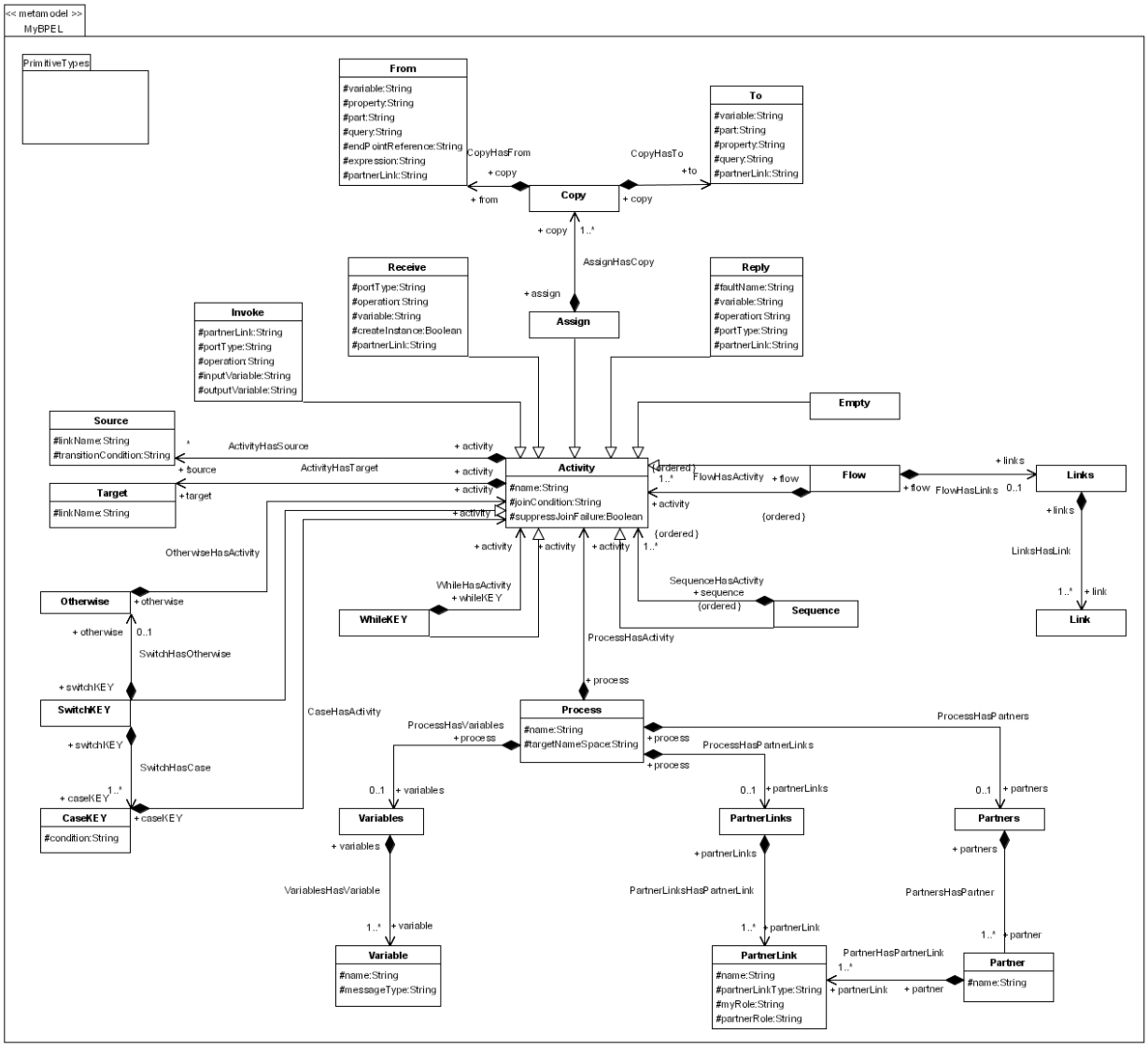

Figure 23 The MyBPEL Metamodel

8.2. The MyBPEL2UML Transformation Definitions

The following sections contain the transformation definitions that make up the MyBPEL2UML mapping.

8.2.1. Assign2ActionState

```
Assign2ActionState

Trafosource
Assign Assign

Trafotarget
ActionState ActionState Activity_Graphs

meta
Entry
```

```
Exit

enums

structs

variables
Stereotype Stereotype Core;

mapping

ActionState =: Entry;
ActionState =: Exit;

Assign.name =: ActionState.name;

Assign.copy -> *;

$"Assign" =: Stereotype.name;
$"ActionState" =: Stereotype.baseClass;

ActionState <> Stereotype A_stereotype_extendedElement Core;

Process2Model:Model <> Stereotype A_namespace_ownedElement Core;
Process2Model:CompositeState <> ActionState A_container_subvertex
State_Machines;

foreach collection Flow2CompositeState:Trans transvec {
      variables
      Transition RefObject;

      mapping
      transvec!0 =: Transition;

      foreach collection Assign.source sourcevec {
            variables
            Source RefObject;

            mapping
            sourcevec!0 =: Source;
            #Transition.name == Source.linkName#   Transition <>
            ActionState A_outgoing_source State_Machines;
      };
      foreach collection Assign.target targetvec {
            variables
            Target RefObject;

            mapping
            targetvec!0 =: Target;
            #Transition.name == Target.linkName#   Transition <>
            ActionState A_incoming_target State_Machines;
      };
};
```

8.2.2. Case2CompositeState

```
Case2CompositeState

Trafosource
Case CaseKey

Trafotarget
---

meta
Entry
```

```
Exit
Sub

enums

structs

variables

mapping

Case.activity -> * =: Sub;

foreach array Sub subVec {
   variables

   mapping
   subVec!0.Entry =: Entry;
   subVec!0.Exit =: Exit;
};

foreach collection Entry entryVec {
   variables
   Transition Transition State_Machines;
   Guard Guard State_Machines;
   BooleanExpression BooleanExpression Data_Types;

   mapping
   Process2Model:ActivityGraph <> Transition
   A_stateMachine_transitions State_Machines;

   Case.condition =: BooleanExpression.body;
   BooleanExpression =: Guard.expression;
   Guard =: Transition.guard;

   entryVec!0 <> Transition A_incoming_target State_Machines;
   Switch2CompositeState:JoinDecision <> Transition
   A_outgoing_source State_Machines;
};

foreach collection Exit exitVec {
   variables
   Transition Transition State_Machines;

   mapping
   Process2Model:ActivityGraph <> Transition
   A_stateMachine_transitions State_Machines;

   exitVec!0 <> Transition A_outgoing_source State_Machines;
   Switch2CompositeState:ForkDecision <> Transition
   A_incoming_target State_Machines;

};
```

8.2.3. Copy2CallAction

```
Copy2CallAction

Trafosource
Copy Copy

Trafotarget
CallAction CallAction Common_Behavior

meta
```

```
enums

structs

variables
ActionExpression ActionExpression Data_Types;
To metamodel1.To;
From metamodel1.From;

mapping

Copy.to =: To;
Copy.from =: From;

(String)(To.variable + "/" + To.part + " := " + From.variable +
From.expression + "/" + From.part) =: ActionExpression.body;
ActionExpression =: CallAction.script;

Assign2ActionState:ActionState    <>    CallAction    A_state_entry

State_Machines;
```

8.2.4. Flow2CompositeState

```
Flow2CompositeState

Trafosource
Flow Flow

Trafotarget
---

meta
Entry
Exit
Trans
Subs

enums

structs

variables
Links RefObject;

mapping

Flow.links =: Links;
Links.link -> *.Transition =: Trans;

Flow.activity -> * =: Subs;

foreach collection Subs subsvec {
      variables
      mapping
      subsvec!0.Entry =: Entry;
      subsvec!0.Exit =: Exit;

};
```

8.2.5. Invoke2ActionState

```
Invoke2ActionState

Trafosource
Invoke Invoke

Trafotarget
ActionState ActionState Activity_Graphs

meta
Entry
Exit

enums

structs

variables
CallAction CallAction Common_Behavior;
ActionExpression ActionExpression Data_Types;
Stereotype Stereotype Core;

mapping

ActionState =: Entry;
ActionState =: Exit;
ActionState <> CallAction A_state_entry State_Machines ;

Invoke.name =: ActionState.name;
(String)(Invoke.outputVariable + " := " + Invoke.operation  + "(" +
Invoke.inputVariable + ")") =: ActionExpression.body;
ActionExpression =: CallAction.script;

$"Invoke" =: Stereotype.name;
$"ActionState" =: Stereotype.baseClass;

ActionState <> Stereotype A_stereotype_extendedElement Core;
Process2Model:Model <> Stereotype A_namespace_ownedElement Core;

Process2Model:CompositeState <> ActionState A_container_subvertex
State_Machines;

foreach collection Flow2CompositeState:Trans transvec {
      variables
      Transition RefObject;

      mapping
        transvec!0 =: Transition;

        foreach collection Invoke.source sourcevec {
                variables
                Source RefObject;

                mapping
                sourcevec!0 =: Source;
               #Transition.name == Source.linkName#   Transition <>
               ActionState A_outgoing_source State_Machines;
        };
        foreach collection Invoke.target targetvec {
                variables
                Target RefObject;

                mapping
                targetvec!0 =: Target;
                #Transition.name == Target.linkName#   Transition <>
                ActionState A_incoming_target State_Machines;
        };
};
```

8.2.6. Link2Tansition

```
Link2Transition

Trafosource
Link Link

Trafotarget
Transition Transition State_Machines

meta

enums

structs

variables

mapping

Link.name =: Transition.name;
Process2Model:ActivityGraph <> Transition A_stateMachine_transitions
State_Machines;
```

8.2.7. MyBpelPackage2UmlPackage

```
MyBpelPackage2UmlPackage

Trafosource
MyBpelPackage MyBpelPackage

Trafotarget
UmlPackage UmlPackage

meta

enums

structs

variables

mapping

MyBpelPackage.Process -> *;
```

