On developing a multi-model repository

Ammerlaan, B.

Award date:
2009

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

On developing a multi-model repository

Boris Ammerlaan

Supervisor TU/e
Marc Voorhoeve

Supervisors GloMidCo
Marcel Grauwen
Harco Smit

Committee members
Tom Verhoeff
Ruurd Kuiper

August 18, 2009

Abstract

In February 2007 I came into contact with Global Middleware Consultancy. They wanted to develop a tool for the integration of the models that are used by middleware implementations such as Tibco Rendez-vous or SonicMQ. Some previous work in this direction had been done by a a student resulting in a general framework, and they needed another student to perform one of several possible next steps.

The proposed framework mainly concerned the technical infrastructure. In this thesis, its functionality is designed as a repository allowing the storage and maintenance of a large variety of models and views, allowing for arbitrary consistency checks. The designed repository tool allows the addition of new types of models and consistency rules with little effort.

Acknowledgments

It has taken a lot of research, hard work and perseverance to get to this point, but I am pleased with the result. Of course, I could never have done all of it alone and I would like to thank the following people:

My friends, who always made it worth it. They should know who they are, but I would particularly like to mention Ad, Angelo, Hennes, Jan Willem, Johan and Stefan. Dean and Marcel have also been invaluable.

Marcel Grauwen (GloMidCo) and Marc Voorhoeve (Eindhoven University of Technology) for their assistance in this research.

Harco Smit and Dick van den Broek (GloMidCo) for assisting in the evaluation of the previous research.

Evert, Kirsten, Lonneke, Renée, Ties and all the others for keeping me going even when I didn't want to.

Andre, Bill, Kenneth, Laila, Michael, Natasja, Patrick, Paul A., Paul D., Rebecca and Surani for helping and inspiring me all these years.

Anyone else who has helped me over the years. Even though your name was not included, your help was still appreciated.

Boris Ammerlaan, August 2009

Contents

1 Introduction **6**
- 1.1 Background . . . 6
  - 1.1.1 An example . . . 6
- 1.2 Motivation . . . 8
- 1.3 Goal . . . 9

2 Initial Research **10**
- 2.1 Modelling . . . 10
  - 2.1.1 IEEE1471 reference model . . . 10
  - 2.1.2 Metamodel construction . . . 11
- 2.2 Middleware . . . 12
  - 2.2.1 Service-Oriented Architecture, Enterprise Service Bus, etc. . . . 13
  - 2.2.2 Message-Oriented Middleware . . . 13
- 2.3 Conceptual framework & metamodels . . . 14
- 2.4 Eclipse . . . 16

3 System requirements **17**
- 3.1 Approach . . . 17
- 3.2 Scenario . . . 18
- 3.3 Usage . . . 19
  - 3.3.1 Modelling . . . 19
  - 3.3.2 Meta-modelling and rule maintenance . . . 19

4 Models and consistency examples **22**
- 4.1 UML Class Diagram . . . 22
  - 4.1.1 Abstract definition . . . 22
  - 4.1.2 Example . . . 24
  - 4.1.3 Informal description of visual elements . . . 25
- 4.2 UML State Machine Diagram . . . 27
  - 4.2.1 Abstract definition . . . 27
  - 4.2.2 Example . . . 28
- 4.3 UML Activity Diagram . . . 28
  - 4.3.1 Abstract definition . . . 28
  - 4.3.2 Informal description of visual elements . . . 30
- 4.4 UML Sequence Diagram . . . 31
  - 4.4.1 Abstract definition . . . 32
  - 4.4.2 Example . . . 32
  - 4.4.3 Informal description of visual elements . . . 33

5 Examples of inter-model consistency **34**
5.1 USD / UCD 34
5.1.1 Example 34
5.1.2 In General 34
5.1.3 Formal rules 35
5.2 USD / USMD 35
5.2.1 Example 35

6 Conclusions **37**
6.1 Summary 37
6.2 Recommendations 38

A List of References **39**

B List of Definitions **41**

List of Figures

1.1 Employment lifecycle in the HR department 7
1.2 Employment lifecycle in the IT department 7
1.3 Enterprise communication involving multiple buses 8

2.1 The IEEE1471 reference model 11
2.2 OSI and TCP/IP layers 12
2.3 Conceptual framework 14
2.4 MOM message flow metamodel 15
2.5 Tool environment 15

3.1 Navigation between models 18
3.2 Typical usage by a modeller 19
3.3 Typical system maintenance 20

4.1 Example UML Class Diagram 24
4.2 Example UML State Diagram 29
4.3 Example UML Activity Diagram 29
4.4 Example UML Sequence Diagram 31

5.1 Example USD/UCD 34
5.2 Example USD/USMD: USD 35
5.3 Example USD/USMD: USMD 36

Chapter 1

Introduction

1.1 Background

Enterprises (and organisations in general) in present-day society are supporting a large number of processes in order to fulfill their goals. These processes have a tendency to become evermore complex and interconnected. Existing processes are becoming components of new, integrated processes. Integrating these components requires a good understanding of their purpose, design and implementation, described by means of several models. A wide range of modelling techniques makes this integration quite a challenge, since it is nearly impossible for any one modeller to fully understand all these techniques. This necessitates the development of new tools for the comparison and integration of models. A necessary first step is developing a model repository that allows the integration of a consistent set of models.

1.1.1 An example

Figures 1.1 and 1.2 show a model that describes the stages an employee goes through.

Figure 1.1 shows a Petri net describing an employee's life cycle within the Human Resources department. Initially, a new employee is hired and the relevant data is registered, such as his name, social security number, and the temporary or permanent nature of the employment. During the employment, changes can occur. Examples of changes are extending the period of employment, making the employment permanent, or even something else. At some point, the employment ends, e.g. because the contract period has expired. The salary payment is stopped and the end of the employment is registered.

Figure 1.2 shows another Petri net that describes the process for the IT department. In this scenario, the focus is towards the employee's computer account. At some point, the IT department is informed about a new employee needing an account. Depending on the employment's temporary or permanent nature, the account created is temporary or permanent as well. A temporary account will be extended (and remain temporary), automatically expire, or be made permanent. If it expires it will first be disabled and eventually deleted. Permanent accounts must be dismissed explicitly. Temporary accounts may be extended or moved to a permanent one.

As Figures 1.1, 1.2 and 1.3 show, the HR department and the IT department do not interact. The various processes used at the HR department and the IT department are supported by separate "software buses". The software components used by financial officers, operators, etc. are connected to a bus which handles

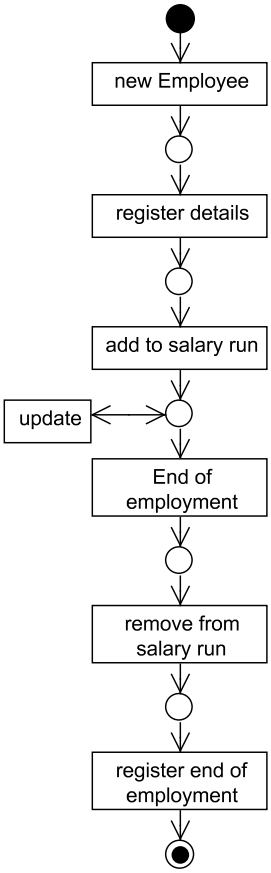

Figure 1.1: Employment lifecycle in the HR department

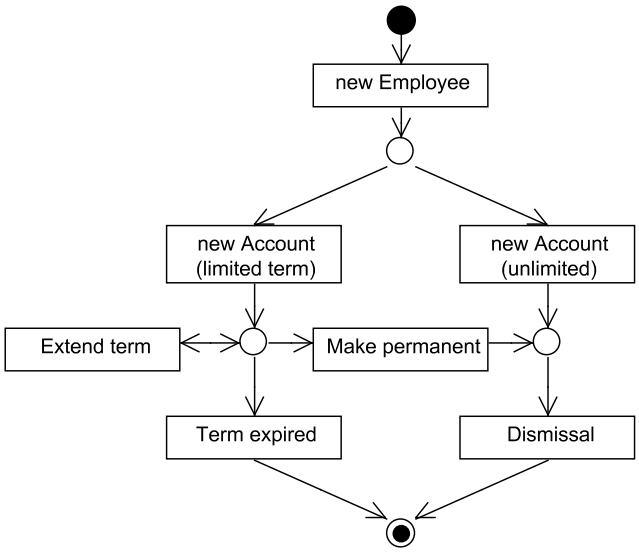

Figure 1.2: Employment lifecycle in the IT department

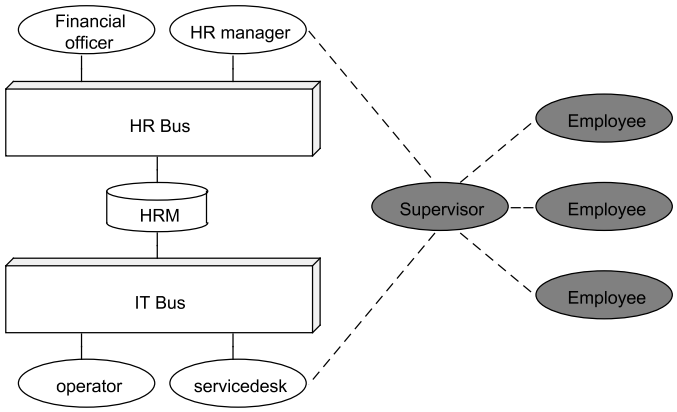

Figure 1.3: Enterprise communication involving multiple buses

communication with other components. Both buses can read data from the HRM database, but there is no direct communication between the departments.

The employee's boss needs to initiate both processes and to effect non-automatic actions. (I.e., "update" in Figure 1.1 and "Extend term" and "Make permanent" in Figure 1.2.) If he does not ensure that both processes are properly synchronised, the employee may end up without an account or with one, but without employment. Only when these processes are fully integrated can synchronisation be enforced. This integration could be done by identifying which elements are common to both diagrams and which are not. Common elements can be identified either by matching names (e.g. "new Employee" in both diagrams) or manually linking them ("End of employment" in 1.1 and "Term expired"/"Dismissal" in 1.2).

