University of Oslo
Department of Informatics

UMLexe – UML virtual machine

A framework for model execution.

Kai Fredriksen

Master thesis

12th May 2005

Abstract

The aim of this thesis is the specification and development of a new UML virtual machine – UMLexe- capable of executing platform independent system specifications. For executing models, computational completeness is required and UMLexe propose a subset of UML and operational semantics for executing those models. UMLexe will provide prototype functionality to prove the concept of executing components combined with interaction models.

The first part of the thesis describes a case scenario illuminating the model notation. After a more detailed look at the specification and implementation, this case is executed to prove the concept. The last part of the thesis is dedicated to the specification and development of the UMLexe virtual machine and the evaluation of the implementation in terms of defined requirements and existing solutions executing UML models.

Foreword

This thesis is submitted for the fulfillment of Master of information technology degree at the Department of Informatics, University of Oslo (UIO). The work of this thesis has been supervised by Arne Jørgen Berre and Jon Oldevik (SINTEF). The thesis has been a project at the department of Cooperative and Trusted systems, SINTEF Oslo.

I would like to thank Jon Oldevik for his invaluable support and linguistic help, Arne-Jørgen Berre for his tolerance and my family for their constant support during the final period of writing this thesis.

Oslo,
May 2005 Kai Fredriksen

List of figures

List of tables

Contents

1 Introduction

This thesis addresses the problem of executing models representing software systems. In this context the most interesting models are Unified Modeling Language (UML) models. Using models to create software systems has the advantage of raising the level of abstraction. Models simplify program description from textual syntax to graphical diagrams. However, models lack the capability of being executable the way programming languages are.

Using models to describe software systems started with NIAM and Entity Relationship (ER) diagrams for describing database schemas. These languages were the first steps toward using graphical models creating database systems. Another approach was reverse engineering of database models in order to create user interfaces. One of the main contributing tools for this approach was Systemator [1]. Systemator provided users with transformation tools from database to user interface transformations. Systemator early addressed model driven development using models to something more than specification artifacts.

Since the introduction of Systemator there has been a paradigm shift from procedural programming towards object oriented programming. Given the change towards object oriented programming, a desire for representing software with models introduced the Object Modeling Techniques (OMT). OMT was later on unified with techniques from Booch and Jacobson which became the Unified Modeling Language (UML) [2]. UML is a graphical modeling language for modeling software systems.

Today, models have become an integral and important aspect of software development. Further utilization of these models has forced a desire for executing models. However, there is a gap between executable programming language and executable models. For executing models, a modeling language needs operational semantics [3] similar to programming languages. There are tools today which execute UML models, although they only rely on state machine diagrams to describe system behavior. This is a problem because most systems described using UML today use activity, collaboration or sequence diagrams to describe behavior.

This thesis addresses the problem of executing UML models and presents an approach - UMLexe - which aims to use UML 2.0 models containing components and interaction diagrams to execute software operational behavior.

1.1 Problem definition

Within this thesis the aim is to describe an executable engine for UML models called UMLexe. The goal is to support execution of UML components and interaction models. UMLexe will provide openness for existing tools to integrate with. This thesis and UMLexe will provide a proof of concept that can serve as basis for further discussion within the field of executable UML.

1.2 Motivation

There are various kinds of projects dealing with executable UML models. When it comes to executing UML models there are three main approaches:

- Transforming models to code.
- Executable UML.
- UML virtual machine.

Transforming models to code aim at creating executable code from models, unfortunately this approach do not take advantage of models describing behavior and generated code have to be completed with behavioral expressions. Executable UML mainly uses class diagrams and state machines added with an abstract action language. There are basically three categories of model execution covered by the term UML virtual machine:

- Passive models where a description of how to execute the model is added.
- Dynamic models where objects are created and users can execute operations on those objects.
- Visualization which aim at providing execution either by test generators, simulation or analysis tool.

All executable UML tools today take advantage of class diagrams for modeling objects structure and state diagrams modeling object behavior. This is acceptable for developers of hardware and real time systems, but we need a way to execute models specified with components and interactions as well. Components will give us the possibility to model "real objects" [4] and the roles they play with collaborating objects. Interactively instantiating UML components will provide us with the possibility to execute components behavior. This is valuable for testing distributed system behavior, where components represents systems or business processes.

1.3 Contribution

The main contribution of this work is the design and implementation of a UML virtual machine using the Java language and the Eclipse platform. We define a notation and operational semantics for executing UML components and interactions. The proposed mechanism especially implements support for executing components and interaction models.

1.4 Structure of the thesis

The rest of this thesis is structured as follows.

Chapter 2 – Background. This chapter introduces basic terminology of the executable UML models, thereby describing the theoretical context of this thesis.

Chapter 3 - Example case. This chapter describes the example case, Min-max replenishment. This UML 2.0 model constitutes a basis for precise description of models executed and presented in following chapters.

Chapter 4 – Requirements for UML virtual machine. This chapter presents requirements for executing UML models.