8.2.8. Otherwise2CompositeState

```
Otherwise2CompositeState

Trafosource
Otherwise Otherwise

Trafotarget
---

meta
Entry
Exit
Sub

enums
```

```
structs

variables

mapping

Otherwise.activity -> * =: Sub;

foreach array Sub subVec {
   variables
   mapping
   subVec!0.Entry =: Entry;
   subVec!0.Exit =: Exit;
};

foreach collection Entry entryVec {
   variables
   Transition Transition State_Machines;
   Guard Guard State_Machines;
   BooleanExpression BooleanExpression Data_Types;

   mapping
   Process2Model:ActivityGraph <> Transition
   A_stateMachine_transitions State_Machines;

   $"otherwise" =: BooleanExpression.body;
   BooleanExpression =: Guard.expression;
   Guard =: Transition.guard;

   entryVec!0 <> Transition A_incoming_target State_Machines;
   Switch2CompositeState:JoinDecision <> Transition
   A_outgoing_source State_Machines;
};

foreach collection Exit exitVec {
   variables
   Transition Transition State_Machines;

   mapping

   Process2Model:ActivityGraph <> Transition
   A_stateMachine_transitions State_Machines;

   exitVec!0 <> Transition A_outgoing_source State_Machines;
   Switch2CompositeState:ForkDecision <> Transition
   A_incoming_target State_Machines;

};
```

8.2.9. Process2Model

```
Process2Model

Trafosource
Process Process

Trafotarget
Model Model Model_Management

meta

enums

structs
```

```
variables
UseCase  UseCase     Use_Cases;
ActivityGraph ActivityGraph     Activity_Graphs;
CompositeState CompositeState State_Machines;

mapping

Process.name =: Model.name;
Model <> UseCase A_namespace_ownedElement Core;
Process.name =: UseCase.name;
UseCase <> ActivityGraph A_behavior_context State_Machines;

CompositeState <> ActivityGraph A_top_stateMachine State_Machines;

Process.activity -> *;
```

8.2.10. Receive2ActionState

```
Receive2ActionState

Trafosource
Receive Receive

Trafotarget
ActionState ActionState Activity_Graphs

meta
Entry
Exit

enums

structs

variables
CallAction CallAction Common_Behavior;
ActionExpression ActionExpression Data_Types;
Stereotype Stereotype Core;

mapping

ActionState =: Entry;
ActionState =: Exit;
ActionState <> CallAction A_state_entry State_Machines;

Receive.name =: ActionState.name;
(String)( Receive.operation +  "(" + Receive.variable + ")" ) =:
ActionExpression.body;
ActionExpression =: CallAction.script;

$"Receive" =: Stereotype.name;
$"ActionState" =: Stereotype.baseClass;

ActionState <> Stereotype A_stereotype_extendedElement Core;
Process2Model:Model <> Stereotype A_namespace_ownedElement Core;

Process2Model:CompositeState <> ActionState A_container_subvertex
State_Machines;

foreach collection Flow2CompositeState:Trans transvec {
      variables
      Transition javax.jmi.reflect.RefObject;

      mapping
```

```
      transvec!0 =: Transition;

        foreach collection Receive.source sourcevec {
                variables
                Source RefObject;

                mapping
                sourcevec!0 =: Source;
                #Transition.name == Source.linkName#   Transition <>
                ActionState A_outgoing_source State_Machines;
        };
        foreach collection Receive.target targetvec {
                variables
                Target RefObject;

                mapping
                targetvec!0 =: Target;
                #Transition.name == Target.linkName#   Transition <>
                ActionState A_incoming_target State_Machines;
        };

};
```

8.2.11. Reply2ActionState

```
Reply2ActionState

Trafosource
Reply Reply

Trafotarget
ActionState ActionState Activity_Graphs

meta
Entry
Exit

enums

structs

variables
CallAction CallAction Common_Behavior;
ActionExpression ActionExpression Data_Types;
Stereotype Stereotype Core;

mapping

ActionState =: Entry;
ActionState =: Exit;
ActionState <> CallAction A_state_entry State_Machines;

Reply.name =: ActionState.name;
(String)( Reply.operation + "()" + " := " + Reply.variable) =:
ActionExpression.body;
ActionExpression =: CallAction.script;

$"Reply" =: Stereotype.name;
$"ActionState" =: Stereotype.baseClass;

ActionState <> Stereotype A_stereotype_extendedElement Core;
Process2Model:Model <> Stereotype A_namespace_ownedElement Core;

Process2Model:CompositeState <> ActionState A_container_subvertex
State_Machines;
```

```
foreach collection Flow2CompositeState:Trans transVec {
      variables
      Transition javax.jmi.reflect.RefObject;

      mapping
      transVec!0 =: Transition;

        foreach collection Reply.source sourceVec {
                variables
                Source RefObject;

                mapping
                sourceVec!0 =: Source;
                #Transition.name == Source.linkName#   Transition <>
                ActionState A_outgoing_source State_Machines;
        };
        foreach collection Reply.target targetVec {
                variables
                Target RefObject;

                mapping
                targetVec!0 =: Target;
                #Transition.name == Target.linkName#   Transition <>
                ActionState A_incoming_target State_Machines;
        };
};
```

8.2.12. Sequence2CompositeState

```
Sequence2CompositeState

Trafosource
Sequence Sequence

Trafotarget
---

meta
Entry
Exit
Subs

enums

structs

variables

mapping

Sequence.activity -> * =: Subs;

foreach array Subs subsVec{
      variables
      mapping

      subsVec!0.Entry =: Entry;
      subsVec!.Exit =: Exit;
};

foreach collection Subs subsVec {
     variables
     mapping
     foreach collection subsVec!0.Exit exitVec{
           variables
```

```
            mapping
            foreach collection subsVec!1.Entry entryVec{
                  variables
                  Transition Transition State_Machines;

                  mapping
                  exitVec!0 <> Transition A_outgoing_source
                  State_Machines;
                  entryVec!0 <> Transition A_incoming_target
                  State_Machines;

                  Process2Model:ActivityGraph <> Transition
                  A_stateMachine_transitions State_Machines;
            };
      };

};
```

8.2.13. Switch2CompositeState

```
Switch2CompositeState

Trafosource
Switch SwitchKEY

Trafotarget
---

meta
Entry
Exit

enums
PKJoin PseudostateKind Data_Types pk_join
PKJunction PseudostateKind Data_Types pk_junction
PKFork PseudostateKind Data_Types pk_fork

structs

variables
Join Pseudostate State_Machines;
Fork Pseudostate State_Machines;
JoinDecision Pseudostate State_Machines;
ForkDecision Pseudostate State_Machines;
JJTrans Transition State_Machines;
FFTrans Transition State_Machines;

mapping

PKJoin =: Join.kind;
PKJunction =: JoinDecision.kind;
PKJunction =: ForkDecision.kind;
PKFork =: Fork.kind;

Process2Model:CompositeState <> Join A_container_subvertex
State_Machines;
Process2Model:CompositeState <> Fork A_container_subvertex
State_Machines;
Process2Model:CompositeState <> JoinDecision A_container_subvertex
State_Machines;
Process2Model:CompositeState <> ForkDecision A_container_subvertex
State_Machines;

Process2Model:ActivityGraph <> FFTrans A_stateMachine_transitions
State_Machines;
```

```
Process2Model:ActivityGraph <> JJTrans A_stateMachine_transitions
State_Machines;

Join <> JJTrans A_outgoing_source State_Machines;
JoinDecision <> JJTrans A_incoming_target State_Machines;

Fork <> FFTrans A_incoming_target State_Machines;
ForkDecision <> FFTrans A_outgoing_source State_Machines;

Join =: Entry;
Fork =: Exit;

Switch.caseKEY -> *;

Switch.otherwise -> *;
```

8.3. The UML2UML+DI Transformation Definitions

The following sections contain the transformation definitions that make up the UML2UML+DI mapping.

8.3.1. ActionExpression2GraphNode

```
ActionExpression2GraphNode

Trafosource
ActionExpression ActionExpression

Trafotarget
GraphNode GraphNode Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )

variables
SimpleSemanticModelElement SimpleSemanticModelElement
Diagram_Interchange;

mapping

$true =: GraphNode.isVisible;
Point =: GraphNode.position;

$"Expression" =: SimpleSemanticModelElement.typeInfo;
$"" =: SimpleSemanticModelElement.presentation;

GraphNode <> SimpleSemanticModelElement A_graphElement_semanticModel
Diagram_Interchange;
```

8.3.2. ActionState2GraphNode

```
ActionState2GraphNode

Trafosource
```

```
ActionState ActionState

Trafotarget
GraphNode GraphNode Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )

variables
Uml1SemanticModelBridge Uml1SemanticModelBridge Diagram_Interchange;

mapping

$true =: GraphNode.isVisible;
Point =: GraphNode.position;
$"" =: Uml1SemanticModelBridge.presentation;

Uml1SemanticModelBridge <> ActionState
A_uml1SemanticModelBridge_element Diagram_Interchange;
Uml1SemanticModelBridge <> GraphNode A_graphElement_semanticModel
Diagram_Interchange;
ActivityGraph2Diagram:Diagram <> GraphNode A_container_contained
Diagram_Interchange;

ActionState.entry -> *.GraphNode <> GraphNode A_container_contained
Diagram_Interchange;
```