There is a need for the integration of different models and for the generation of different views on the system. A view is a subset of a model that takes certain concerns into account. The visual presentation of that subset is a diagram. If a formal method is applied to construct a diagram, that method uses an appropriate diagram definition.

The component models in Figures 1.1 and 1.2 were described by the same modelling technique (Petri nets). This allows us to interpret and analyse the integrated model as a Petri net as well, insofar as integrating them is at all possible. If different modelling techniques had been used, integrating the models and interpreting the integrated model would have been much harder.

Integrating processes is usually done using middleware, a software component which enables the decoupling of information providers and information consumers. Middleware often uses a so-called Enterprise Service Bus, which not only loosely connects providers and consumers, but may also perform other tasks like data transformation.

Traditional modelling techniques do not provide for this integration. Dick van den Broek has done research for Global Middleware Consultancy into the integration of different models which was recorded in [BRO07]. This thesis is a continuation of that research.

1.2 Motivation

In an enterprise environment, communication is often performed through a communication *bus*. This is illustrated by the clients connected to the different buses

in Figure 1.3. More than one bus may have been used because each bus was better suited for a specific task, but far more likely they were previously deployed separately from each other and later needed to be integrated into a coherent whole. A bus may be connected to another bus as a client or through the use of a bridge, which functionally merges them into one bus.

A *client* that needs information (a *consumer*) can request it from the bus, and a client producing information (a *producer*) can provide it to the bus. No client needs to know about any others, as all routing of information and scheduling of processes is done by the bus. In some cases, an *adapter* connecting the client to the bus performs the necessary data transformation between the client's data format and one of the data formats supported by the bus.

Several implementations of enterprise buses exist. Each one has its own modelling tools, and in most cases the models are tightly coupled with the implementation of the process; when a model of a bus is changed, the behaviour of that bus is altered as well.

A bus is optimised for a specific task. The integration of buses interconnecting their tasks is rather complex, since it often requires profound knowledge of their proprietary modelling techniques. A necessary first step towards such an integration is the combination of the relevant models into a single repository, allowing consistency checks.

1.3 Goal

In order to solve some of the problems mentioned in the previous paragraphs, an open source modelling tool needs to be developed that offers features as described in [BRO07]:

- Storage of all models in a common repository;
- Navigation between different models using common elements. An example of such a common element would be *HRM* in Figure 1.3. Given a separate model for each bus (with its attached clients), an architect viewing the model for *HR Bus* could select *HRM* and discover it is also part of the model for *IT Bus*, which he might then open as well;
- If an element is common to more than one model, any changes to it in one model are immediately visible in all the other models;
- Different aspects of the same model can be shown according to the wishes of the modeller; and
- Connecting model elements and their middleware implementation.

Such a tool, rather than a collection of different proprietary tools, would set a standard for interoperability. The integration of different models ensures a better overview of the entire system. Eliminating data duplication, but still being able to look at the same models in different ways, will enable anyone to keep using the techniques he is most comfortable with.

Chapter 2

Initial Research

[BRO07] contains extensive research into the field of message-oriented middleware (MOM, see subsection 2.2.2). It presents a general framework (described in section 2.3) to model situations common to MOM, as well as some specific examples of models for some of those situations. Since this framework and the examples had to be evaluated, there was a need to do some research into middleware and related subjects.

In order to enable understanding of the framework from [BRO07], a few concepts are treated first. Section 2.1 explains two subjects: the [IEEE1471] reference model (which [BRO07] refers to for the use of the concepts *View*, *Viewpoint*, *Library Viewpoint* and *Model*) and the [MOF] meta-model construction method (which [BRO07] refers to in the design of the (meta-)modelling tools). Section 2.2 explains several terms related to the concept of middleware, since the framework is intended to support modelling in a middleware environment. Section 2.3 summarises the framework and section 2.4 explains why Eclipse should be used for the implementation.

2.1 Modelling

2.1.1 IEEE1471 reference model

[IEEE1471] describes twelve terms related to an architectural description. They can be roughly divided into two categories: terms representing the system (Mission, Environment, System, Architecture, Stakeholder and Concern) and terms used to document the architecture of the system (Stakeholder, Concern, Architectural Description, Rationale, Viewpoint, Library Viewpoint, View and Model).

An *Architecture* is the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

A *Stakeholder* is any person or user with an interest in the modelled System. Examples: a manager, a systems architect, a database designer, etcetera. Usually, these interests are represented by *Concern*s. (I.e., interests pertaining to the system.)

At the centre of any description of a modelled system is the *Architectural Description* (AD). It is the collection of documents describing the system's that are used to document the architecture.

An AD contains a large number of *Models*, describing certain aspects according to a predefined graphical syntax. Apart from models, it may contain other documents that are less formal.

The AD should address the concerns of its stakeholders. A *Viewpoint* is a function that projects an AD, highlighting a few concerns for a few stakeholders.

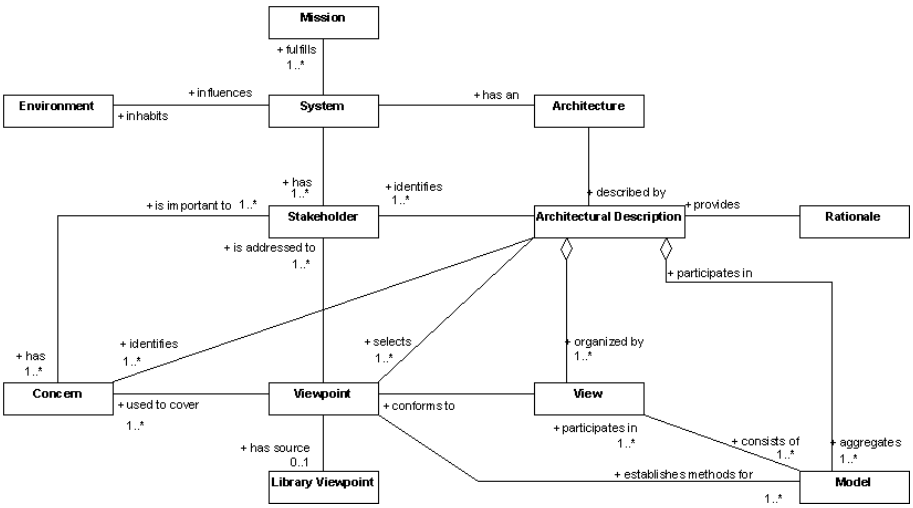

Figure 2.1: The IEEE1471 reference model

Viewpoints can be dedicated to the current AD or predefined (*Library Viewpoints*). The result obtained by applying a viewpoint onto the AD is called *View*. A view is related to part of the AD's models.

As an example, take the description of the employee lifecycle in chapter 1 as presented in Figures 1.1 and 1.2 and the accompanying text. Both diagrams together can be considered the AD. Separately, they are models. Figure 1.2 is intended for stakeholders like a system operator, a servicedesk employee or an IT manager. Their main concern is the continued and correct operation of the computer systems. A servicedesk employee may also be concerned about explaining any procedures to the end user. The previous explanation of the stakeholders, their concerns and why which part of the AD was chosen together form the viewpoint. This specific collection of models, views, concerns, viewpoints and stakeholders describe (an aspect of) the AD together with the *Rationale*. The AD describes how a new employee is treated (the *Architecture*) in a certain company (the *System*) by the different departments. What sector the company operates in (the *Environment*) and its purpose (or *Mission*) are not important at the moment.

2.1.2 Metamodel construction

The concept of meta-modeling is well-known and applied both within middleware and in our research. A meta-model is a model to describe other models. In [MOF00], it is stated that four levels of data are conventionally described as follows:

> The user object layer is comprised of the information that we wish to describe. This information is typically referred to as "data."
>
> The model layer is comprised of the meta-data that describes information. Meta-data is informally aggregated as models.
>
> The meta-model layer is comprised of the descriptions (i.e., meta-meta-data) that define the structure and semantics of meta-data. Meta-meta-data is informally aggregated as meta-models. A meta-model can also be thought of as a "language" for describing different kinds of data.
>
> The meta-meta-model layer is comprised of the description of the structure and semantics of meta-meta-data. In other words, it is the "language" for defining different kinds of meta-data.

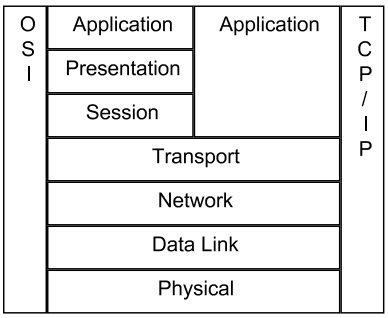

Figure 2.2: OSI and TCP/IP layers

These layers are often referred to as M0 - M3. It is possible to define more "meta" layers in an application; however in most cases these four are sufficient, as any extra layer would be a special application of layer M3. As an example these layers are applied to the messages in Message-Oriented Middleware. This results in the following layers, starting from the bottom:

At the user object layer (M0) there are the actual messages that move through the enterprise system. A message, for example, can contain a specific order: "John, 1, chips, 07/18/06".

In the model layer (M1) the order message is described. The order message "John, 1, chips, 07/18/06" can be described by name, quantity, product and date of the order. In this layer the message containing an order is described in an abstract form. Models are common practice in the design, development and management of IT systems. The problem is that there is not one single description for the same message. There are many techniques to describe the same message. This can lead to confusion about the message, especially if different methods are used to describe the descriptions of messages.

The next layer (M2) describes the description of the message.

The last layer (M3) describes the meta-model.

These abstract layers can be applied in different situations and from different angles. In the above example, it is applied to messages, but it could also be applied to middleware process engines.

2.2 Middleware

In [LIN04, page 116], middleware is described as "*any type of software that facilitates communication between two or more software systems.*". Other authors use similarly broad definitions, mostly to accommodate all the different types of middleware. (E.g. Remote Procedure Call (RPC), Transaction-Oriented, Message-Oriented, etc.) The implied assumptions are that the software's primary goal was to make communication easier and that it is *reusable*. Although ad-hoc communication protocols are covered by the first assumption, they fall short of the second assumption.