Chapter 5 - Evaluation of existing tools. This chapter evaluates existing UML 2.0 modeling tools and execution engines.

Chapter 6 –UMLexe – a UML virtual machine. This chapter presents our solution, UMLexe, in terms of requirements, platform-independent specifications and implementation.

Chapter 7 - Proof of concept

Steps through an execution of the example case.

Chapter 8 - Discussion

This chapter discusses how UMLexe evaluates against existing requirements described in Chapter 4.

Chapter 9 - Conclusion and further work. This chapter summarizes the results of previous chapters as well as discussion of possible future extensions.

The reader should have knowledge of UML [2], and understand specifications created with UML 2.0 and OCL. A good book on UML 2.0 [5] is the Unified Modeling Language Reference by Rumbaugh, Booch and Jacobson. The UML 2.0 notation summary in this book is a useful resource when reading. Furthermore, an extended version of OCL, executable OCL (XOCL), is used to describe operational semantics within UMLexe.

2 Background

This chapter presents executable UML terminology and describes how the main subject of this thesis - UML virtual machine - is related to these terms. Furthermore, a description of actual approaches for executing UML models is given.

2.1 UML virtual machine terms

The UML virtual machine terms is related to the term virtual machine in general. Virtual machine is defined in [6]:

"A running program is often referred to as a virtual machine - a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn't merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.... The virtual machine: A way of understanding software that frees us to think of software design as machine design."

With this definition in mind this thesis use the term UML virtual machine from [7]:

"UML virtual machine is an abstract computing machine (like any VM), which provides an instruction set and a memory model for representing objects. Instruction set of a UML virtual machine is UML itself and memory model of the UML virtual machine is the memory management facilities of the implementation language"

The following sections elaborate on terms of the concept of UML virtual machine, dynamic UML models and related concepts.

2.1.1 Overview of terms

The relations between some of the most central concepts introduced in this chapter are shown in Figure 1. A few details of the diagram require some notes:
In order to create executable UML models the UML virtual machine is able to execute the modeling language requires operational semantics to be added. Operational semantics is an extension to UML 2.0 and provides necessary semantics for executing these models.

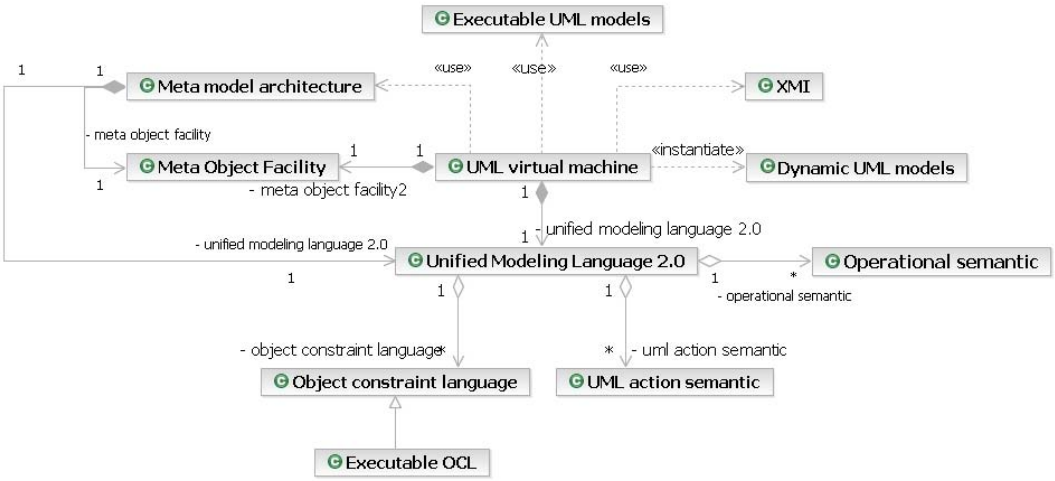

Figure 1 UML virtual machine terms

2.1.2 Computationally complete UML models

UML has been criticized for being a semi formal modeling language, with which it is difficult to produce precise models. One cause of imprecise models is the fact that modelers use UML in different ways, like sketching or on the other hand the communities defining UML models in a fine grained manner. With this in mind we need to define a set of models which defines structure and behavior of a system in order to be executed. Computationally complete is defined in [8]:

"A computationally complete subset of UML is one that is sufficiently expressive to allow definitions of models that can be automatically executed on a computer by an execution tool"

For a UML virtual machine to execute UML models these models must be computationally complete.

2.1.3 Dynamic UML models

Dynamic UML models are instantiated in a UML virtual machine and define the specification of a system. It is possible to create object instances from these models and let the user interact with these objects. Models may answer questions about the current state of the system, what is the type of a model object and even reflect changes in the system specification to the runtime instances. Furthermore, it is possible to call operations on objects instantiated.

In order to realize dynamic UML models UMLexe need architecture to reflect changes between the model and the runtime environment.