8.3.3. ActivityGraph2Diagram

```
ActivityGraph2Diagram

Trafosource
ActivityGraph ActivityGraph

Trafotarget
Diagram Diagram Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )
Dimension org.omg.uml.diagraminterchange.Dimension
Diagram_Interchange ( Double 400 ) ( Double 300 )

variables
Uml1SemanticModelBridge Uml1SemanticModelBridge Diagram_Interchange;
SimpleSemanticModelElement SimpleSemanticModelElement
Diagram_Interchange;

mapping

$"ActivityDiagram" =: SimpleSemanticModelElement.typeInfo;
$1.0 =: Diagram.zoom;
$true =: Diagram.isVisible;
$"DemoDiagram" =: Diagram.name;

$"" =: Uml1SemanticModelBridge.presentation;
$"" =: SimpleSemanticModelElement.presentation;
```

```
Diagram <> SimpleSemanticModelElement A_graphElement_semanticModel
Diagram_Interchange;
Diagram <> Uml1SemanticModelBridge A_diagram_owner
Diagram_Interchange;
ActivityGraph <> Uml1SemanticModelBridge
A_uml1SemanticModelBridge_element Diagram_Interchange;

Point =: Diagram.viewport;
Point =: Diagram.position;
Dimension =: Diagram.size;

ActivityGraph.top -> *;
ActivityGraph.transitions -> *;
```

8.3.4. AIncomingTarget2GraphConnector

```
AIncomingTarget2GraphConnector

Trafosource
AIncomingTarget AIncomingTarget

Trafotarget
---

meta

structs
Point Point Diagram_Interchange ( Double 0 ) ( Double 0 )

variables

mapping

foreach collection AIncomingTarget.(AllLinks)  links  {
      variables
      GraphConnector GraphConnector Diagram_Interchange;
      Link Object;

      mapping
      Point =: GraphConnector.position;
      links!0 =: Link;

      Link.(FirstEnd) ?> GraphConnector A_graphEdge_anchor
      Diagram_Interchange;
      Link.(SecondEnd) ?> GraphConnector A_graphElement_anchorage
      Diagram_Interchange;
};
```

8.3.5. AOutgoingSource2GraphConnector

```
AOutgoingSource2GraphConnector

Trafosource
AOutgoingSource AOutgoingSource

Trafotarget
---

meta

enums

structs
```

```
Point Point Diagram_Interchange ( Double 0 ) ( Double 0 )

variables

mapping
foreach collection AOutgoingSource.(AllLinks)  links  {
      variables
      GraphConnector GraphConnector Diagram_Interchange;
      Link Object;

      mapping

      Point =: GraphConnector.position;
      links!0 =: Link;

      Link.(FirstEnd) ?> GraphConnector A_graphEdge_anchor
      Diagram_Interchange;
      Link.(SecondEnd) ?> GraphConnector A_graphElement_anchorage
      Diagram_Interchange;
};
```

8.3.6. CallAction2GraphNode

```
CallAction2GraphNode

Trafosource
CallAction CallAction

Trafotarget
GraphNode GraphNode Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )

variables
Uml1SemanticModelBridge Uml1SemanticModelBridge Diagram_Interchange;

mapping
$true =: GraphNode.isVisible;
Point =: GraphNode.position;
$"" =: Uml1SemanticModelBridge.presentation;

Uml1SemanticModelBridge <> CallAction
A_uml1SemanticModelBridge_element Diagram_Interchange;
Uml1SemanticModelBridge <> GraphNode A_graphElement_semanticModel
Diagram_Interchange;

CallAction.script -> *.GraphNode <> GraphNode A_container_contained
Diagram_Interchange;
```

8.3.7. CompositeState2GraphNode

```
CompositeState2GraphNode

Trafosource
CompositeState CompositeState

Trafotarget
---
```

```
meta

enums

structs

variables

mapping

CompositeState.subvertex -> *;
```

8.3.8. Pseudostate2GraphNode

```
Pseudostate2GraphNode

Trafosource
Pseudostate Pseudostate

Trafotarget
GraphNode GraphNode Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )

variables
Uml1SemanticModelBridge Uml1SemanticModelBridge Diagram_Interchange;

mapping
$true =: GraphNode.isVisible;
Point =: GraphNode.position;
$"" =: Uml1SemanticModelBridge.presentation;

Uml1SemanticModelBridge <> Pseudostate
A_uml1SemanticModelBridge_element Diagram_Interchange;
Uml1SemanticModelBridge <> GraphNode A_graphElement_semanticModel
Diagram_Interchange;

ActivityGraph2Diagram:Diagram <> GraphNode A_container_contained
Diagram_Interchange;
```

8.3.9. Transition2GraphEdge

```
Transition2GraphEdge

Trafosource
Transition Transition

Trafotarget
GraphEdge GraphEdge Diagram_Interchange

meta

enums

structs
Point org.omg.uml.diagraminterchange.Point Diagram_Interchange (
Double 0 ) ( Double 0 )
```

```
variables
Uml1SemanticModelBridge Uml1SemanticModelBridge Diagram_Interchange;

mapping
$true  =: GraphEdge.isVisible;
Point =: GraphEdge.position;
$"" =: Uml1SemanticModelBridge.presentation;

Uml1SemanticModelBridge <> Transition
A_uml1SemanticModelBridge_element Diagram_Interchange;
Uml1SemanticModelBridge <> GraphEdge A_graphElement_semanticModel
Diagram_Interchange;

ActivityGraph2Diagram:Diagram <> GraphEdge A_container_contained
Diagram_Interchange;
```

8.3.10. Uml2UmlDI

```
Uml2UmlDI

Trafosource
UmlPackageSource UmlPackage

Trafotarget
UmlPackageTarget UmlPackage

meta

enums

structs

variables

mapping
UmlPackageSource.Activity_Graphs.ActivityGraph -> *;
UmlPackageSource.State_Machines.A_outgoing_source -> *;
UmlPackageSource.State_Machines.A_incoming_target -> *;
```

8.4. SableCC Grammar for Marius' Transformation Language

```
Package     marius;

Helpers
       letter = (['A' .. 'z'] | '_' | '*');
       digit = ['0' .. '9'];
       extlett = (['A' .. 'z'] | '_' | ':' | '=' | '+' | '-' );
       symbols = ('+' | '-' | '*' | '/' | ',' | ':' | '=' | ';' |
                  ' ' | '(' | ')' | '!' | '§' | '$' | '%' | '&' |
                  '?' | '#' | '_' | '.' | '>' | '<' );

Tokens
      meta_lit = 'meta';
      structs_lit = 'structs';
      enums_lit = 'enums';
      source_lit = 'Trafosource';
      target_lit = 'Trafotarget';
      mapping_lit = 'mapping';
      variables_lit = 'variables';
      foreach_lit = 'foreach';
      index_var = 'i';
      null = '---';
```

```
      returnvalue = '*';

      operator = ('+' | '-' | '*' | '/' | ',');
      boolean = 'true' | 'false';

      identifier = letter (letter | digit)*;
      number = digit*;
      exclamation = '"';
      exclamationidentifier = '"' (letter | symbols)* '"';

      collection_lit = 'collection';
      array_lit = 'array';
      condtag = '#';
      siblink = '?>';
      normlink = '<>';
      colon = ':';
      special = '!';
      idxopen = '[';
      idxclose = ']';
      argopen = '(';
      argclose = ')';
      open = '{';
      close = '}';
      strstart = '$';
      begin = 'BEGIN';
      end = 'END';
      dot = '.';

      arrow = '->';
      equal = '=:';
      isequal = '==';

      semicolon = ';';
      blank = (' ' | 13 | 10 | 10 13 | '\n' | '\t' | 13 10)+;

Ignored Tokens
      blank;

Productions
  toplevel =
    title
    source
    target
    meta
    enums
    structs
    variables
    mapping;

    title = identifier;

    source        = source_lit sourceelement;
    sourceelement = identifier name type?;

    target        = target_lit targetelement;
    targetelement =
      {identifier} identifier name type? |
      {null} null;

    meta = meta_lit identifier*;

    enums        = enums_lit enumselement*;
    enumselement = identifier type package name;

    structs        = structs_lit structselement*;
    structselement = identifier type package arglist*;
```

```
arglist = argopen arg argclose;
arg =
  {name} name |
  {number} name number |
  {identifier} name identifier;

variables       = variables_lit variableselement*;
variableselement=
    identifier
    type
    package?
    semicolon;

method = dot argopen name argclose;
type   = identifier subtype*;
subtype = dot identifier;
mapping = mapping_lit trafos;

equalrightside = isequal name attribute?  condtag;
condition = condtag name attribute? equalrightside;

name        = identifier;
vectorname  = identifier;
parent      = identifier colon;
association = identifier;
package     = identifier;

dotnumber = dot number;
string =
  {boolean} boolean |
  {number} number |
  {numberdot} number dotnumber |
  {name}  exclamationidentifier;

index = number;

cast       = argopen name argclose;
connector  = operator;
concatelem =
   {name} name attribute* |
   {string} string;
connectorconcatelem = connector concatelem;
concat     = argopen concatelem connectorconcatelem* argclose;

foreachtype = identifier;
foreachheadline =
  {parent} parent name vectorname open |
  {normal} name attribute* method? vectorname open |
  {special} name special index attribute* vectorname open;
foreach =
  foreach_lit foreachtype foreachheadline
  variables
  mapping
  close;

condtrafo = condition? trafo semicolon;
trafos = condtrafo+;

trafo =
    {foreach}          foreach                              |
    {string2equ}       string_sourceexpr equ_targetexpr     |
    {name2arr}         name_sourceexpr arr_targetexpr       |
    {name2equ}         name_sourceexpr equ_targetexpr       |
    {name2normlink}    name_sourceexpr normlink_targetexpr  |
    {name2siblink}     name_sourceexpr siblink_targetexpr   |
    {parent2arr}       parent_sourceexpr arr_targetexpr     |
    {parent2equ}       parent_sourceexpr equ_targetexpr     |
```

```
        {parent2normlink}   parent_sourceexpr normlink_targetexpr |
        {parent2siblink}    parent_sourceexpr siblink_targetexpr  |
        {concat2equ}        concat_sourceexpr equ_targetexpr      |
        {collection2arr}    collection_sourceexpr arr_targetexpr  |
        {collection2equ}    collection_sourceexpr equ_targetexpr  |
        {collection2normlink}      collection_sourceexpr
                                   normlink_targetexpr           |
        {collection2siblink}       collection_sourceexpr
                                   siblink_targetexpr   ;

      string_sourceexpr   = strstart string;
      name_sourceexpr     = name attribute* method?;
      parent_sourceexpr     = parent name;
      concat_sourceexpr     = cast concat;
      collection_sourceexpr = vectorname special index? attribute*
                              method?;

      arr_targetexpr      = arrow returnvalue attribute*
                            targetexprextension*;
      equ_targetexpr      = equal name attribute*;
      normlink_targetexpr = normlink name association package;
      siblink_targetexpr  = siblink name association package;

      targetexprextension =
         {normlink} normlink_targetexpr |
         {equ} equ_targetexpr;

      attribute = dot name;
```