According to [BRO07, page 12], "*Middleware is implemented as a separate layer between the application and the network interface, thus providing a new platform that facilitates the interaction between applications.*" In [KRA04, see Figure 2.2] a similar definition is used, placing the middleware framework between applications and the network protocol. Considering the OSI reference model, one could consider middleware as an implementation of the Transport and Session layers, with some added functionality in the Presentation layer.

2.2.1 Service-Oriented Architecture, Enterprise Service Bus, etc.

In [CHA04, page 57], a Service Oriented Architecture (SOA) is described as "*a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation.*".

Even these days, much communication in distributed computing is done by use of ad-hoc or proprietary (usually point-to-point) protocols. Two objects might communicate using some protocol, but if an extra object is added to the mix, there is a need to duplicate functionality from the first two and possibly modify their communication functions to make them aware of the new object. However, none of these objects might actually *need* to know about the others. Mostly, all that is needed is that some relevant data is sent and that possibly some other data is received. An Enterprise Service Bus (ESB) solves this dilemma by decoupling the communication channel and the communication of several objects over that channel.

2.2.2 Message-Oriented Middleware

Our research was aimed specifically at MOM. The communication provided by MOM is asynchronous and decoupled, providing independence among the interacting applications. An application can send a message without waiting for a direct response or confirmation. Thus it can move on; this is also known as non-blocking communication. In contrast, RPC middleware resembles a function call in a program, but realised by another function that is expected to execute on another computer node. The advantage of RPC is that it fits in with a procedural programming paradigm. The calling program, however, has to wait until the called function returns. This possibly takes a long time, considering the extra work needed to complete the request across the network.

The *transport system* is at the core of a MOM implementation. The transport is provided through the concept of *message channels*. The *applications* use a *programming interface* or an *adapter* to utilize the middleware channels. Applications *send a message* to a channel and applications *listen* to a channel. The channels are also referred to as *topics* or *subjects*. The sending applications are referred to as *publishers* and receiving applications as *subscribers* or *consumers*. The terminology may differ for each product implemented by the middleware suppliers.

The messaging system holds a message if it is not possible to deliver the message. This situation occurs, for example, if the network is temporarily down or the consumer is overloaded. Depending on the implementation, the messaging system offers different levels of reliability for each channel. (The highest level of reliability is the transactional message channel.)

A channel can be used for different forms of communication, depending on the number of publishers and consumers connected to the channel, or the type of messages on the channel. It is not necessary for the consumers on a channel to immediately consume a message.

"*The primary advantage of a message-based communications protocol lies in its ability to store, route or transform messages in the process of delivery.*" [URL01] This enables publish-subscribe communication. Strictly speaking, it also enables one-to-one communication, but this offers fewer advantages over point-to-point communication. Additionally, communicating processes need not use the same data format because the middleware adapter offers the possibility to transform data.

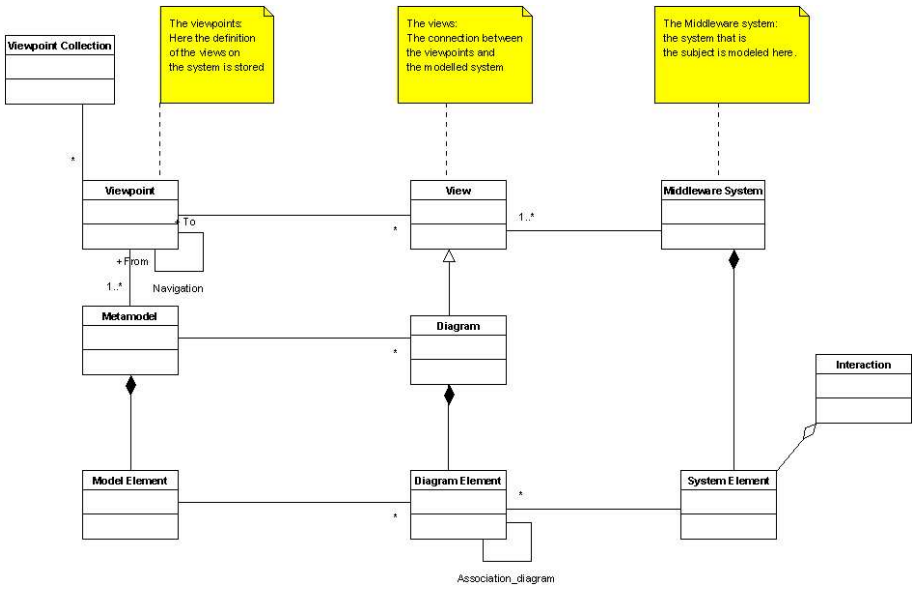

Figure 2.3: Conceptual framework

2.3 Conceptual framework & metamodels

Figure 2.3 illustrates the conceptual framework and metamodels from [BRO07]. An adjusted version of it was used early in the research when the main goal was still on producing a program. In determining the technical requirements it was decided to focus on determining validity of models, so it is only used to illustrate the long-term goal.

A *Model*, a *View*, and a *Viewpoint* are all the definitions from [IEEE1471] (see subsection 2.1.1 that were used in the design of the conceptual framework. This would seem to exclude *Library viewpoints*, but when considering the rest of [BRO07], it seems that all *Viewpoints* are predefined and therefore should be called *Library viewpoints*. Also – since a viewpoint (as described by [IEEE1471]) defines the translation from a model of a system to a view, the design needs to be adjusted slightly. This translation uses knowledge about the system, the model and the diagram type, so the *Architectural Description* should also be included. But since the AD is connected to all models and model elements, it is less useful to include in the framework as a separate entity. Strictly speaking, both *Concerns* and *Stakeholder*s should also be considered. For the moment, we will consider only one *Stakeholder* and can therefore (momentarily) disregard others.

One of the diagram metamodels that is defined in [BRO07] is shown in Figure 2.4. It *could* be used for modelling middleware but there is no guarantee that any model that uses it would actually be correct. A *Message Exchange* will rarely take place between two *Adapter*s since an *Adapter* offers connectivity between, for example, an *Application* and a *Channel*. Two *Database*s will also rarely communicate (but two *database management systems* might). Further refinement is necessary into a metamodel that allows for different types of *Message Exchange*s.

The model in Figure 2.5 was presented for the tool in [BRO07]. However, it is neither sufficient nor entirely correct:

- It illustrates *which* components are to be used, but not *why* or *how*.
- GMF is not a framework on top of EMF and GEF, but a framework to make the transition from an EMF model to a GEF-based diagram environment

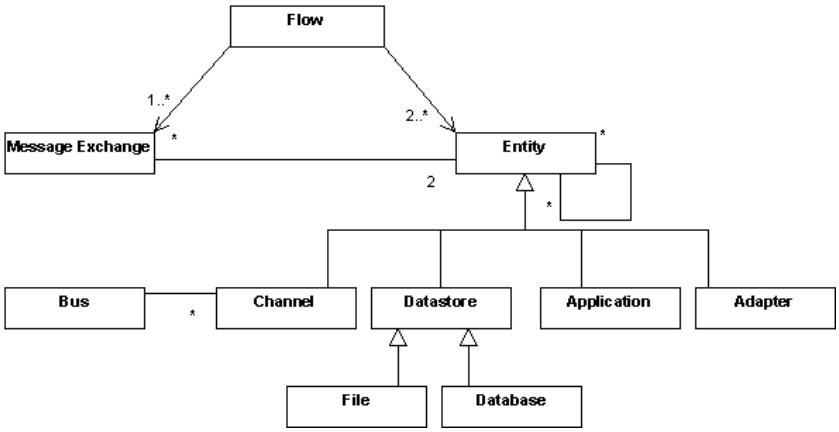

Figure 2.4: MOM message flow metamodel

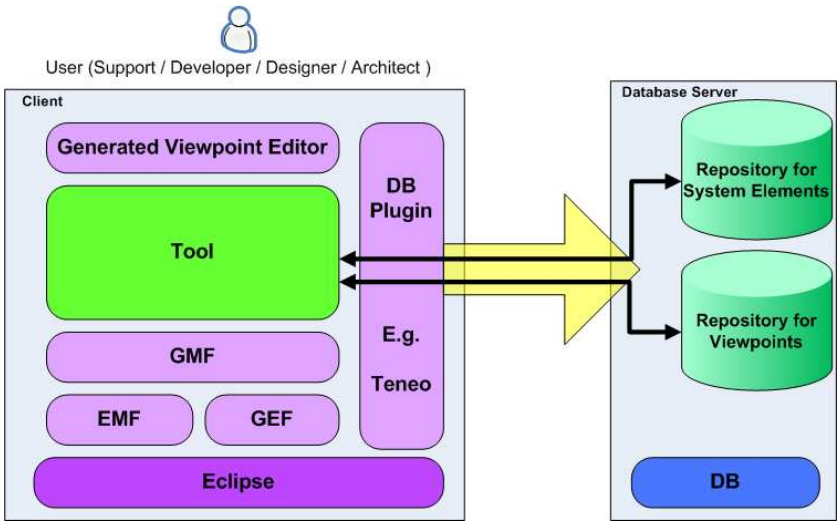

Figure 2.5: Tool environment

easier by using predefined figures.

- There should be at least two separate tools – one to add/modify/delete models using views, which will be used by the system architect, and one to add/modify/delete viewpoints, which will be used by the tool architect.

The design of the tool presented in Figure 2.5 is one from a technological viewpoint. It mentions several frameworks, but offers very little information about what the "Tool" part of the design is supposed to do.

In order to specify requirements for the tool presented in Figure 2.5, several assumptions were made:

- It should offer the ability to get an "overview" of the entire (modelled) system;
- It should offer the ability to navigate between different views of the same model.

In order to be able to do this, more research was required into what a model *is*, and how to determine whether two views overlap. This will be discussed in the following chapters.