2.1.4 Meta Object Facility

In order to create dynamic models there must be an "instance of" relationship [9] between objects in runtime and Meta level objects. The generally accepted conceptual framework for Meta modeling is based on architecture with four layers. Meta Object Facility (MOF) provides a set of generic domain independent concepts, relations and it is self-defined [10]. MOF is defined in [11] as:

"Meta Object Facility (MOF) is a model driven, distributed object framework for specifying, constructing, managing, interchanging and integrating Meta data in software systems"

Figure 2, gives an overview of OMG Meta hierarchy.

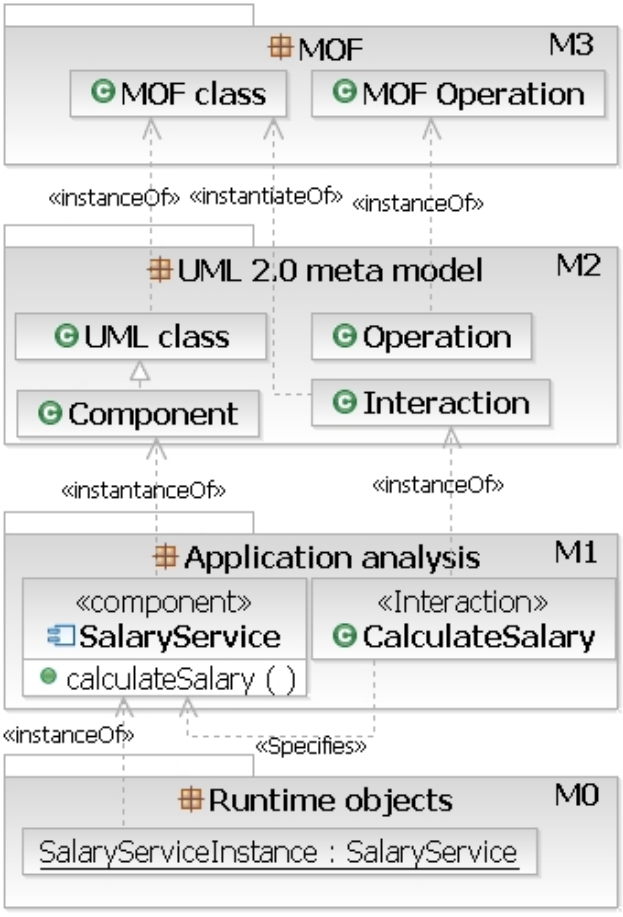

Figure 2 Meta hierarchy

The Meta Object Facility (MOF) is a standard adopted by Object Management Group (OMG). The MOF architecture consists of four Meta levels, also known as M3, M2, M1 and M0:

- **Level M3** is the MOF, whose elements are constructs of MOF which supplies for defining Meta models. These elements include Class, Attribute, Association and operation etc.

- **Level M2** is populated by Meta models defined via the MOF elements. Elements in these Meta models are defined using MOF Class, MOF Operation etc. M2 elements are instances of M3 elements and in Figure 2 M2 level consist of UML 2.0 Meta model elements.

- **Level M1** is populated with models consisting of M1 elements. M1 elements are the one defined by an engineer during analyses and design phases. M1 elements define a template for domain entities. M1 elements are instances of M2 elements.

- **M0 Level** consist of objects and data, which in are instances if M1 elements. If we use our example with SalaryService, a specific SalaryServiceInstance (M0) is an instance of SalaryService (M1).

In a UML tool you will most of the time only construct M1 models where classes are things like SalaryService. The M2 and M3 levels are valuable for tool vendors and provide principles and standards for creating development tools. Finally, MOF provides core elements for defining modeling languages, like UML 2.0. Any kind of Meta models can be developed using this standard.

2.1.5 Unified Modeling Language 2.0

As mentioned in previous sections, executing models in general requires computationally complete models. This thesis elaborates on the new Unified modeling language standard and aim at creating an executable subset of UML. Unified modeling language (UML) [2] is a graphical modeling language for visualizing, specifying, constructing, and documenting artifacts of software systems.

"UML is the open, industry standard visual modeling language approved by the OMG" [5]

UML 2.0 has introduced four new diagrams (Composite, Interaction overview, Interaction) in addition to the nine diagrams defined in UML 1.x. There are nine different views in UML 2.0 with the two main categories in UML are, behavior models and static structure models [5].

Table 1 describes the overall view of diagrams and notation in UML 2.0, although in order to execute UML we need action semantics as well. Action semantic was introduced in version 1.5 of UML and defines the core action semantics for UML 2.0. The action semantic is elaborated in the next section.