8.5. “PurchaseOrder” Example Transformation

Below is a BPEL listing describing the “PurchaseOrder” business process. The file is an adaptation from IBM’s original “purchase.bpel” distributed as a sample with alphaWorks’ BPEL engine [BPWS]. It is fitted to be compatible with BPEL version 1.1.

```
<process name="purchaseOrderProcess"
         targetNamespace="http://acme.com/ws-bp/purchase"
         xmlns:lns="http://manufacturing.org/wsdl/purchase"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
process/">

  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice"
              messageType="lns:InvMessage"/>
    <variable name="POFault"
              messageType="lns:orderFaultType"/>
    <variable name="shippingRequest"
              messageType="lns:shippingRequestMessage"/>
    <variable name="shippingInfo"
              messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule"
              messageType="lns:scheduleMessage"/>
  </variables>

  <partnerLinks>
    <partnerLink name="customer"
             partnerLinkType="lns:purchaseLT"
             myRole="purchaseService"/>
    <partnerLink name="invoiceProvider"
```

```
            partnerLinkType="lns:invoiceLT"
            partnerRole="invoiceService"/>
    <partnerLink name="shippingProvider"
            partnerLinkType="lns:shippingLT"
            partnerRole="shippingService"/>
    <partnerLink name="schedulingProvider"
            partnerLinkType="lns:schedulingLT"
            partnerRole="schedulingService"/>
  </partnerLinks>

  <sequence>
    <receive partnerLink="customer"
          name="receivePO"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder" variable="PO">
    </receive>
    <flow>
      <links>
        <link name="ship-to-invoice"/>
        <link name="ship-to-scheduling"/>
      </links>
      <sequence>
        <assign name="initialiseShippingRequest">
          <copy>
            <from variable="PO" part="customerInfo"/>
            <to variable="shippingRequest"
                part="customerInfo"/>
          </copy>
        </assign>
        <invoke partnerLink="shippingProvider"
                name="requestShipping"
                portType="lns:shippingPT"
                operation="requestShipping"
                inputVariable="shippingRequest"
                outputVariable="shippingInfo">
          <source linkName="ship-to-invoice"/>
        </invoke>
        <receive partnerLink="shippingProvider"
                 name="receiveSchedule"
                 portType="lns:shippingCallbackPT"
                 operation="sendSchedule"
                 variable="shippingSchedule">
          <source linkName="ship-to-scheduling"/>
        </receive>
      </sequence>
      <sequence>
        <invoke partnerLink="invoiceProvider"
                name="initiatePriceCalculation"
                portType="lns:computePricePT"
                operation="initiatePriceCalculation"
                inputVariable="PO">
        </invoke>
        <invoke partnerLink="invoiceProvider"
                name="sendShippingPrice"
                portType="lns:computePricePT"
                operation="sendShippingPrice"
                inputVariable="shippingInfo">
          <target linkName="ship-to-invoice"/>
        </invoke>
        <receive partnerLink="invoiceProvider"
                 name="receiveInvoice"
                 portType="lns:invoiceCallbackPT"
                 operation="sendInvoice"
                 variable="Invoice"/>
      </sequence>
      <sequence>
        <invoke partnerLink="schedulingProvider"
```

```
                name="requestScheduling"
                portType="lns:schedulingPT"
                operation="requestProductionScheduling"
                inputVariable="PO">
        </invoke>
        <invoke partnerLink="schedulingProvider"
                name="sendShippingSchedule"
                portType="lns:schedulingPT"
                operation="sendShippingSchedule"
                inputVariable="shippingSchedule">
          <target linkName="ship-to-scheduling"/>
        </invoke>
      </sequence>
    </flow>
    <reply partnerLink="customer" portType="lns:purchasePT"
           name="returnInvoice"
           operation="sendPurchaseOrder"
           variable="Invoice"/>
  </sequence>
</process>
```

The following Activity Diagram in Figure 24 is the result of the above BPEL code being subject to the MyBPEL2UML and the UML2UML+DI mapping. However, due to the fact that Poseidon is not supporting „swimlanes“, this missing feature is added manually. The „PurchaseOrder“ example is also used in [IBM03]. However, there is a small difference between the PurchaseOrder Avtivity Diagram proposed there and the one used in this work. In comparison, the „ship-to-invoice“ transition in [IBM03] points into the opposite direction. Yet, this work interprets the „ship-to-invoice“ transition only to be meaningful in the context of this business process if it is directed as below.

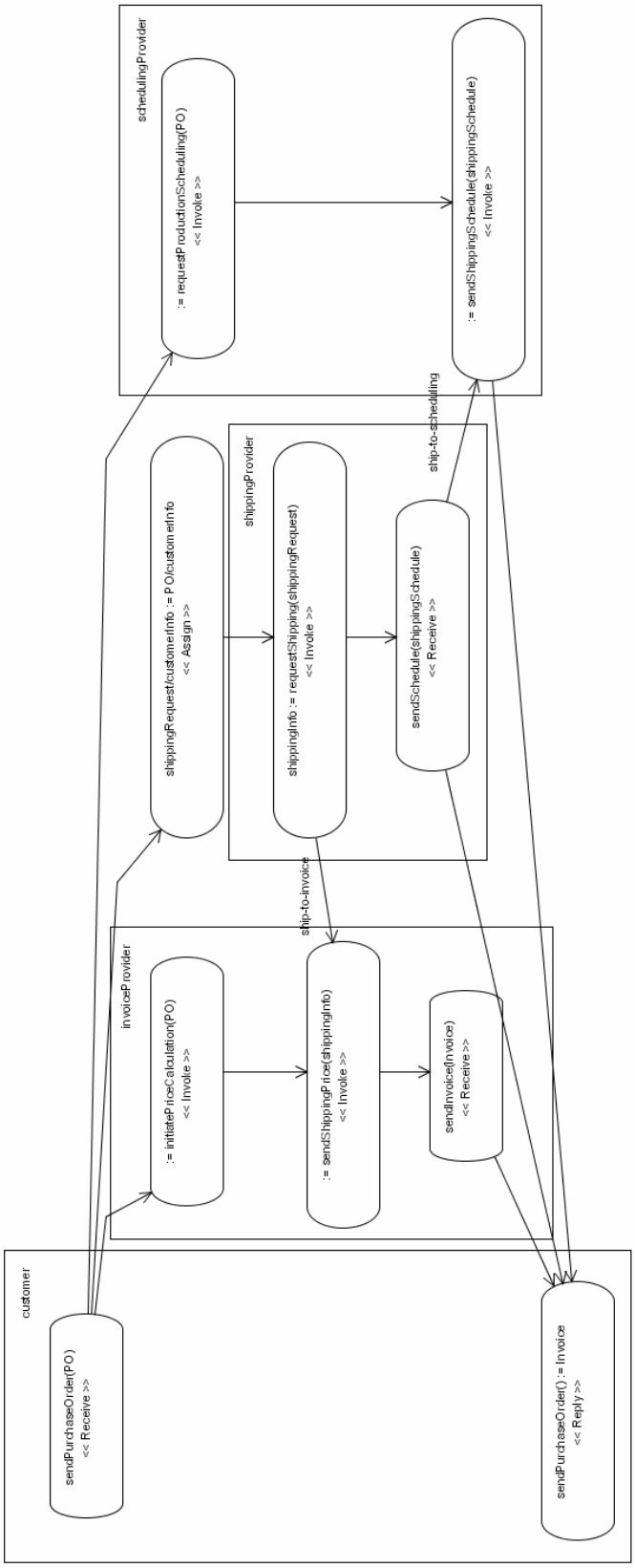

Figure 24 The 'PurchaseOrder' Activity Diagram

8.6. "Marketplace" Example Transformation

Below is a BPEL listing describing the "Marketplace" business process. The file is an adaptation from IBM's original "marketplace.bpel" distributed as a sample with alphaWorks' BPEL engine [BPWS]. It is fitted to be compatible with BPEL version 1.1.

```
<process name="marketplace"
targetNamespace="urn:samples:marketplaceService"
xmlns:tns="urn:samples:marketplaceService"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
   <partnerLinks>
      <partnerLink name="seller"
                   partnerLinkType="tns:salesSLT"
                   myRole="sales" />
      <partnerLink name="buyer"
                   partnerLinkType="tns:buyingSLT"
                   myRole="buying" />
   </partnerLinks>

   <variables>
      <variable name="sellerInfo"
                messageType="tns:sellerInfoMessage" />
      <variable name="buyerInfo"
                messageType="tns:buyerInfoMessage" />
      <variable name="negotiationOutcome"
                messageType="tns:negotiationMessage" />
   </variables>

   <sequence name="MarketplaceSequence">
      <flow name="MarketplaceFlow">
         <receive partnerLink="seller"
                  portType="tns:sellerPT"
                  operation="submit"
                  variable="sellerInfo"
                  createInstance="yes"
                  name="SellerReceive">
         </receive>
         <receive partnerLink="buyer"
                  portType="tns:buyerPT"
                  operation="submit"
                  variable="buyerInfo"
                  createInstance="yes"
                  name="BuyerReceive">
         </receive>
      </flow>

      <switch name="MarketplaceSwitch">
         <case condition="bpws:getContainerData('sellerInfo',
                         'askingPrice') &lt;=
                         bpws:getContainerData('buyerInfo',
                         'offer')">
            <assign name="SuccessAssign">
               <copy>
                 <from expression="'Deal Successful'" />
                 <to variable="negotiationOutcome" part="outcome" />
               </copy>
            </assign>
         </case>
         <otherwise>
            <assign name="FailedAssign">
               <copy>
```

```
                <from expression="'Deal Failed'" />

                <to variable="negotiationOutcome" part="outcome" />
              </copy>
            </assign>
          </otherwise>
        </switch>
        <flow>
          <reply partnerLink="seller"
                 portType="tns:sellerPT"
                 operation="submit"
                 variable="negotiationOutcome"
                 name="SellerReply" />
          <reply partnerLink="buyer"
                 portType="tns:buyerPT"
                 operation="submit"
                 variable="negotiationOutcome"
                 name="BuyerReply" />
        </flow>
    </sequence>
</process>
```

Analogous to the previous example in 8.5, the following Activity Diagram in Figure 25 is the transformation result of the above BPEL listing. The condition in the "case-path" is not rendered as an XPath expression, but imported from the BPEL definition unchanged.