2.4 Eclipse

[BRO07] recommends using Eclipse for the development of the tool. The advantages of using Eclipse as the platform to implement the framework in are:

- Eclipse was developed using Java, so it is largely platform independent.
- The "Eclipse Public License" is an open source license.
- Several plugins are available for Eclipse that make making graphical editors for models less complicated.
- Most major middleware vendors' offerings are implemented in Eclipse, or will be in the near future. Some of them have even formed alliances to make the modeling of their offerings more uniform. One example is the SCA specification by OASIS [SCA].

However, there is also a disadvantage:

- Because Eclipse is implemented in Java, the system requirements are relatively high.

Since we have total control over what we will implement and use, version conflicts will not occur. The system requirements are high, but not compared to those of the systems being modelled. Because no major disadvantages exist and some major advantages do exist, Eclipse is a reasonable choice, albeit a premature one.

The only aspects from the Eclipse platform that were considered are:

- its use as a Java Development IDE
- EMF, the *Eclipse Modeling Framework* (of which *Ecore* is a part)
- GEF, the *Graphical Editing Framework*
- GMF, the *Graphical Modeling Framework*

Chapter 3

System requirements

As outlined in the previous chapters, the tool should enable a consultant to not only add models, but new model types as well. A new model type should always be accompanied by an appropriate parser. Using XML as the data and meta-model format ensures that there are sufficient basic tools to manipulate and compare the models.

Another requirement is navigability. If you are viewing (or editing) a diagram, it should be possible to view overlapping models as well.

3.1 Approach

In [BRO07] a general framework for modelling middleware is presented, with some possible applications. To evaluate the framework, it is necessary to study the underlying concepts of modelling and middleware. Many tools already exist for modelling and middleware, and they needed to be evaluated as well. This evaluation consists of:

- Determining what the existing tools are able to model and whether the tool supports the requirements of the framework;
- Determining the differences, but especially the similarities, between the tools; and
- Determining which data formats are used and which is best suited for our purpose.

I have examined several tools and and their file formats. All of them are able to model basic UML models, but some of them disagree on the interpretation of specific model types. According to one tool's interpretation of Sequence Diagrams, no two messages can be sent or received at the same time; according to another's they can. Both also offer their own extensions that are not covered explicitly by the UML standard.

Most tools persist their models in some XML format. They store the logical elements of the models and the graphical presentation of those elements separately; it should be possible to compare parts of the logical structures automatically. An open format like [XMI] will ensure independence from all existing and future tools.

A first step toward the goals from section 1.3 is to determine what exactly constitutes a model.

It is also important to determine what is required to compare a model to another model, because navigation between models – i.e., the transition from one model to

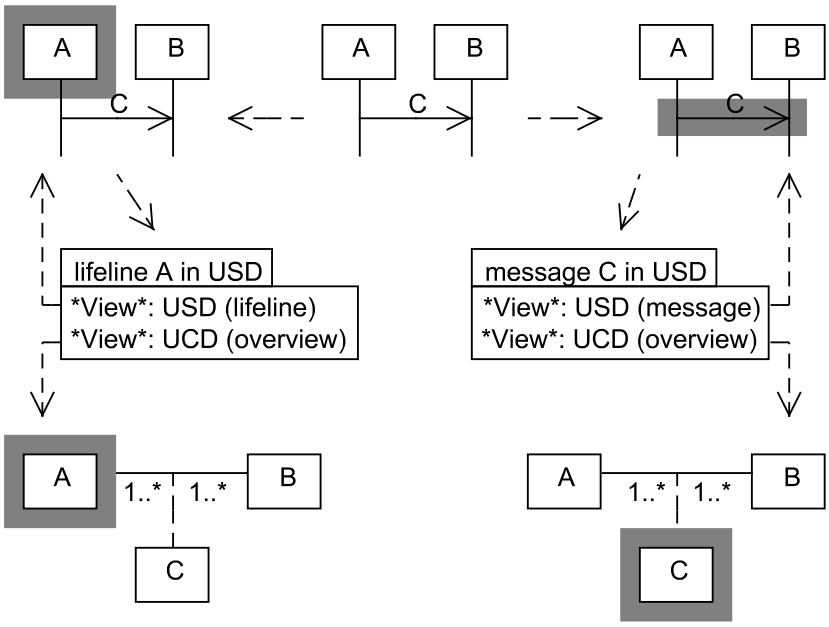

Figure 3.1: Navigation between models

another using a formal procedure – cannot be done without having some way to compare them.

As shown in figure 3.1, one way to navigate between models is to select an element in one model, request a list of all models this element appears in, and choosing another model from that list, which is then opened. The original element may (if possible) even be selected in the "new" model.

Finally, definitions for models and consistency are formulated, as are abstract definitions of specific model types and examples of (in)consistency between them.

3.2 Scenario

A typical example of a modeller using the tool will look like the following.

He starts his client for the system and he logs in. Logging in is necessary if there is a need to grant access to some models, but not to others and in order to track operations to users.

He selects the model from Figure 1.2, opens it using an external editor (like the UMLet editor or Gentleware's Poseidon) and deletes the step "new Account (unlimited)". He saves the change (thereby de-selecting it) and tells the system to check for inconsistencies.

It discovers that the modified diagram is inconsistent with the one from Figure 1.1, since the second place assumes a coloured token that can indicate a permanent contract. (As after the step "new Employee" in 1.1, a new employee has been registered for either a limited or an unlimited term. After the corresponding step in 1.2, a new employee could only have been registered for a limited term.) It reports this.

He opens that diagram and explicitly adds branches for both options. He creates a new diagram to model the employee's department and adds relationships from steps in the first two diagrams to steps in the new diagram.

He continues doing this until there are no inconsistencies left, or until he decides not to fix those that are left. At that time, he logs off.

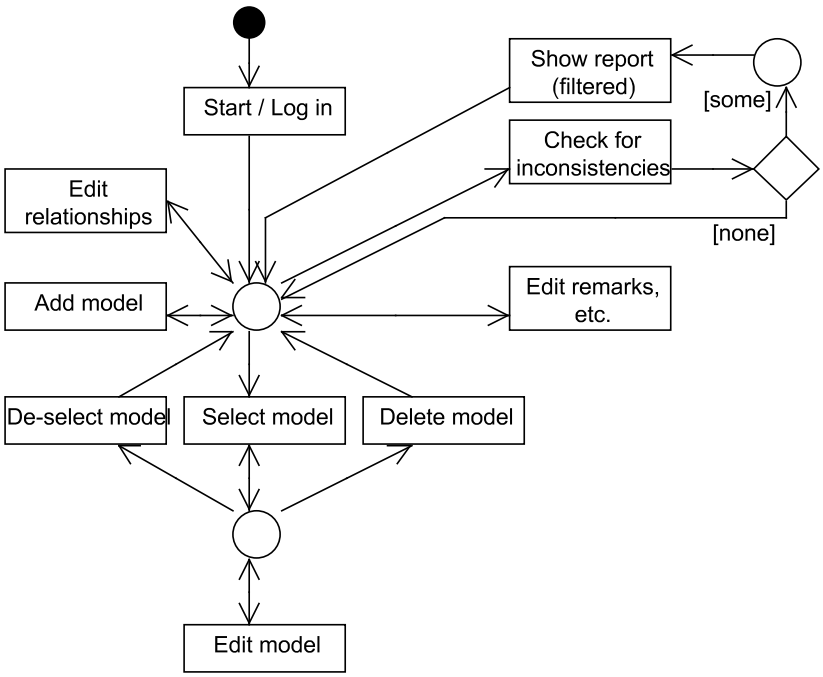

Figure 3.2: Typical usage by a modeller

3.3 Usage

3.3.1 Modelling

As Figure 3.2 shows, the tool should allow the adding, editing and deleting of models. Models can be related to each other. The tool can be exited at any time, even if any model or group of models is inconsistent. This is omitted from the Figure for simplicity. A group of related models can be automatically checked for consistencies - either internal to one model or between two or more models. When the system discovers inconsistencies in the models, it notifies the modeller. He can then decide to ignore them, let the system fix them (if at all possible), or fix them himself (possibly assisted by the system).

3.3.2 Meta-modelling and rule maintenance

Adding new model types should be possible as well, but this is done by the tool architect and not by the modeller. It consists of both defining a meta-model for the new type and creating a parser for that meta-model.

A tool can only decide whether models are demonstrably inconsistent by determining if certain formal rules apply. Therefore, there is a need to maintain these rules (as shown in Figure 3.3). A rule is a predicate on a set of models. This set can contain any number of models and the rule can be either formal (e.g., "All class names start with 'cls_'") or informal. Formal rules can be checked by the system, informal rules cannot.

Some example rules are:

- "In a class diagram, all class names must be unique."
- "In a state machine diagram, the end state should be reachable from any state."

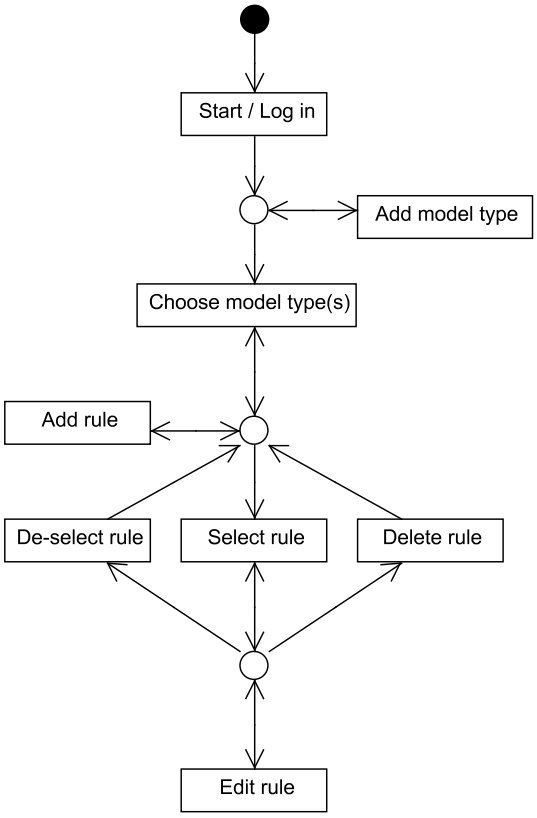

Figure 3.3: Typical system maintenance

- "In any model, company naming rules apply."
- "Given a USD and USMD pair, all messages in the USD will be sent and all states in the USMD can be reached, even when taking the other model's ordering into account."
  For example: if a USD contains the partial trace $(newemployee(temp), createaccount, deleteaccount)$, a related USMD will not contain the partial trace $(accountdeleted, newaccount, newuser)$.