Views	Diagram	Concepts
Use case view	Use case	Actor, association, extend, include, use case, generalization
Static view	Class	Association, class, dependency, generalization, realization
Design view	Internal structure of objects, collaboration, component	Connector, interface, part, port, provided interface, role, required interface, collaboration, component, dependency, subsystem, realization
Interaction view	Interaction, communication	Occurrence specification, execution specification, interaction, interaction fragment, interaction operand, lifeline, message, signal, collaboration, guard condition, role, sequence number
State machine view	State machine	Completion transition, do activity, effect, event, region, state, transition, trigger
Activity view	Activity	Action, activity, control flow, control node, data flow, exception, expansion region, fork, join, object node, pin
Deployment view	Deployment	Artifact, dependency, manifestation, node
Model management view	Package	Import, model, package
Profiles	Package	Constraint, profile, stereotype, tagged value

Table 1 UML 2.0 views

2.1.6 UML Action Semantics

Action semantics defines a set of actions a runtime environment must handle. Runtime environment is for instance, an execution environment like Java virtual machine [12], Squeak [13], Smalltalk [14].

Action Semantics [15] (AS) extends UML with mechanisms for creating platform independent models. That is, models will not only define the information structure for instance classes, their attributes and relations, but also the manner it may react in response to a particular stimulus, for instance, what changes are made within the model and how they are applied. I fact, the abstract action language introduced in UML makes it executable; a software engine would be in position to determine the result of the occurrence of a particular event on a particular situation. Expected benefits are easier and earlier validation of the model, an optimization of code writing by increasing the portions of generated code, and automatic test generation [10]. In principle, the action semantics can be combined with a textual notation to write statements which operates on UML models directly [16].

Table 2 gives an overview of UML 1.5 and 2.0 actions.

Action	Description
Composite actions	Group actions together
Read and Write actions	Get, write, modify values and create and destroy objects
Computation actions	Transform a set of input values to a set of output values
Collection actions	Each collection action contains a sub action, an embedded action that is executed for each element in the input collection.
Message action	Exchange messages among objects. Messages may be asynchronous or synchronous.
Jump action	Jump is a condition that occurs synchronously during the execution of an action.

Table 2 UML runtime actions

Action semantics provides basic elements for realizing dynamic UML models, although to operate on these models, UML should provide textual notation for these actions as well. There are several projects which have defined an abstract action language with textual notation. The example languages are:

- The Action Specification Language (ASL). A public domain language of which there have been several implementations[1].
- The Bridge Point Action Language (AL). An action language supported by the Bridge Point modeling tool[2].
- The Kabira Action Semantics (Kabira AS). An action language for the ObjectSwitch middle-tier server suite[3].
- The action language subset of Specification and Description Language (SDL). An international standard widely used in the telecom industry[4].

[1] See The Action Specification Language Reference Manual available at www.kc.com.
[2] More information is available from www.projtech.com.
[3] More information is available at www.kabira.com.
[4] More information is available at http://www.sdl-forum.org.

2.1.7 Object Constraint Language (OCL)

The UML diagrams represent a powerful set of techniques to describe different views on a software system. Generally though, they are not capable of describing every possible property. The textual constraint language OCL, therefore, has been added to the UML in order to describe properties not to be conveniently captured by diagrams. OCL is an object constraint language in which you can write constraints over models, for instance, derivation rules for attributes, the body of query operations, invariants, and pre- and post- conditions. OCL can be used for both UML and other MOF based models. Using OCL extends the power of UML and MOF; it allows the modeler to create more precise and more extensive models [17].

2.1.7.1 OCL Primitive types

The primitive types defined in the OCL standard library are Integer, Real, String and Boolean. They are all instance of the Meta class Primitive from the UML core package. The standard type Real, Integer represents the mathematical concept of real and integer. [18]. The standard type String represents strings, which can be either ASCII or Unicode. String is itself an instance of the Meta type primitive [18]. The standard Boolean type represents the common true/false values. Boolean is it self an instance of the Meta type Primitive.

All of these primitive types provide well defined semantics for different kinds of operations. For instance Integer types provides support for +, -, *, / and strings provides operation for size, concatenates substring and so forth.

2.1.8 Types of constraints in OCL

OCL provide four types of constraints:

- An invariant is a constraint that states a condition that always must be met by all instances of the class, type, or interface. An invariant is described by using an expression that evaluates to true if the invariant is met. Invariants must always be true.
- A precondition to an operation is a restriction that must be true at the moment the operation is going to be executed. The obligations are specified by post conditions.
- A post condition to an operation is a restriction that must be true at the moment the operation is finished.
- A guard is a constraint on a state that must be true before state transition fires.

Furthermore, constraints are always coupled to items used in a model. The connected item is called the context of the constraint. OCL is declarative language and do not change the state of objects it refers to. In some situations this is a good thing because it guarantees side-effect free evaluation. However this limitation makes it very difficult to describe operational behavior in a way which can be readily executed. Standard OCL provides pre-/post- conditions as a way of specifying the effect of an operation, however in general these cannot be executed. An alternative approach is to augment OCL with action primitives. The executable OCL (XOCL) language [3] extends OCL with a number of key behavior primitives. The action primitives provided by XOCL are slot

update, object creation and a sequential operator. An example of use of actions in XOCL is given in Table 3.

```
context Person
 @operation calculateSalary()
  let salary := self.hours * rate;
 end
end
```

Table 3 XOCL state change example

This example defines an operation calculateSalary in context of the type Person and calculates salary based on hours worked and a rate per hour.