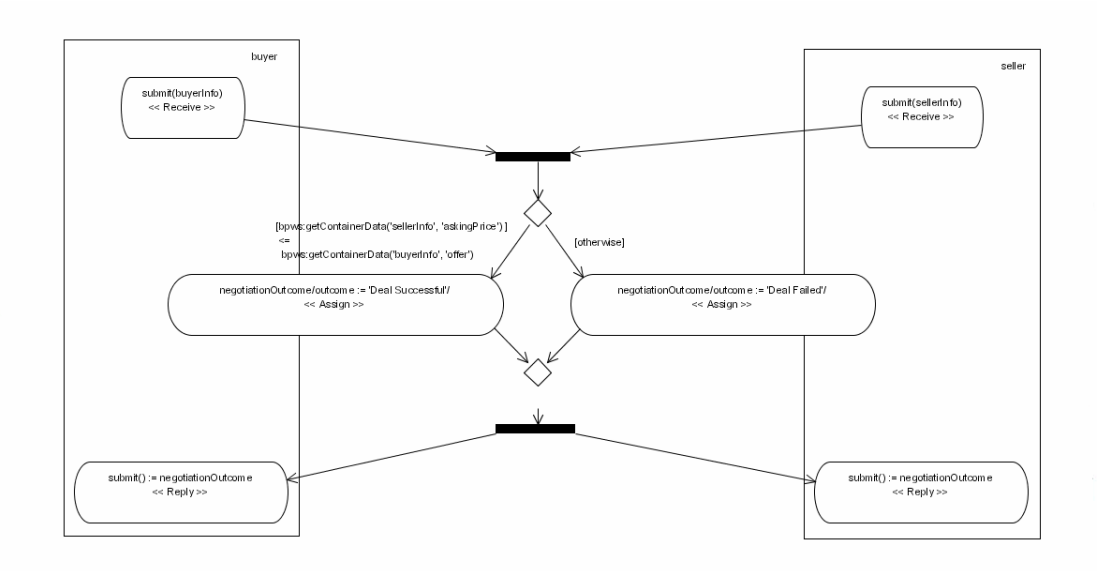

Figure 25 The 'Marketplace' Activity Diagram

8.7. Parser XSL-Sheet for MyBPEL 1.1

```
<?xml version="1.0"?>
<xsl:stylesheet   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                     xmlns:java="http://xml.apache.org/xslt/java"
                       xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/
                       business-process/"
                  version="1.0">
<xsl:output method="xml" indent="yes"/>
<xsl:param name="myBpel"/>
```

```
<!-- ========================================================= -->
<!-- parse <activity> -->
<xsl:template name="parse-standard-stuff">
  <xsl:param name="child"/>
  <xsl:param name="name"/>
  <xsl:param name="suppressJoinFailure"/>
  <xsl:param name="joinCondition"/>

 <xsl:if test="$name">
     <xsl:if test="java:setName($child, $name)"/>
 </xsl:if>
  <xsl:if test="$suppressJoinFailure">
    <xsl:choose>
      <xsl:when test="$suppressJoinFailure = 'yes'">
        <xsl:if test="java:setSuppressJoinFailure($child, true ())"/>
      </xsl:when>
      <xsl:when test="$suppressJoinFailure='no'">
        <xsl:if test="java:setSuppressJoinFailure($child, false ())"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
          <xsl:text>Value for '</xsl:text>
          <xsl:value-of select="$suppressJoinFailure"/>
          <xsl:text>' not allowed for 'suppressJoinFailure'!
          </xsl:text>
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:if>
  <xsl:if test="$joinCondition">
    <xsl:if test="java:setJoinCondition ($child, $joinCondition)"/>
  </xsl:if>

  <xsl:for-each select="bpws:source">
    <xsl:variable name="sourceClass"
                  select="java:getSource($myBpel)"/>
    <xsl:variable name="source"
                  select="java:createSource($sourceClass)"/>
    <xsl:variable name="activityHassource"
                  select="java:getActivityHasSource($myBpel)"/>
    <xsl:if test="java:add($activityHassource, $child, $source)"/>
    <xsl:if test="@linkName">
      <xsl:if test="java:setLinkName ($source, @linkName)"/>
    </xsl:if>

    <xsl:if test="@transitionCondition">
      <xsl:if test="java:setTransitionCondition ($source,
                    @transitionCondition)"/>
    </xsl:if>
  </xsl:for-each>

  <xsl:for-each select="bpws:target">
    <xsl:variable name="targetClass"
                  select="java:getTarget($myBpel)"/>
    <xsl:variable name="target"
                  select="java:createTarget($targetClass)"/>
    <xsl:variable name="activityHastarget"
                  select="java:getActivityHasTarget($myBpel)"/>
    <xsl:if test="java:add($activityHastarget, $child, $target)"/>
    <xsl:if test="@linkName">
      <xsl:if test="java:setLinkName($target, @linkName)"/>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <variables> -->
<xsl:template match="bpws:variables">
  <xsl:param name="parent"/>
  <xsl:variable name="variablesClass"
                select="java:getVariables($myBpel)"/>
  <xsl:variable name="variables"
                select="java:createVariables($variablesClass)"/>
  <xsl:variable name="variablesHasvariable"
                select="java:getVariablesHasVariable($myBpel)"/>

  <xsl:if test="java:setVariables($parent, $variables)"/>
  <xsl:for-each select="bpws:variable">
    <xsl:variable name="variableClass"
                  select="java:getVariable($myBpel)"/>
    <xsl:variable name="variable"
                  select="java:createVariable($variableClass)"/>

    <xsl:if test="@name">
```

```
      <xsl:if test="java:setName($variable, @name)"/>
    </xsl:if>
    <xsl:if test="@messageType">
      <xsl:if test="java:setMessageType ($variable, @messageType)"/>
    </xsl:if>
    <xsl:if test="java:add($variablesHasvariable, $variables,
                  $variable)"/>
  </xsl:for-each>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <partnerLinks> -->
<xsl:template match="bpws:partnerLinks">
  <xsl:param name="parent"/>
  <xsl:variable name="partnerLinksClass"
                select="java:getPartnerLinks($myBpel)"/>
  <xsl:variable name="partnerLinks"
                select="java:createPartnerLinks($partnerLinksClass)"/>


  <xsl:if test="java:setPartnerLinks($parent, $partnerLinks)"/>
  <xsl:variable name="partnerLinksHaspartnerLink"
                select="java:getPartnerLinksHasPartnerLink($myBpel)"/>

  <xsl:for-each select="bpws:partnerLink">
    <xsl:variable name="partnerLinkClass"
                  select="java:getPartnerLink($myBpel)"/>
    <xsl:variable name="partnerLink"
                select="java:createPartnerLink($partnerLinkClass)"/>

    <xsl:if test="@name">
      <xsl:if test="java:setName($partnerLink, @name)"/>
    </xsl:if>
    <xsl:if test="@partnerLinkType">
      <xsl:if test="java:setPartnerLinkType($partnerLink,@partnerLinkType)"/>
    </xsl:if>
    <xsl:if test="@myRole">
      <xsl:if test="java:setMyRole($partnerLink, @myRole)"/>
    </xsl:if>
    <xsl:if test="@partnerRole">
      <xsl:if test="java:setPartnerRole($partnerLink,@partnerRole)"/>
    </xsl:if>

    <xsl:if test="java:add($partnerLinksHaspartnerLink,
                  $partnerLinks, $partnerLink)"/>
  </xsl:for-each>
</xsl:template>

<!-- ========================================================= -->
<!-- parse <partners> -->
<xsl:template match="bpws:partners">
  <xsl:param name="parent"/>
  <xsl:variable name="partnersClass"
                select="java:getPartners($myBpel)"/>
  <xsl:variable name="partners"
                select="java:createPartners($partnersClass)"/>

  <xsl:if test="java:setPartners($parent, $partners)"/>

  <xsl:variable name="partnersHaspartner"
                select="java:getPartnersHasPartner($myBpel)"/>

  <xsl:for-each select="bpws:partner">
    <xsl:variable name="partnerClass"
                  select="java:getPartner($myBpel)"/>
    <xsl:variable name="partner"
                  select="java:createPartner($partnerClass)"/>
    <xsl:if test="@name">
      <xsl:if test="java:setName($partner, @name)"/>
    </xsl:if>
    <xsl:if test="java:add($partnersHaspartner, $partners, $partner)"/>
  </xsl:for-each>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <reveive> -->
<xsl:template match="bpws:receive">
  <xsl:param name="parent"/>
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="receiveClass"
                select="java:getReceive($myBpel)"/>
  <xsl:variable name="receive"
                select="java:createReceive($receiveClass)"/>

  <xsl:call-template name="parse-standard-stuff">
```

```
    <xsl:with-param name="child" select="$receive"/>
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

  <xsl:if test="@partner">
    <xsl:if test="java:setPartner($receive, @partner)"/>
  </xsl:if>
  <xsl:if test="@portType">
    <xsl:if test="java:setPortType($receive, @portType)"/>
  </xsl:if>
  <xsl:if test="@operation">
    <xsl:if test="java:setOperation($receive, @operation)"/>
  </xsl:if>
  <xsl:if test="@variable">
     <xsl:if test="java:setVariable($receive, @variable)"/>
  </xsl:if>

  <xsl:if test="@createInstance">
    <xsl:choose>
      <xsl:when test="@createInstance='yes'">
        <xsl:if test="java:setCreateInstance ($receive, true ())"/>
      </xsl:when>
      <xsl:when test="@createInstance='no'">
        <xsl:if test="java:setCreateInstance ($receive, false ())"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
          <xsl:text>Value '</xsl:text>
          <xsl:value-of select="@createInstance"/>
          <xsl:text>' for 'createInstance' not allowed.
          </xsl:text>
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:if>

  <xsl:if test="java:add($parentHaschild, $parent, $receive)"/>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <reply> -->
<xsl:template match="bpws:reply">
  <xsl:param name="parent"/>
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="replyClass"
                select="java:getReply($myBpel)"/>
  <xsl:variable name="reply"
                select="java:createReply($replyClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$reply"/>
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

   <xsl:if test="@partner">
    <xsl:if test="java:setPartner ($reply, @partner)"/>
  </xsl:if>
  <xsl:if test="@portType">
    <xsl:if test="java:setPortType ($reply, @portType)"/>
  </xsl:if>
  <xsl:if test="@operation">
    <xsl:if test="java:setOperation ($reply, @operation)"/>
  </xsl:if>
  <xsl:if test="@variable">
    <xsl:if test="java:setVariable ($reply, @variable)"/>
  </xsl:if>
  <xsl:if test="@faultName">
    <xsl:if test="java:setFaultName ($reply, @faultName)"/>
  </xsl:if>

  <xsl:if test="java:add($parentHaschild, $parent, $reply)"/>
</xsl:template>
<!--  ======================================================   -->
<!--  parse <empty>   -->
<xsl:template match="bpws:empty">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
```

```
  <xsl:variable name="emptyClass"
                select="java:getEmpty($myBpel)"/>
  <xsl:variable name="empty"
                select="java:createEmpty($emptyClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$empty"/>
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

 <xsl:if test="java:add($parentHaschild, $parent, $empty)"/>
</xsl:template>
<!--  ========================================================   -->
<!--  parse <while>   -->
<xsl:template match="bpws:while">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="whileClass"
                select="java:getWhileKey($myBpel)"/>
  <xsl:variable name="while"
                select="java:createWhileKey($whileClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$while"/>
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure" select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>
  <xsl:if test="@condition">
    <xsl:if test="java:setCondition($while, @condition)"/>
  </xsl:if>

  <xsl:if test="java:add($parentHaschild, $parent, $while)"/>
  <xsl:variable name="whileHasactivity"
                select="java:getWhileHasActivity($myBpel)"/>

  <xsl:apply-templates select="*">
    <xsl:with-param name="parent" select="$while"/>
    <xsl:with-param name="parentHaschild" select="$whileHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!--  ========================================================   -->
<!--  parse <case>   -->
<xsl:template match="bpws:case">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="caseClass"
                select="java:getCaseKey($myBpel)"/>
  <xsl:variable name="case"
                select="java:createCaseKey($caseClass)"/>

  <xsl:if test="@condition">
    <xsl:if test="java:setCondition($case, @condition)"/>
  </xsl:if>

  <xsl:if test="java:add($parentHaschild, $parent, $case)"/>
  <xsl:variable name="caseHasactivity"
                select="java:getCaseHasActivity($myBpel)"/>

  <xsl:apply-templates select="*">
    <xsl:with-param name="parent" select="$case"/>
    <xsl:with-param name="parentHaschild"
                    select="$caseHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!--  ========================================================   -->
<!--  parse <otherwise>   -->
<xsl:template match="bpws:otherwise">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="otherwiseClass"
                select="java:getOtherwise($myBpel)"/>
  <xsl:variable name="otherwise"
                select="java:createOtherwise($otherwiseClass)"/>

  <xsl:if test="java:add($parentHaschild, $parent, $otherwise)"/>
  <xsl:variable name="otherwiseHasactivity"
                select="java:getOtherwiseHasActivity($myBpel)"/>

  <xsl:apply-templates select="*">