Chapter 4

Models and consistency examples

The specified repository must be able to contain models of various types and allow the definition of rules for maintaining their consistency. In this chapter some examples of models (based on UML) and rules for "internal" consistency are given. The UML standard defines a cognitive semantics of models. In order to be able to define consistency rules, we provide objectivist semantics (which is less informal) of the treated models. One must bear in mind that these are example rules that do not pretend to be complete.

In the next chapter, examples of inter-model consistency rules will be given.

4.1 UML Class Diagram

A UML Class Diagram (UCD) is a tuple (C, A), where C is a set of classes, connected by associations A. In a set of instances (O, R) conforming to a UCD, objects correspond to classes and links correspond to associations.

An alternative formal semantics, including n-ary associations, is given in [SZL06].

Readers who are unfamiliar with UCDs, should read "Informal description of visual elements" first.

4.1.1 Abstract definition

Syntax

$$UCD = \{(C, A)\}$$

where

1. C is a finite set of *classes*.
2. A is a finite set of *associations*.
3. A *Class* is a triple (*l*: label, *P, M*) where
   (a) P is a set of *attributes*, and
   (b) M is a set of *operations*.
4. An *attribute* is a 4-tuple (*l*:label, *t*:type, *i*:initial value, *vt*:visibility type) where

   (a) t may be undefined, and
   (b) i may be undefined.

5. An *operation* is a 4-tuple (l:label, V, r:return type, vt:visibility type) where
   (a) V is a (possibly empty) list of arguments. An argument is a pair (l:label, t:type), and
   (b) r is optional and may be undefined.

6. An *Association* is a tuple (l:label, at, $From$, To, ac:class) where
   (a) $at \in \{basic, aggregation, composition, generalization\}$,
   (b) To and $From$ are 4-tuples (c:class, min:minimum, max:maximum, $role$:label), with $0 \leq min$ and either $max = *$ or $min \leq max$, and
   (c) ac (which is optional) is a class with the properties of the association.

Semantics

An UCD defines a set of states corresponding to the model. Each state corresponds to a set of *objects* and *links* that are connected through a set of *predicates*. An UCD $M = (C, A)$ corresponds to a set S of possible states, consisting of a set O of objects and a set R of relations:

$$S = \{(O, R) : R \subseteq O \times O \wedge PRED(C, A, O, R)\}$$

The predicate PRED(C,A,O,R) is the conjunction of the following requirements: (Only the first two requirements are formal.)

1. O is a set of objects $(id : uniqueidentifier, l', P', M')$; each object belongs to a class in C.
   $\langle \forall o \in O : \langle \exists c \in C : o[ofclass]c \rangle \rangle$
   The predicate o[ofclass]c, with c = (l, P, M), holds iff o = (id, l, P', M'), where P' is a set of pairs (lab, val) (i.e., variables) satisfying
   $\langle \forall p \in P : \langle \exists q \in P' : \pi_l(p) = \pi_{lab}(q) \wedge \Pi_{val}(q) \in \Pi_t(p) \rangle \rangle$

2. R is a set of object pairs; each pair belongs to an association in A.
   $\langle \forall (o, q) \in R : \langle \exists a \in A : (o, q)[isin]a \rangle \rangle$
   The predicate (o,q)[isin]a, with a = (l, at, $From$, To), holds iff
   $o[ofclass]\Pi_{From}(a) \wedge q[ofclass]\Pi_{To}(a)$

3. Instances of aggregation and composition associations connect exactly one "source".

4. The number of links conforms to the bounds as defined in the model. In other words: if classes A and B are connected by an x:y association, any particular instance of A will be connected by links to y instances of B (and any instance of B to x instances of A).

5. All class names must to be unique within the model.

6. All attribute names must be unique within the class or object.

7. All operation signatures (i.e. operation name plus typed arguments) must be unique in the class or object.

8. Aggregation (and composition) associations cannot be cyclic. (In other words, the transitive closure of the union of aggregation and composition is not reflexive.)

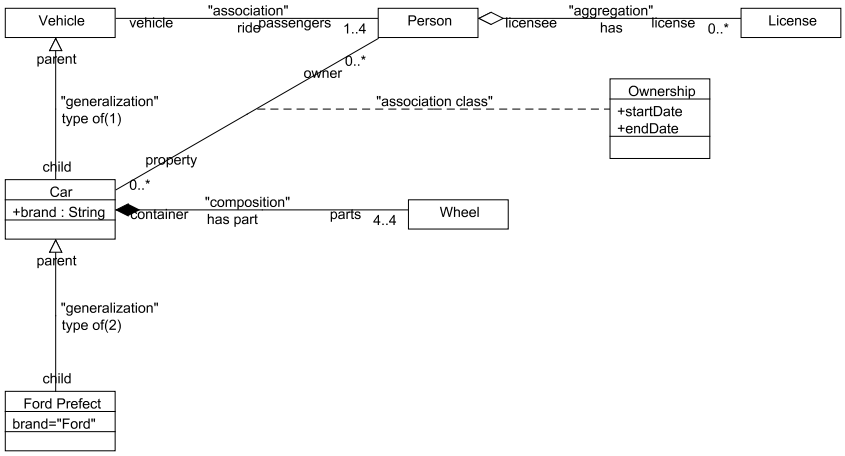

Figure 4.1: Example UML Class Diagram

9. An instance of a child class in a generalization association is also considered an instance of the parent class.

10. An object can be the "destination" of at most one composition association.

4.1.2 Example

The diagram in Figure 4.1 has the following properties:

- All class names are unique.
- All attribute names in a class are unique.
- **Generalization**/specialization: A *Car* is also a *Vehicle* and a *Ford Prefect* is also a *Car*, so a *Ford Prefect* is also a *Vehicle*. Any statement that holds for a *Vehicle* also holds true for a *Car*, and any statement that holds for a *Car* also holds true for a *Ford Prefect*.
- A *Car* has an attribute called *brand* of type *String*. A *Ford Prefect Car*'s *brand* attribute has the value *Ford*.
- **Associations and roles**: *Person*s can ride in a *Vehicle*. This association is called *ride*. *Person*s can inspect this association by accessing its role *vehicle*. Likewise, a *Vehicle* can determine its *passengers* by inspecting that role; because of specialization, the same holds true for a *Car*.
- **Aggregation and multiplicity**: A *Person* can have any number of *License*s, and a *License* will belong to one *Person*.
- **Composition**: A *Car* has exactly 4 *Wheel*s, which exist only as long as the *Car* which they are a part of does.
- **Association class**: The way *Ownership* is shown suggests that a *Person* owns a Car only between a *startDate* and an *endDate*.
- **Visibility**: A *Wheel* can, by using the association *has part* through the role *container*, access the attribute *brand* and see what its value is.

4.1.3 Informal description of visual elements

Classes

Each class has one name and may have any number of attributes and operations. Both attributes and operations have one of four different visibility types. In the diagram, a class is represented by a rectangle that may be divided vertically into three sections. The class name is always displayed in the top section. The second section may contain the attributes (if any) and the third section may contain the operations (if any).

If in a diagram no attributes or operations are shown, this does not mean that they have not been defined in the model. Although you could view this as two separate models, there are programs that do not. Both Omondo's UML editor and Gentleware's Poseidon editor offer the option to temporarily hide attributes and operations. In UMLet, on the other hand, what is displayed and what is in the model are tightly coupled.

Attributes

Attributes are properties of the class. Each attribute has a name, an optional type and an optional initial value. In a diagram, an attribute may only be displayed in the second section of its class. Usually, each attribute is displayed on a separate line. The absence in the diagram of either a type or an initial value (or both) should not be used to infer that the model does not contain them. A line containing an attribute has the following form:
"name" [: "attribute type"] [= "initial value"]

Operations

Operations indicate methods or procedures that can be performed on (or by) an object. (An object is an instance of a class.) Each operation has a name, an optional list of parameters, an optional return type and optional contents. When displayed, it has the following form:
"name"(["parameter list"]) [: "return type"] ["contents"]

Visibility types

Each attribute and each operation has a visibility type. In a diagram, they may be indicated by a mark before the name. The visibility types are: "public" (+), "protected" (#), "private" (-) and "package" ( ). When an attribute or operation does not have a mark in the diagram this does not indicate any particular visibility.

Associations

Classes in a model can be connected to one or more other classes. This can be done by using the following association types.

- **(basic) Association**
  An association connects two classes. It may have a name, a direction, multiplicity and other properties.
  In a diagram, it is represented by a solid line between (the edge of) the classes. Its name (if any) is displayed close to the middle of the line and at its ends there may be indicators for the multiplicity and other attributes.

- **Aggregation**
  An aggregation is an association that indicates a "has a" or "part of" relationship. When the containing class is destroyed, its parts need not be. A part may participate in multiple aggregation associations. Booch, Rumbaugh

and Grady[BOO05, page 143] state simple aggregation is mostly conceptual. (They still see this type of relationship as useful, if only to distinguish a "whole" from a "part".) However, an important (semantic) difference between aggregation and basic associations is that aggregation should not be cyclic.
In a diagram, it is indicated by a clear diamond shape at the end of the containing class.

- **Composition**
  A composition is similar to an aggregation. When the containing class is destroyed, its parts are destroyed as well. A part may participate in only one composition.
  In a diagram, it is represented by a solid diamond shape.