2.1.9 XML MetaData interachange (XMI)

For UML tools need a standard way of persisting model objects and to exchange models between modeling tools and UML execution environments. XMI is a XML document format developed by OMG, which support a standard way of serializing objects. XMI is a widely used interchange format for sharing objects using XML. Sharing objects in XML is a comprehensive solution that builds on sharing data with XML. XMI is applicable to a wide variety of objects: analysis (UML), software (Java, C++), components (EJB, IDL, CORBA Component Model), and databases (CWM). XMI defines many of the important aspects involved in describing objects in XML. The representation of objects in terms of XML elements and attributes is the foundation. XMI is applicable to all levels of objects and Meta objects. Since objects are typically interconnected, XMI includes standard mechanisms to link objects within the same file or across files [19]. On the other hand there are some issues according to XMI versions and tool implementations. XMI was presented as "the solution" for managing interoperability between modeling tools, but since XMI do not have support for saving the position of graphical elements tool vendors have populated XMI with proprietary tags.

Using XMI is probably the best way of serializing model objects because of the number of tool vendors supporting this standard. An architecture supporting dynamic models should serialize objects due to XMI this would ease the interoperability with existing modeling tools.

2.1.10 Meta model architecture

It is important that the UML virtual machine provides an appropriate architecture to provide dynamic UML models. This architecture is defined in a Meta-Object Protocol [20]. A Meta object protocol is a set of classes and methods that allow a program to inspect the state of, and alter the behavior of its Meta model at run-time. This makes it possible to adjust the Meta modeling language to maintain different types of behaviors. For instance, changing the way that inheritance works, or modifying how the compiler works without having to change the code for the compiler. This adds further flexibility to the Meta modeling process [3].

2.1.11 Operational Semantics

This thesis use the definition of operational semantics defined in [3]:

"An operational semantics describes how models or programs written in a language can be directly executed."

This description provides the ability to express operational semantics on the language itself, although writing an interpreter for the Meta model requires that the Meta modeling language is executable. If this is the case, operations on the UML Meta model can be created and add operational behavior for making UML executable.

2.2 Approaches for executable UML models

UML models are divided into three categories.

- Object Management Group (OMG) has defined the Model Driven Architecture initiative (MDA). MDA derive computationally complete models in platform independent level (PIM), then these models are transformed to platform specific models (PSM) and finally PSM are transformed to code, which is executable [21].
- Another approach is Kennedy Carters xUML. XUML provides a process for creating the necessary PIM models. These models are populated with abstract action language expressions and then these models are either executed through simulation or as generated code [22].
- The third approach which is in an early phase is UML virtual machine. UML virtual machine may take advantage of existing virtual machine architectures, such as Java virtual machine [12] and Smalltalk virtual machine [14].

MDA as proposed by OMG is an approach for implementing systems that need to be mapped and/or integrated to different platforms. MDA applies the basic principle of separation of concerns by separating the specification of the system functionality from its implementation on a specific platform. The former is defined as a Platform Independent Model (PIM), the latter as Platform Specific Model (PSM). The mapping from PIM to PSM is performed using transformation rules.

Even if MDA is a relatively new approach there are several open source projects and commercial tools making MDA a reality for executing UML. Table 4 gives an overview of existing tools.

Commercial and open source tools	Description
OptimalJ	Product from Compuware that uses a notation of patterns to achieve PSM

	transformations. Has an integrated UML tool for analysis, but uses a slightly different notation (structural) for the MDA-part of the tool.
ArcStyler	Is a commercial MDA tool from Interactive Objects. It is bundled with MagicDraw UML tool, but can also support other UML tools through tool adaptors.
Codagen Architect	A commercial product, integrates with several commercial UML tools.
AndroMDA	An open source template based tool for J2EE code generation from UML/XMI. Uses VTL (Velocity Template Engine) as scripting language and Netbeans MDR as a model API.
OMELET	Is another, newly started (may 2004), Eclipse project, was originally part of the GMT project. Now, it aims to provide a general framework for plugging in and integrating models.
UMT (UML Model Transformation Tool)	UMT is an open source UML/XMI-based tool for model transformation and code generation purposes.

Table 4 MDA tools

Executable UML depends on abstract action language in order to create precisely defined models. Not all Executable UML tools provides an abstract action language [23], but use existing programming language such as Java or C++. In addition to action language most of the behavioral modeling is done by state diagrams. Sequence diagrams are primarily used to describe behavioral patterns of a system because they lack of the expressive power in UML 1.x to describe program flow. Table 5 list three tools supporting executable UML. Furthermore, these tools primarily use UML 1.x and take no advantage of UML 2.0 at the moment.