```

```
    <xsl:with-param name="parent" select="$otherwise"/>
    <xsl:with-param name="parentHaschild"
                    select="$otherwiseHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!--  ========================================================   -->
<!--  parse <switch>   -->
<xsl:template match="bpws:switch">
  <xsl:param name="parent"/>
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="switchClass"
                select="java:getSwitchKey($myBpel)"/>
  <xsl:variable name="switch"
                select="java:createSwitchKey($switchClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$switch" />
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

  <xsl:if test="java:add($parentHaschild, $parent, $switch)"/>
  <xsl:variable name="switchHascase"
                select="java:getSwitchHasCase($myBpel)"/>
  <xsl:variable name="switchHasotherwise"
                select="java:getSwitchHasOtherwise($myBpel)"/>

  <xsl:apply-templates select="bpws:case">
    <xsl:with-param name="parent" select="$switch" />
    <xsl:with-param name="parentHaschild" select="$switchHascase"/>
  </xsl:apply-templates>
  <xsl:apply-templates select="bpws:otherwise">
    <xsl:with-param name="parent" select="$switch" />
    <xsl:with-param name="parentHaschild"
                    select="$switchHasotherwise"/>
  </xsl:apply-templates>
</xsl:template>
<!--  ========================================================   -->
<!--  parse <invoke>   -->
<xsl:template match="bpws:invoke">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="invokeClass"
                select="java:getInvoke($myBpel)"/>
  <xsl:variable name="invoke"
                select="java:createInvoke($invokeClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$invoke" />
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

  <xsl:if test="@partnerLink">
    <xsl:if test="java:setPartnerLink($invoke, @partnerLink)" />
  </xsl:if>
  <xsl:if test="@portType">
    <xsl:if test="java:setPortType($invoke, @portType)" />
  </xsl:if>
  <xsl:if test="@operation">
    <xsl:if test="java:setOperation($invoke, @operation)" />
  </xsl:if>
  <xsl:if test="@inputVariable">
    <xsl:if test="java:setInputVariable($invoke, @inputVariable)" />
  </xsl:if>
  <xsl:if test="@outputVariable">
    <xsl:if test="java:setOutputVariable($invoke, @outputVariable)" />
  </xsl:if>
 <xsl:if test="java:add($parentHaschild, $parent, $invoke)"/>
</xsl:template>
<!-- ========================================================== -->
<!-- parse - <from> -->
<xsl:template name="parse-from">
  <xsl:param name="from-elem" />
  <xsl:param name="from" />
  <xsl:variable name="fe" select="$from-elem" />

  <xsl:choose>
    <xsl:when test="$fe/@variable">
```

```
      <xsl:if test="java:setVariable ($from, $fe/@variable)" />
      <xsl:choose>
        <xsl:when test="$fe/@part">
          <xsl:if test="java:setPart ($from, $fe/@part)" />
        </xsl:when>
        <xsl:when test="$fe/@property">
          <xsl:if test="java:setProperty ($from, $fe/@property)" />
        </xsl:when>
      </xsl:choose>
      <xsl:if test="$fe/@query">
        <xsl:if test="java:setQuery ($from, $fe/@query)" />
      </xsl:if>
    </xsl:when>
    <xsl:when test="$fe/@partnerLink">
      <xsl:if test="java:setPartnerLink($from, $fe/@partnerLink)" />
        <xsl:if test="$fe/@endpointReference">
        <xsl:if test="java:setEndpointReference($from,
                      $fe/@endpointReference)" />
      </xsl:if>
    </xsl:when>
    <xsl:when test="$fe/@expression">
      <xsl:if test="java:setExpression($from, $fe/@expression)" />
    </xsl:when>
    <xsl:otherwise>

    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <to> -->
<xsl:template name="parse-to">
  <xsl:param name="to-elem" />
  <xsl:param name="to" />
  <xsl:variable name="te" select="$to-elem" />
  <xsl:choose>
      <xsl:when test="$te/@variable">
        <xsl:if test="java:setVariable($to, $te/@variable)" />
        <xsl:choose>
          <xsl:when test="$te/@part">
            <xsl:if test="java:setPart($to, $te/@part)" />
          </xsl:when>
          <xsl:when test="$te/@property">
            <xsl:if test="java:setProperty($to, $te/@property)" />
          </xsl:when>
        </xsl:choose>
        <xsl:if test="$te/@query">
           <xsl:if test="java:setQuery($to, $te/@query)" />
        </xsl:if>
      </xsl:when>
    <xsl:when test="$te/@partnerLink">
      <xsl:if test="java:setPartnerLink($to, $te/@partnerLink)" />
    </xsl:when>
  </xsl:choose>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <copy> -->
<xsl:template name="parse-copy">
  <xsl:param name="assign" />
  <xsl:variable name="copyClass" select="java:getCopy($myBpel)" />
  <xsl:variable name="copy"
                select="java:createCopy($copyClass)" />
  <xsl:variable name="assignHascopy"
                select="java:getAssignHasCopy($myBpel)"/>
  <xsl:if test="java:add($assignHascopy, $assign, $copy)"/>

  <xsl:variable name="fromClass" select="java:getFrom($myBpel)" />
  <xsl:variable name="from"
                select="java:createFrom($fromClass)" />
  <xsl:if test="java:setFrom($copy, $from)" />

  <xsl:call-template name="parse-from">
    <xsl:with-param name="from-elem" select="bpws:from" />
    <xsl:with-param name="from" select="$from" />
  </xsl:call-template>

  <xsl:variable name="toClass" select="java:getTo($myBpel)" />
  <xsl:variable name="to"      select="java:createTo($toClass)" />
  <xsl:if test="java:setTo($copy, $to)" />

  <xsl:call-template name="parse-to">
    <xsl:with-param name="to-elem" select="bpws:to" />
    <xsl:with-param name="to" select="$to" />
  </xsl:call-template>
```

```
</xsl:template>
<!-- ========================================================= -->
<!-- parse <assign> -->
<xsl:template match="bpws:assign">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild" />
  <xsl:variable name="assignClass"
                select="java:getAssign($myBpel)" />
  <xsl:variable name="assign"
                select="java:createAssign($assignClass)" />

  <!--  parse-standard-stuff   -->
  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$assign" />
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

 <xsl:for-each select="bpws:copy">
     <xsl:call-template name="parse-copy">
       <xsl:with-param name="assign" select="$assign" />
     </xsl:call-template>
 </xsl:for-each>

 <xsl:if test="java:add($parentHaschild, $parent, $assign)"/>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <links> -->
<xsl:template match="bpws:links"  mode="__flow__">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild" />
  <xsl:variable name="linksClass"
                select="java:getLinks($myBpel)"/>
  <xsl:variable name="links"
                select="java:createLinks($linksClass)"/>
  <xsl:variable name="linksHaslink"
                select="java:getLinksHasLink($myBpel)"/>

  <xsl:if test="java:add($parentHaschild, $parent, $links)"/>
      <xsl:for-each select="bpws:link">
         <xsl:variable name="linkClass"
                       select="java:getLink($myBpel)"/>
         <xsl:variable name="link"
                       select="java:createLink($linkClass)"/>
       <xsl:if test="@name">
          <xsl:if test="java:setName ($link, @name)" />
       </xsl:if>
       <xsl:if test="java:add($linksHaslink, $links, $link)" />
    </xsl:for-each>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <flow> -->
<xsl:template match="bpws:flow">
  <xsl:param name="parent" />
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="flowClass"
                select="java:getFlow($myBpel)"/>
  <xsl:variable name="flow"
                select="java:createFlow($flowClass)"/>
  <xsl:if test="java:add($parentHaschild, $parent, $flow)"/>

  <xsl:call-template name="parse-standard-stuff">
   <xsl:with-param name="child" select="$flow" />
   <xsl:with-param name="name" select="@name"/>
   <xsl:with-param name="suppressJoinFailure"
                   select="@suppressJoinFailure"/>
   <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

  <xsl:variable name="flowHasactivity"
                select="java:getFlowHasActivity($myBpel)"/>
  <xsl:variable name="flowHaslinks"
                select="java:getFlowHasLinks($myBpel)"/>

  <xsl:apply-templates select="bpws:links"  mode="__flow__">
    <xsl:with-param name="parent" select="$flow" />
    <xsl:with-param name="parentHaschild" select="$flowHaslinks"/>
  </xsl:apply-templates>
  <xsl:apply-templates select="*[name() != 'bpws:links']">
    <xsl:with-param name="parent" select="$flow" />
    <xsl:with-param name="parentHaschild"
```

```
                    select="$flowHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!-- ========================================================= -->
<!-- parse <sequence> -->
<xsl:template match="bpws:sequence">
  <xsl:param name="parent"/>
  <xsl:param name="parentHaschild"/>
  <xsl:variable name="sequenceClass"
                select="java:getSequence($myBpel)"/>
  <xsl:variable name="sequence"
                select="java:createSequence($sequenceClass)"/>

  <xsl:call-template name="parse-standard-stuff">
    <xsl:with-param name="child" select="$sequence"/>
    <xsl:with-param name="name" select="@name"/>
    <xsl:with-param name="suppressJoinFailure"
                    select="@suppressJoinFailure"/>
    <xsl:with-param name="joinCondition" select="@joinCondition"/>
  </xsl:call-template>

  <xsl:if test="java:add($parentHaschild, $parent, $sequence)"/>
  <xsl:variable name="sequenceHasactivity"
                select="java:getSequenceHasActivity($myBpel)"/>

  <xsl:apply-templates select="*">
    <xsl:with-param name="parent" select="$sequence"/>
    <xsl:with-param name="parentHaschild"
                    select="$sequenceHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!-- ========================================================== -->
<!-- parse <process> -->
<xsl:template match="bpws:process">
  <xsl:variable name="processClass"
                select="java:getProcess($myBpel)"/>
  <xsl:variable name="process"
                select="java:createProcess($processClass, @name,
                        @targetNamespace)"/>

  <xsl:apply-templates select="bpws:variables |
                               bpws:partners |
                               bpws:partnerLinks">
    <xsl:with-param name="parent" select="$process"/>
  </xsl:apply-templates>

  <xsl:variable name="processHasactivity"
                select="java:getProcessHasActivity($myBpel)"/>

  <xsl:apply-templates select="bpws:receive   |
                               bpws:reply     |
                               bpws:invoke    |
                               bpws:throw     |
                               bpws:wait      |
                               bpws:empty     |
                               bpws:sequence  |
                               bpws:switch    |
                               bpws:while     |
                               bpws:pick      |
                               bpws:flow      |
                               bpws:scope     |
                               bpws:terminate |
                               bpws:assign">
    <xsl:with-param name="parent" select="$process"/>
    <xsl:with-param name="parentHaschild" select="$processHasactivity"/>
  </xsl:apply-templates>
</xsl:template>
<!-- ========================================================== -->
<xsl:template match="/">
  <xsl:apply-templates select="/bpws:process"/>
</xsl:template>

</xsl:stylesheet>
```