- **Generalization**
  This is an association relation between two classes. A *subclass* (sometimes also called a *subtype*, *child*, *derived class*, *derived type*, *inheriting class* or *inheriting type*) is considered to be a specialized form of the other. A *superclass* (sometimes also called a *supertype*, *parent*, *base class* or *base type*) is considered to be a generalization of a subclass.
  In a diagram, it is represented by a clear triangle at the end of the line that is connected to the superclass.

If an association itself has properties, this is indicated by attaching a class with those properties to the association with a dashed line. The association together with this attached class is called an *association class*.

Two objects that are related to each other can access the other object by accessing their appropriate role, which is displayed near the association end at the corresponding class.

Multiplicity

In most relationships, an object of type *class A* may be related to more (or less) than one object of type *class B*, and vice-versa. This is called multiplicity. If an object of type *class A* is connected to b objects of type *B* and an object of type *class B* is connected to a objects of type *class A*, we call the relationship between A and B an "**a:b**" association. For example, if each object of type *class A* is connected to *3* objects of type *class B* and each object of type *class B* is connected to *5* objects of type *class A*, this is a "5:3" association.

In a diagram, this may be indicated by displaying "a" near the end of the line connecting the classes that is closest to class A and by displaying "b" near class B. This may be done in several ways:

Multiplicity	**Indicator in diagram**
$\geq m$ and $\leq n$	m..n
n	n or n..n
$\geq m$	m..* or m*
≥ 0	0..* or 0* or *

Beyond the basics

Although the elements described in the previous pages are sufficient for most purposes, UML offers even more. A conscious decision was made not to include them for the consistency requirements, but they are mentioned here for completeness.

- **Dependency association**
  Class A is dependent on class B if a change in (the definition of) class B would

result in a change to class A.
In a diagram, it is indicated by a dashed arrow from the dependent class to the independent class.

- **Realization association**
  This is an association between two classes indicating that one class (the client) implements the behaviour that the other class (the supplier) specifies.
  In a diagram, it is indicated by a dashed line with an clear triangle from the client to the supplier.
- **Comments**
  Comments are pieces of text that can be placed anywhere on the corresponding diagram. Other than basic character spacing and line breaks, no formatting is applied. To indicate that they apply to a specific element, a dashed line is often drawn from the edge of the text to the element (usually either a class or an association). They are usually short, but this is common practice, not a rule.
- **Typed classes**
  If a class has a type, this means it has a specialization relation with the class representing its type.
  In a diagram this is indicated by adding " : class type" after the name.
- **Extra association adornments**
  Examples of this are qualifications.

Diagram best practices

1. All multiplicities should be displayed the same way. (i.e. "0..*", "0*" and "*" should not appear in the same diagram.)
2. If visibility is indicated on any attribute or operation, it should be indicated on all (displayed) attributes and operations.
3. If the type of any attribute in a class is displayed, the (known) types of all (displayed) attributes in that class should be displayed.
4. If the return type of any operation in a class is displayed, the (known) types of all (displayed) operations in that class should be displayed.
5. The argument list of all displayed operations should be displayed.
6. Any rectangle and any other element should not overlap.

4.2 UML State Machine Diagram

4.2.1 Abstract definition

Syntax

A UML state machine diagram (USMD) is a triple (S, E, T), where S is a set of states, connected by directed edges T that are labelled with events from E. The set S is the disjoint union of the sets SS, CS, ES and XS, respectively the simple, composite, entry, and exit states. The formal definition is as follows.

$$USMD = \{(S, E, T) \mid T \subseteq S \times E \times S\}$$

Entry (exit) states possess no incoming (outgoing) arcs:

$$\langle \forall (s, e, s') \in T : (s \notin XS \wedge s' \notin ES) \rangle$$

Semantics

A USMD M shows a state machine, emphasizing the flow of control from state to state. A composite state h in M corresponds to another USMD M^h with entry (exit) states and events from M^h corresponding to all states immediately preceding (following) s. The formal definition is as follows.

Each USMD $M = (S, E, T)$ and each composite state $h \in CS$ should correspond to a subordinate USMD $M^h = (S^h, E^h, T^h)$ such that $S \cap S^h = \emptyset$. An edge $(s, e, s') \in T^h$ satisfying $s \in ES^h$ should correspond to one or more edges $(r, e, h) \in T$. Vice versa, to each edge $(r, e, h) \in T$ should correspond an edge $(s, e, s') \in T^h$. Edges $(s, e, s') \in T$ satisfying $s' \in XS^h$ should correspond in the same way to edges $(h, e, r) \in T$. Edges $(h, e, h) \in T$ should correspond to one or more edges $(s, e, s') \in T^h$ with $s, s \notin ES^h \cup XS^h$. There may exist edges $(s, e, s') \in T^h$ with $s, s' \notin ES^h \cup XS^h$ that do not correspond to an edge $(h, e, h) \in T$.

It is possible to expand the composite state h in M, resulting in an expanded USMD $M[h] = (S', E', T')$ satisfying

- $S' = (S \setminus \{h\}) \cup SS^h \cup CS^h, E' = E \cup E^h$,
- $T' = T_1 \cup T_2 \cup T_3 \cup T_4$,
- $T_1 = \{(s, e, s') \in T : s <> h \wedge s' <> h\}$,
- $T_2 = \{(r, e, s') : (r, e, h) \in T \wedge (\exists s' \in ES^h : (s, e, s') \in T^h)\}$,
- $T_3 = \{(s, e, r) : (h, e, r) \in T \wedge (\exists s' \in XS^h : (s, e, s') \in T^h)\}$, and
- $T_4 = \{(s, e, s') : (s, e, s') \in T^h \wedge (s, s' \notin ES^h \cup XS^h)\}$.

A USMD (S, E, T) is internally consistent if every state in S is reachable from a state in ES. Let $\rightarrow$ be the relation over S defined by $s \rightarrow s' \equiv \langle \exists e \in E : (s, e, s') \in T \rangle$ and $\rightarrow^*$ its transitive closure. Then for every $s \in S$ there must exist an $i \in ES$ such that $i \rightarrow^* s$.

Let U be a set of USMDs such that each composite state h in any USMD $M \in U$ corresponds to a subordinate USMD $M^h \in U$ An USMD $M \in U$ is consistent within U if it is internally consistent and either does not contain composite states or can be expanded to an USMD consistent within U.

4.2.2 Example

In Figure 4.2, an USMD M is depicted with a composite state H, its subordinate M^H and its expansion $M' = M[H]$.

4.3 UML Activity Diagram

An UML Activity Diagram (UAD; see Figure 4.3) is used mostly to model workflows. It consists of a set of nodes which are connected to each other by control structures in a directed graph. I use a somewhat simplified version; for a more in-depth examination of the semantics of UADs, see [ESH01].

4.3.1 Abstract definition

Syntax

$$UAD = \{(N, S, C)\}$$

where

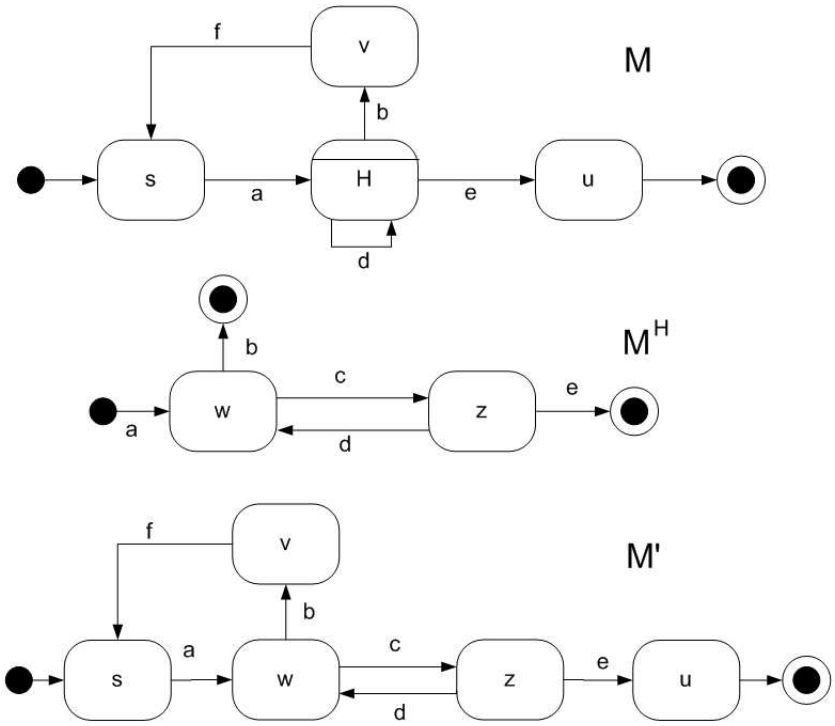

Figure 4.2: Example UML State Diagram

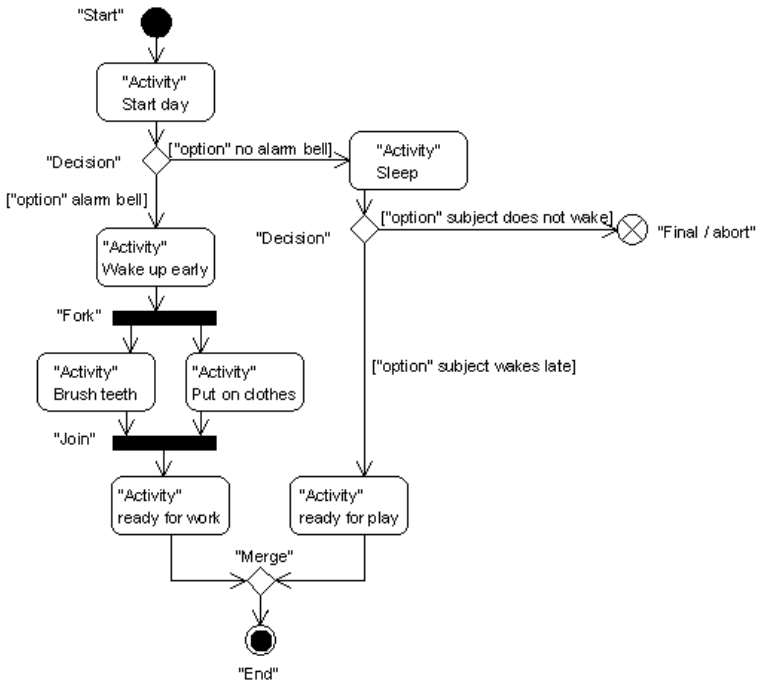

Figure 4.3: Example UML Activity Diagram

1. N is a set of *nodes*; a *node* is a pair (l:label, t), where $t \in \{start, activity, end, abort\}$.
2. S is a set of *control structures*; a *control structure* is an element of the set $\{decision, fork, join, merge\}$.
3. C is a set of *connections*; a *connection* is a triple (*o1*:from, *o2*:to, l:label) where $o1, o2 \in N \cup S$.