"Executable UML is a subset of the Unified Modeling Language incorporating a complete action language that allows system developers to build executable domain models and then use these models to produce high quality code for their target systems." [22]

Executable UML tools	Description
iUML	Product from Kennedy Carter [22] which provides a modeler and model simulator for state diagrams.
ILogix	Tool from Rhapsody which provides a Model Driven Development environment based on UML to create PIM and execute them.
Artisan Real time studio	Product from Artisan software that provides execution of state models for system behavior.

Table 5 Executable UML tools

The UML Virtual Machine approach is a new software development scheme. Developers describe software systems with UML and directly execute software models without any intermediate code generation step. There are several research projects within this field and the foundation for this work is the new UML 2.0 standard [2]. UML 2.0 includes major improvements, in addition to the alignment with MOF, the sequence diagram, has introduced control structures and may now have hierarchical structures. The activity diagram supports control and data flow with combined token flow semantics. It is now possible to model behavior of a system using interactions and/or activity diagrams, as well as the state diagrams in 1.x. Table 6 lists three projects experimenting with the new UML 2.0 standard.

UML virtual machine initiatives	Description
BabyUML	Research project started by Trygve Reenskaug [24].
Architecture of UML Virtual Machine	Project presented by Dirk Rhiele [25]
UML Virtual Machine	Research project started autumn 2004 at University in Boston [26].

Table 6 UML virtual machine projects

BabyUML [24] is a tool implemented in Squeak [13]. BabyUML compares OMGs Meta hierarchy [17] with the existing architecture in Smalltalk. These two concepts of Meta hierarchy maps perfectly within the static structure, however, modeling behavior is planned to be implemented in the future.

Another approach is the project presented by Rhiele et al [25]. This project focuses on implementing the static structure of UML 1.x within the Java virtual machine for modeling system behavior they use state diagrams. Rhiele et al implements executable UML models with static structure as classes and state machines, which are instantiated in a Java virtual machine.

UML virtual machine [26] is a new project started at the University of Boston. Within this project they plan to parse existing XMI files in order to load UML into a Java

virtual machine. Furthermore they plan to generate java byte code from UML model specification objects. All input models to UML virtual machine are described using class models and sequence diagrams.

As explained, UML virtual machine is only in the planning phase and do not provide any modeling tools or execution engine yet. In this thesis we give proposal for a UML virtual machine, UMLexe. Within the next chapters we elaborate on an executable subset of UML 2.0 extended with operational semantics.

3 Example case; Min Max Replenishment

In this chapter a case scenario representing challenges developing software systems today is described. Building software today most of the time includes interoperability between business processes and communication between these systems. This change in programming for monolithic systems towards communicating systems requires us to change the way we program as well. This case represents this challenge of communication oriented programming and how we can use components to support developing communication oriented systems

This section describes the case example, the min-max replenishment system. This problem was chosen because it is familiar and has interesting design and architectural problems according communication oriented systems. This case is used as a proof of concept for the UMLexe. The main goal is to execute Min-Max replenishment with UMLexe. Min-max case is designed with UML 2.0 and consists of a subset of UML 2.0 Meta model instances. The UML 2.0 Meta model elements are component model where interactions are mapped to component operations and specified component interactions.

Figure 3 gives an overview of the five components collaborating to fulfill a total lifecycle of replenish a car producer's warehouse. Table 7 describes the involved services in Min-Max replenishment. This scenario is further detailed in the next sections.

Service	Short name	Description
AR car manufacturer	AR	Car producer
AJB Min-Max Service	Min-Max	Replenishment service.
Philips Warehouse	Philips	Light bulb producer
XE Transport	XE	Transport service
AKBA auction service	AKBA	Auction service, representing eServices.

Table 7 Min-Max replenishment services

3.1 Min-Max replenishment

The Min-Max replenishment system represents a lifecycle for automatically handle warehouse replenishment. To fulfill this task a car manufacturers searches for Min-Max services (Min-Max) to set up a contract for car part delivery, in this case it is car lights. When a car manufacturer has found appropriate min-max service it sets up a contract for replenishment products, in this case it is car lights. When contract is registered, Min-Max service probes for current product level at car manufacturer.