List of Figures

References

[ACTI05] ActiveBPEL, LLC, Active Endpoints, “ActiveBPEL”, Link, Version 1.0.6, 2005, http://www.activebpel.org/index.html

[AGKR] A. Gerber, K. Raymond, DSTC, University of Queensland “MOF to EMF: There and Back Again”, Proc. Eclipse Technology Exchange Workshop, OOPSLA 2003, http://www.dstc.edu.au/Research/Projects/Pegamento/publications/MOF-to-EMF-There-and-Back-Again.pdf

[AGM05] S. W. Ambler, “Agile Modeling”, Link, 2005, http://www.agilemodeling.com/

[AGRA03] A. Agrawal, “Graph Rewriting and Transformation (GReAT): A Solution for the Model Integrated Computing (MIC) Bottleneck”, 2003, 18th IEEE International Conference on Automated Software Engineering, Montreal

[AGRA04] A. Agrawal, “GreAT: Graph Rewriting and Transformation”, Presentation, 2004, http://www.cis.uab.edu/softcom/seminar/spring04/linpresentation.ppt

[ANDR04] AndroMDA, “AndroMDA”, Link, Version 2.1.2, 2004, http://www.andromda.org/

[ARC04] Interactive Objects, “ArcStyer”, Link, Version 4.0, 2004, http://www.arcstyler.com/products/arcstyler_overview.jsp

[ATL04] INRIA Atlas, Université de Nantes, “The ATL Homepage”, Link, 2004, http://www.sciences.univ-nantes.fr/lina/atl/

[ASTT03] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen, “Response to the MOF 2. Query / View / Transformations RFP”, Version 1.0, 2003, http://www.omg.org/docs/ad/03-08-05.pdf

[BERN03] P. Bernstein, Microsoft Research, “Applying Model Management to Classical Meta Data Problems”, Proceeding, CIDR 2003, pp. 209-220, http://research.microsoft.com/users/philbe/PBernsteinCIDR12ext.pdf

[BKKR03] M. Bernauer, G. Kappel, G. Kramler, W. Retschitzegger, “Specification of Interorganizational Workflows - A Comparison of Approaches”, Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003), July 2003, Orlando, USA, pp. 30-36, [ISBN 980-6560-01-9], http://www.big.tuwien.ac.at/research/publications/2003/0603.pdf

[BOTL] Institut für Informatik, TU München, “The BOTL Tool”, Link, 2005, http://www4.in.tum.de/~marschal/botl/index.htm

[BPWS] IBM alphaWorks, “BPWS4J”, Link, 2004, http://alphaworks.ibm.com/tech/bpws4j

[BPEL03] BEA, IBM, Microsoft, SAP, Siebel, “Business Process Execution Language for Web Services Specificaion”, Version 1.1, 2003, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[BROO75] F.Brooks, “The Mythical Man-Month”, 1975, Book

[BTREE] Wikipedia, “B-Tree”, Link, http://en.wikipedia.org/wiki/B-Tree

[CLE01] J.C. Cleaveland, “Program Generators with XML and Java”, 2001, Prentice-Hall, http://www.craigc.com/pg/

[COLL03] Collaxa, “Collaxa’s Tutorial on BPEL4WS”, 2003, http://www.jcollaxa.com/

[CWM03] OMG, “Common Warehouse Metamodel (CWM) Specification”, 2003, Version 1.1, http://www.omg.org/docs/formal/03-03-02.pdf

[CZAR03] K. Czarnecki, S.Helsen, “Classification of Model transformation Approaches”, 2003, OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture

[DIA03] Gentleware, DaimlerChrysler, Telelogic, Adaptive, Rational, Sun, “UML 2.0 Diagram Interchange”, Submission in Response to OMG Document ad/2002-12-20, Version 1.0, 2003

[DIC03] DSTC, IBM, CBOP, “MOF Query / Views / Transformations”, Second Revised Submission, 2004, http://www.dstc.edu.au/Research/Projects/Pegamento/publications/ad-04-01-06.pdf

[DGR03] K. Duddy, A. Gerber, K. Raymond, Pegamento Project, DSTC, “Eclipse Modeling Framework (EMF) import/export from MOF / JMI”, Status Report, 2003