Semantics

The (informal) definition of the semantics of a UAD is as follows:

An UAD defines a set of possible workflow executions. In other words, the semantics of a UAD is defined by the set of (valid) finite lists of nodes.

In all these lists, the first node is the *Start* control structure and the last node is either an *End* or an *Abort* node. The semantics at a *Decision* control structure is the union of the semantics at all outgoing edges, each preceded by the semantics at the incoming edge. The semantics of a Merge control structure is the union of the semantics at all incoming edges, each followed by the semantics at the outgoing edge. The semantics of a *Join* control structure is the union of all possible interleavings of the semantics at all incoming edges (originating from the last common *Split*), each followed by the semantics at the outgoing edge. The semantics of a *Split* control structure is the semantics at the incoming edge followed by the union of all possible interleavings of the semantics at all the outgoing edges.

Some extra requirements that are made of an UAD are:

1. There is no node before Start.
2. There is no node after End or Abort.
3. All Forks and Decisions have at least two outgoing edges and exactly one incoming edge.
4. All Joins and Merges have at least two incoming edges and exactly one outgoing edge.

4.3.2 Informal description of visual elements

The nodes are:

1. A **start** node;
2. An **activity** node;
3. A **end** node, indicating successful termination; and
4. A **final** (also called "**abort**") node, indicating unsuccessful termination.

Processing is the execution of a workflow starting at the start node and continuing with activities as determined by the control structures. Processing ends when processing of all threads has ended in either an end or a final node.

The set of control structures is:

1. **Decision**, indicating a choice between two or more alternatives;
2. **Fork**, for starting parallel threads;
3. **Join**, for joining parallel threads into one; processing will not continue until all threads have finished; and

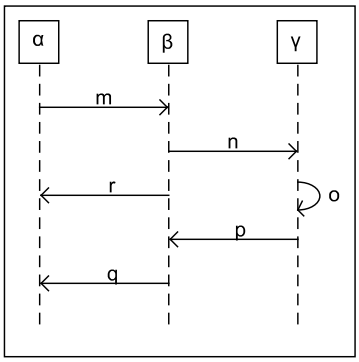

Figure 4.4: Example UML Sequence Diagram

4. **Merge**, for joining alternatives; no condition is necessary.

In a diagram, a *start node* is represented by a solid circle, an *end node* by a clear circle containing a solid, smaller circle and a *final node* by a clear circle filled with a cross. An *activity node* is represented by a labelled rectangle with rounded corners.

A *Decision* is represented by an unlabelled diamond shape with one incoming edge and several outgoing edges. Options are indicated between "[" and "]" at outgoing edges. A *Fork* is represented by a black bar with one incoming edge and at least two outgoing edges. A *Join* is represented by a black bar with one outgoing edge and at least two incoming edges. A *Merge* is represented by a diamond shape with several incoming edges and one outgoing edge.

Edges are represented by arrows.

Beyond the basics

An extension is the possibility to divide all the activities into so-called "swim lanes" where a certain person, role or department is responsible for executing all the activities in his lane. Although this might be useful in some cases, we feel most of those cases will be better served by using a sequence diagram.

Diagram best practices

1. Each non-trivial option is shown near the start of the corresponding arrow.
2. Each condition is shown near the start of the outgoing arrow.

Model requirements

The model must meet the following requirements:

1. A Decision has a defined option for each outgoing edge. "Obvious" options that some modellers leave out are still considered defined options.

4.4 UML Sequence Diagram

A UML Sequence Diagram (USD) is used to clarify the process flow. Similar to a collaboration diagram, it shows the order of messages between objects. A more complete semantics is given in [LI04] and [MEN08].

4.4.1 Abstract definition

A (simplified) USD consists of a set A of actors, a set M of messages and an ordering function O describing the ordering of events (sending and receiving messages) per actor. ($m!$ denotes the sending of message m and $m?$ denotes the receiving of message m.) The lists *O(a)* and M together define a transitively closed ordering relation "$<$" between events, that should be irreflexive for the USD to be consistent. It should also not occur that two messages are sent in a certain order and received in the opposite order.

Note that an actor can send messages to itself. We can add a labelling function to messages, so that different messages may have the same label.

Also note that this description deviates slightly from the one given in [LI04]. Using our syntax, multiple messages can be sent at the same time because the order of messages is particular to a specific lifeline, not the whole system. It also omits the concept of "initial locations" from [MEN08].

Syntax

$$USD = (A, M, O)$$

where

1. A, M sets
2. $O : A \to L(E)$
3. $E = \{m! : m \in M\} \cup \{m? : m \in M\}$ (with *m!* denoting a sent message and *m?* denoting a received message)
4. $L(E)$ is the set of E-lists.

Semantics

An USD defines a set of possible message sequences. In other words, the semantics of a USD is defined by the set of (valid) finite lists of messages.

Consistency requirements:

1. Each event is unique: if $a, b \in A; e \in E; u, v, w, z \in L(E)$ such that $O(a) = uev \wedge O(b) = wez$, then $a = b \wedge u = w \wedge v = z$.
2. Let "$<$" be the smallest transitively closed relation on E that satisfies $e < f$ if $\langle \exists a, u, v, w : a \in A \wedge u, v, w \in L(E) : O(a) = uevfw \rangle \vee \langle \exists m \in M : e = m! \wedge f = m? \rangle$. Then $\langle \forall e \in E : \neg(e < e) \rangle$ and $\langle \forall m, n \in M : \neg(m! < n! \wedge n? < m?) \rangle$.
3. The lists *O(a)* contain no duplicates and the sets *E(a)* are disjoint.
4. The global partial order contains no cycles.

4.4.2 Example

As an example, we take the USD as defined in Figure 4.4. In that diagram,
$A = \{\alpha, \beta, \gamma\}$,
$M = \{m, n, o, p, q, r\}$,
$O(\alpha) = [m!r?q?]$,
$O(\beta) = [m?n!r!p?q!]$ and

$O(\gamma) = [n?o!o?p!]$.
This diagram is consistent; it would have been inconsistent if e.g. $O(\gamma) = [n?o?o!p!]$.

A completed trace is an E-list containing every E-element once. A completed trace t accords to the order "$<$" if $\langle \forall e < f \in E : \langle \exists u, v, w \in L(E) : t = uevfw \rangle \rangle$.

The semantics of an USD is the set of completed traces. A completed trace of Figure 4.4 is e.g. $[m!m?n!n?o!r!o?p!p?q!r?q?]$.

4.4.3 Informal description of visual elements

An object that can send or receive messages is called a *lifeline*. In a diagram, it is represented by a rectangle with the object's name in the middle and a dashed line at the bottom indicating that it can send or receive messages. The way to read a USD is vertically, with the presumption that higher arrows indicate messages that have been sent earlier.

A *message* is represented by an arrow from the dashed line of one lifeline to the dashed line of another lifeline. An object can also send a message to itself; the arrowhead usually ends slightly below the starting point of the arrow. Any message must be labelled. A *label* contains the message name and possibly other parts, like a condition ("$[a > 0]$"), an iteration marker ("*[for all database entries]") or an assignment ("[a:=-1]").

Diagram requirements

1. Message signatures (name and sending/receiving lifeline) should be clearly indicated. (This is especially important when a message with the same name is sent between different objects.)

Chapter 5

Examples of inter-model consistency

The previous chapter describes the requirements that models of four specific types (UCDs, USMDs, USDs and UADs) must meet in order to be considered internally consistent. This chapter contains examples of consistency rules across models of the above mentioned types. The considered models should all refer to the same system; moreover, some naming convention should allow the identification of elements of either model. This makes it possible to determine the overlap between the given models.

5.1 USD / UCD

5.1.1 Example

The USD in Figure 5.1 contains lifelines (actors) that are reflected as classes in the UCD. The possible occurrence of messages in the USD is reflected as a relation between the classes. Finally, the A-to-B message labelled x in the USD corresponds to an operation x of class B (and likewise for the y-labelled message).

5.1.2 In General

To every USD should correspond a UCD containing classes that correspond to its lifelines. So there exists an injection I from the set L of lifelines in the USD to classes in the UCD. The classes in $I(L)$ should satisfy:

1. If there exists a message (in either direction) between A, B in L, then there exists a relation between $I(A)$ and $I(B)$.
2. A message labelled x from A to B should correspond to an operation labelled x in $I(A)$.

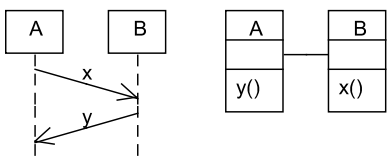

Figure 5.1: Example USD/UCD

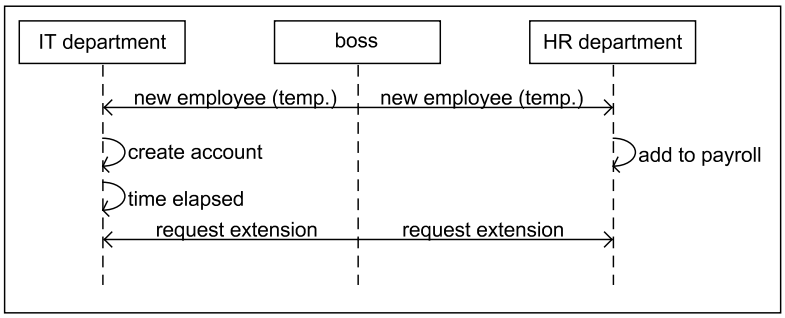

Figure 5.2: Example USD/USMD: USD

It should be noted that in the case of message-orientation, operations should not have a return value. Operations x that do have a return value should correspond to a pair of messages (e.g. x, x_r) in the USD. In the following, we assume operations without return values.