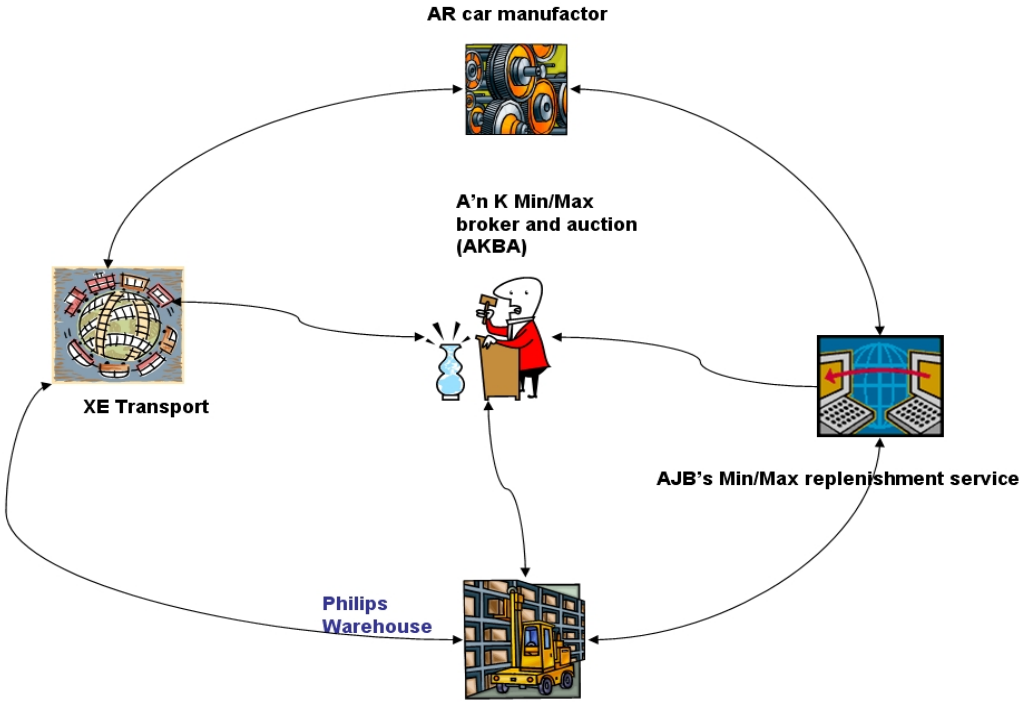

Figure 3 Min-max replenishment case scenario

If current level is out of contract range Min-Max either buys lights by contract or by auction. Buying by contract is an agreement between car manufacturer and light producer on a certain criteria such as price, guaranteed delivery in 24 hours. Buying car lights by auction is more like ad hoc shopping. Min-max service asks AKBA for the best supplier based on min-max criteria given and AKBA auction buys car lights from most appropriate supplier.

When light producers warehouse receives an order, it prepares the shipment and order transport from a transport service.

In order to transport goods from light producers to car manufacturers, an agreement with a pool of transport companies has been setup to supply the car manufacturers. This agreement gives the opportunity to buy transport by auction, but it is vital that transport service do not delay the car production process. All transport companies registers their service at AKBA. Furthermore, light producer's probes for available and most appropriate transport service. Thereafter, transport service receives the transport order and the transport company picks up delivery and transports the goods to the car manufacturer. Car manufacturer signs the freight and the transport company delivers the

receipt to the light producer. (When the light producer has received a receipt, they send invoice to the car manufacturer).

3.1.1 Platform independent description

One of the main goals of this case scenario is to develop a platform independent model representing the Min-Max system. This model consists of components, port types, interfaces and interaction diagrams. These diagrams are described in the next sections.

Figure 4 describes the main view of the Min-Max replenishment system. This view merges declarations from port types and interfaces which is specified in class diagrams. As described, each component represents corresponding business or service presented in Figure 3. Operations on each component are necessary for the communication between all parts to fulfill the task of replenish a warehouse.