[EMF04] Eclipse Project, “Eclipse Modeling Framework”, Link, Version 2.1.0, 2004, http://www.eclipse.org/emf/

[ETTK04] IBM Alphaworks, “Emerging Technologies Toolkit”, Software Development Kit, Version 2.1, 2004, http://www.alphaworks.ibm.com/tech/ettk

[FRAN03] D. Frankel, “Model Driven Architecture: Applying MDA to Enterprise Computing”, 2003, published by John Wiley & Sons

[GAR03A] T. Gardner, IBM, “UML Modelling of Automated Business Processes with a Mapping to BPEL4WS”, 2003

[GAR03B] T. Gardner, IBM, “Mapping from UML to the Business Process Execution Language for Web Services (BPEL4WS)”, Presentation, 2003, OMG MDA Implementer’s Workshop

[GENT] Gentleware, “Poseidon for UML”, Link, http://www.gentleware.com/

[GMT04] Eclipse GMT, “Generative Model Transformer”, Link, 2004, http://www.eclipse.org/gmt/

[IBM03] J. Amsden, T. Gardner, C. Griffin, S. Iyengar, J. Knapman, IBM, “Draft UML 1.4 Profile for Automated Business Processes with a Mapping to BPEL 1.0“, 2003

[IKKB04] I. Kurtev, K. Van den Berg, “Unifying Approach of Model Transformations in the MOF Metamodelling Architecture”, 2004, http://wwwhome.cs.utwente.nl/~kurtev/files/MDAIA04.pdf

[INR04] INRIA, Université de Nantes, “Model Transformation at INRIA ModelWare”, Link, 2004, http://modelware.inria.fr/

[IYEN03] S. Iyengar, IBM, “Implementing Model Driven Web Services Architecture using UML, XML, WSDL & BPEL4WS”, Presentation, 2003

[JAXB] Sun Microsystems, “Java Architecture for XML Binding (JAXB)”, Link, http://java.sun.com/xml/jaxb/index.jsp

[JBRL97] J. Bézivin, R. Lemesle, “Ontology-Based Layered Semantics for Precise OA&D Modeling“, Workshop Reader, ECOOP'97

[JET04] R. Popma, “JET Tutorial Part 1 (Introduction to JET)“, „JET Tutorial Part 2 (Write Code that Writes Code)“, Link, 2004, http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html

[JMI02] JSR 040, Java Community Process, “Java Metadata Interface (JMI) Specification”, Version 1.0, 2002, http://www.jcp.org/

[KENT] University of Kent, “KMF - Kent Modelling Framework”, Link, http://www.cs.kent.ac.uk/projects/kmf/index.html

[MART02] A. Martens, “Business Process Execution Language – Begriffe Zusammenhänge, Unklarheiten”, Presentation, 2002, http://www.informatik.hu-berlin.de/top/forschung/projekte/vgp_mit_ws/bpel/download/axel-02-12-13_6.pdf

[MDR03] Netbeans, “Netbeans Metadata Repository - MDR”, Link, 2003, http://mdr.netbeans.org

[MELL05] S. J. Mellor, “Agile MDA”, 2005, http://www.omg.org/agile

[MMGK04] M. Murzek, G. Kramler, “Defining Model Transformations for Business Process Models Graphically”, Proceedings at the Workshop "Enterprise Modelling and Ontology: Ingredients for interoperability" at PAKM2004, 2004, http://www.big.tuwien.ac.at/research/publications/2004/0904.pdf

[MOF02] OMG, “Meta Object Facility (MOF) Specification”, Version 1.4, 2002, http://www.omg.org/docs/formal/02-04-03.pdf

[MRB03] S. Melnik, E. Rahm, P. Bernstein, "Rondo: A Programming Platform for Generic Model Management," Proceeding, SIGMOD 2003, pp. 193-204, http://research.microsoft.com/users/philbe/RondoSIGMOD03.pdf

[MTF04] IBM Alphaworks, “Model Transformation Framework”, Version 1.0, 2004, http://www.alphaworks.ibm.com/tech/mtf

[MTL04] INRIA Triskell, Université de Nantes, “MTL engine”, Link, 2004, http://modelware.inria.fr/rubrique.php3?id_rubrique=8

[NBIDE] Netbeans IDE, “NetBeans”, Link, Version 4.0, 2004, http://www.netbeans.org/

[NBUM] Netbeans, “UML Profile for MOF”, Link, 2003, http://mdr.netbeans.org/uml2mof/profile.html

[OCL20] OMG, “UML 2.0 OCL Specification”, 2003, Final Adopted Specification, http://www.omg.org/docs/ptc/03-10-14.pdf

[OPEN] OpenMDX MDA platform, “openMDX”, Link, Version 1.6.2, 2005, http://www.openmdx.org/

[OPTI04] Compuware Corporation, “OptimalJ”, Link, Version 3.2, 2004, http://www.compuware.com/products/optimalj/

[PBG01] M. Peltier, J. Bezivin, G. Guillaume, “MTRANS: A general framework, based on XSLT, for model transformations”, WTUML: Workshop on Transformations in UML, 2001, http://ase.arc.nasa.gov/wtuml01/submissions/peltier-bezivin-guillaume.pdf

[QVTM04] QVT-Merge Group, “Revised Submission for MOF 2.0 Query / View / Transformations RFP”, Version 1.8, 2004, OMG document ad/2004-10-04

[QVTR02] OMG, “MOF 2.0 Query / View / Transformations RFP”, RFP, 2002, http://www.omg.org/docs/ad/02-04-10.pdf

[QSUB03] QVT-Partners, “Revised Submission for MOF 2.0 Query / View / Transformations RFP”, Version 1.1, 2003, http://www.qvtp.org/downloads/1.1/qvtpartners1.1.pdf

[QPAR04] QVT-Partners, “QVT-Partners”, Link, 2004, http://www.qvtp.org

[VELO04] The Apache Jakarta Project, “Velocity”, Link, 2004, Version 1.4, http://jakarta.apache.org/velocity/

[RINN03] D. Rinner, “Transformation of UML to WSDL/BPEL4WS”, Diploma Thesis, Information Systems Institute - Database and Artificial Intelligence Group, Technical University of Vienna, 2003

[SABL] E. Gagnon et al., “SableCC Java Parser Generator”, Link, Version 2.18.2, http://sablecc.org/

[SOAP03] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, H.F Nielsen, W3C, “SOAP Version 1.2 Part 1: Messaging Framework”, W3C Recommendation, Version 1.2, 2003, http://www.w3.org/TR/soap12-part1/

[SVG03] W3C, “Scalable Vector Graphics (SVG)”, Specification, Version 1.1, http://www.w3.org/TR/SVG/

[TEFKAT] DSTC, “The EMF Transformation Engine”, Link, http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/index.html

[U2MOF] Netbeans, “UML2MOF Tool”, Link, http://mdr.netbeans.org/uml2mof/

[UML03] OMG, “Unified Modeling Language Specification” Version 1.5, 2003, http://www.omg.org/uml/

[UMT04] UML Model Transformation Tool, “UMT-QVT Homepage”, Link, Version 0.81, 2004, http://umt-qvt.sourceforge.net/

[UPMF04] OMG, “UML Profile for Metaobject Facility (MOF) Specification”, Version 1.0, 2004, http://www.omg.org/docs/formal/04-02-06.pdf

[USVG] S. Dumitriu, M. Gîrdea, C. Hrițcu, “uml2svg”, Version 0.12, 2005, http://uml2svg.sourceforge.net/

[VAPA02] D. Varró, A. Pataricza et al., “VIATRA – Visual Automated Transformations for Formal Verification and Validation of UML Models (Tool demonstration)”, 2002, http://www.inf.mit.bme.hu/~varro/publication/ase2002_varro.pdf

[WSDL01] W3C, Ariba, IBM, Microsoft, “Web Services Description Language”, W3C Note, Version 1.1, 2001, http://www.w3.org/TR/wsdl

[WSFL01] F. Leymann, IBM, “Web Services Flow Language” Version 1.0, 2001, http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[XALA04] Apache XML Project, “Xalan-Java” Link, 2004, http://xml.apache.org/xalan-j/

[XLAN01] S. Thatte, Microsoft, “XLANG” Version 1.0, 2001, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[XMI03] OMG, “XML Metadata Interchange (XMI) Specification” Version 2.0, 2003, http://www.omg.org/technology/documents/formal/xmi.htm

[XSCH04] W3C, “XML Schema” Version 1.0, 2004, http://www.omg.org/technology/documents/formal/xmi.htm

[XSLT99] W3C, “XSL Transformations (XSLT)”, Recommendation, Version 1.0, 1999, http://www.w3.org/TR/xslt

Name:	Thomas Reiter
Geburtsdatum:	7.2.1978
Geburtsort:	Ried im Innkreis
Schule:	1988 – 1992 Hauptschule Obernberg am Inn
	1992 - 1997 HTL Braunau, Zweig Nachrichtentechnik, Matura im Juli 1997
Studium:	1998 – 2005 Studium der Informatik an der JKU Linz
November 2004	Abschluß des Bakkalaureats Informatik an der JKU Linz
März 2005	Voraussichtlicher Abschluß des Magisterstudiums Informatik an der JKU Linz
Sommersemester 2002	Studentenaustausch mit der University of Adelaide
Juli – Dezember 2003	Studienaufenthalt im Zuge der Diplomarbeit an der University of Adelaide
Weitere Tätigkeiten:	1997 - 1998 Zivildienst Alten- und Pflegeheim Ried im Innkreis
	Juli - August 1996 Entwicklungshilfe in Nicaragua durch die Schulpartnerschaft HTL Braunau – Polytecnico La Salle, Leon

Besondere Kenntnisse:

Sprachen	Englisch, Spanisch, Mandarin
Programmierkenntnisse	Java, C#, C++, SQL, Chipkartenprogrammierung, UML, etc.

Linz, 22.02.2005