5.1.3 Formal rules

Let (A, M, O) be the USD and (C, A') be the corresponding UCD. Then there should exist an injection I from A to C satisfying:

1. If $\langle \exists X, Y \in A; m \in M : \{m?, m!\} \cap O(X) \neq \emptyset \wedge \{m?, m!\} \cap O(Y) \neq \emptyset \rangle$, then $\langle \exists (l, at, From, To, ac) \in A' : \{From, To\} = \{I(X), I(Y)\} \rangle$.
2. If $\langle \exists X \in A; m \in M : m? \in O(X)$ with $I(X) = (l, P, M') \rangle$, then $\langle \exists y \in M' : y.l = m \rangle$.

5.2 USD / USMD

5.2.1 Example

There are no elements in both diagrams that are equivalent by name and it is the responsibility of the modeller to add the necessary connections. The USMD in Figure 5.3 has been indicated as an elaboration of the "IT department" lifeline from the USD in Figure 5.2. The event *new employee (temp.)*? in the USD corresponds to the initial event leading to the state *new employee form received* in the USMD. Likewise, the events *time elapsed*? and *request extension*? in the USD corresponds to the respective events *timeout* and *extension* in the USMD.

Regarding consistency rules, the partial orders in both diagrams should not be conflicting. The depicted diagrams clearly exhibit a conflict, since the occurrence of *time elapsed*? followed by *request extension*? in the USD cannot be matched in the USMD.

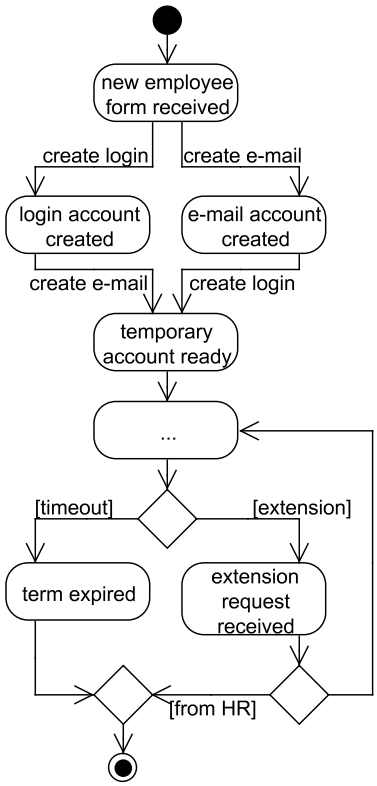

Figure 5.3: Example USD/USMD: USMD

Chapter 6

Conclusions

6.1 Summary

In order to stay competitive, organisations need to integrate their processes. This thesis is part of a project to support system integration through (message-oriented) middleware. The functionality of the systems to be integrated, as well as the containing integrating system, is for a large part determined by models. (See chapter 1.)

In this thesis, we adhere to the IEEE1471 standard towards modelling, with notions such as Model, Viewpoint, Library Viewpoint, View, Concern, Stakeholder and Architectural Description. (See chapter 2, subsection 2.1.1.)

The models that determine a system's functionality are built from various views, represented as diagrams. A major challenge in integration is the fact that models and diagrams have often been developed independently, resulting in incompatibilities when combining them.

A predecessor thesis [BRO07] recommends a framework supporting consultants in storing, maintaining and comparing the various models (and the underlying diagrams) of the systems to be integrated. (See chapter 2, section 2.3.) The thesis is primarily focused towards the necessary infrastructure.

This thesis complements its predecessor by discussing the needed functionality. In essence, the framework should contain the meta-models for the models needing integration, so that all the models can be stored in a common repository. To this repository, conditions can be added checking for consistency, within the same model or between models. The conditions describe the requirements that a consistent set of models should meet. (See chapter 3.)

We describe the syntax of four types of models, and give examples of consistency rules together indicating how they could be verified. (See chapters 4 and 5.) These rules are given in the required context of the semantics. (See $PRED(C, A, O, R)$ from section 4.1 for an example.)

Finally, we recorded some additional requirements for the framework. These include:

- Differentiating between the different user roles;
- System maintenance:
  - Adding new meta-models;
  - Adding and editing rules defining what is required for one or more models to be consistent;
- Modelling:

  - Adding, editing and deleting models;
  - Indicating that models are related (and how);
  - Checking for consistency
- Open source, to prevent being dependent on a specific vendor;
- Storing models in an open, interoperable data format.

6.2 Recommendations

In this thesis, we examined several meta-models and consistency rules. We also formulated a set of system requirements. All of these need to be studied in depth.

Quite a bit of work can still be done on the meta-modelling aspect of this project. MOF can be used to describe not only those meta-models covered by UML, but others as well. MOF is an open industry standard, which makes development of the tool easier to do when the software is also open source.

The semantics of the four meta-models presented in chapter 4 can be explored in more depth. More meta-models may be added and the current meta-models may be expended with more details. For UML, [FOW97, BOO05] contain much information that is useful. This task should not be given priority because the meta-models from this thesis are sufficient for the development of a tool. For those interested, the UML 2 Semantics Project ([UML2S]) has made significant progress in this direction.

At several points, we indicate that parts of models can be related by a modeller, without offering too much detail on what a relationship looks like. It may only indicate that the parts are related, but not how. This kind of relationship is useless for consistency checking, so the modeller should fill in the details after more is known about the models. Exactly which models and model elements can be related and how, as well as which consistency rules may be applied needs to be defined; possible examples of this are relating elements from different models by name, or the injection I from subsection 5.1.2. In order to be able to design a first version of the tool, a few simple relationships should suffice.

After defining relationships, consistency rules on those relationships can be defined. These must necessarily include consistency rules for single models (which may not be covered by the meta-model) but more importantly rules for two or more related models.

Chapter 3 gives a few system requirements for the tool. These have to be written down in a user requirements document, a system requirements document and an architectural design document, according to common software development methods. The different stakeholders (modeller, meta-modeller and system maintainer) that will be working with the system have to be taken into account.

Early on in the research, a need was expressed for the tool to be developed as an open source tool independent of any particular vendor. [BRO07] recommends using Eclipse, EMF and GEF, assisted by GMF because more than one vendor is adopting the Eclipse platform. The status of these developments can be examined.

It is prudent, before developing the entire tool, to determine whether the methods described in [BRO07] are flexible enough to make adding new meta-models more or less easy. In particular, whether GEF is suitable for dynamically adding new modelling types with new visual elements, or whether it can only be used at development time to create a new editor. [BRO07] offers some suggestions on the construction of views and the storage of models in a database, but the details still need to be worked out.

Appendix A

List of References

BOO05 Grady Booch, James Rumbaugh, Ivar Jacobson
The Unified Modeling Language User Guide
Addison Wesley, Second Edition, 2005

BRO07 Dick van den Broek
Abstracting from Message-Oriented Middleware implementations
Master thesis
University of Twente, 2007

CHA04 David A. Chappell
Enterprise Service Bus
O'Reilly Media, 2004

ESH01 Rik Eshuis, Roel Wieringa
A Formal Semantics for UML Activity Diagrams
- Formalising Workflow Models
University of Twente, 2001

FOW97 Martin Fowler, Kendall Scott
UML Distilled - Applying the standard Object Modeling Language
Addison Wesley, 1998

IEEE1471 Software Engineering Standards Committee
IEEE Recommended Practice for Architectural
Description of Software-Intensive Systems
IEEE Computer Society, 2000

KRA04 Dirk Krafzig, Karl Banke, Dirk Slama
Enterprise SOA, Service-Oriented Architecture Best Practices
Prentice Hall, 2004

LI04 Xiaoshan Li, Zhiming Liu, He Jifeng
A Formal Semantics of UML Sequence Diagram
Proceedings ASWEC'04, pages 168-177
IEEE Computer Society, 2004

LIN04 David S. Linthicum
Next Generation Application Integration
Pearson Education, 2004

MEN08 Sun Meng, Luís S. Barbosa
A Coalgebraic Semantic Framework for Reasoning about

UML Sequence Diagrams
Proceedings QSIC'08, pages 17-26
IEEE Computer Society, 2008

MOF Object Management Group
Meta Object Facility
http://www.omg.org/technology/documents/formal/mof.htm

MOF00 Object Management Group
Meta Object Facility (MOF) Specification
version 1.3, March 2000

SCA SCA Working Group
Service Component Architecture Specifications
http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

SZL06 Marcin Szlenk
Formal Semantics and Reasoning about UML Class Diagram
Proceedings DEPCOS-RELCOMEX'06, pages 51-59
IEEE Computer Society, 2006

UML2S **UML2 Semantics Project**
http://research.cs.queensu.ca/ stl/internal/uml2/index.html

URL01 **Message-oriented middleware**
http://en.wikipedia.org/wiki/Message_oriented_middleware

XMI Object Management Group
XML Metadata Interchange
http://www.omg.org/technology/documents/formal/xmi.htm

Appendix B

List of Definitions

AD	Architectural Description
EMF	Eclipse Modeling Framework
ESB	Enterprise Service Bus
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
MOF	Meta Object Facility
MOM	Message-Oriented Middleware
MVC	Model - View - Controller
RPC	Remote Procedure Call
SOA	Service Oriented Architectures
UML	Unified Modelling Language
UAD	UML Activity Diagram
UCD	UML Class Diagram
USD	UML Sequence Diagram
USMD	UML State Machine Diagram

$\pi_x(y)$ or $y.x$	The value of attribute x from tuple y
$\Pi_x(Y)$	$\{\pi_x(y) : y \in Y\}$