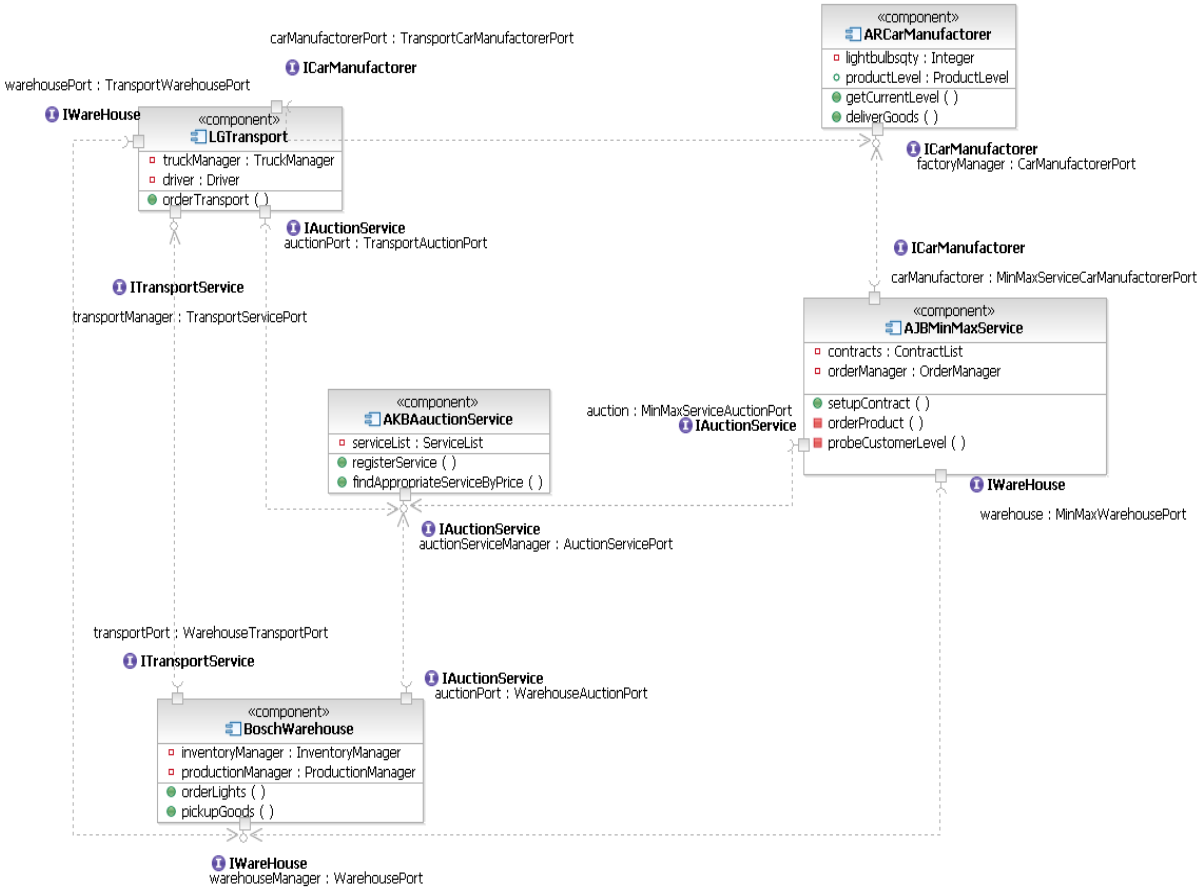

Figure 4 Min-max components

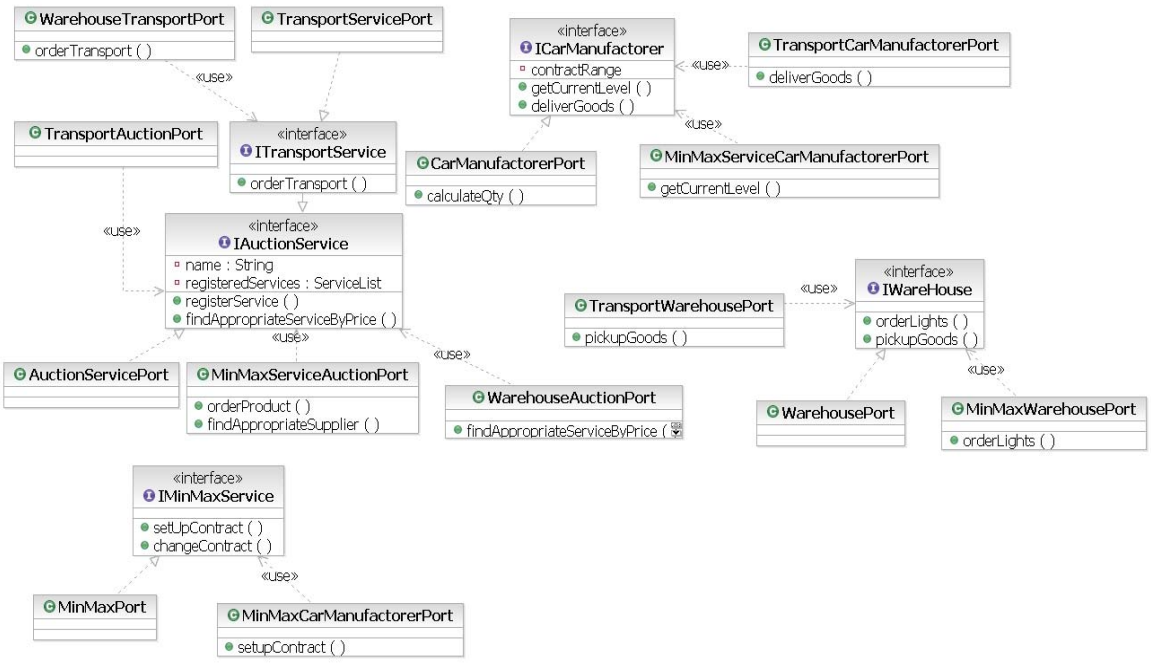

Figure 5 Min-Max replenishment port types and interfaces

All ports on components described in Figure 4 are defined by some class and their interfaces, which in this case are described in Figure 5. As discussed in the next chapter, ports define a communication point and within the context of UMLexe these ports represents input ports. As described in Figure 5 output ports are classes without any operations. This indicates that output port is a reference to a collaborating object.

In order to specify a component operation, interaction diagram is used. For describing the concept only two interactions is provided. The first operation described is "findAppropriateServiceByPrice". Figure 6 illustrates the interaction between a port and the internals parts in AKBA. AKBA contain a list of services which holds a reference to all services registered with the auction. When a car manufacturer or a warehouse needs a service, they ask the auction to return the most appropriate supplier. This is done by calling the "findAppropriateServiceByPrice" operation. This operation is specified by the interaction "findAppropriateServiceByPrice".

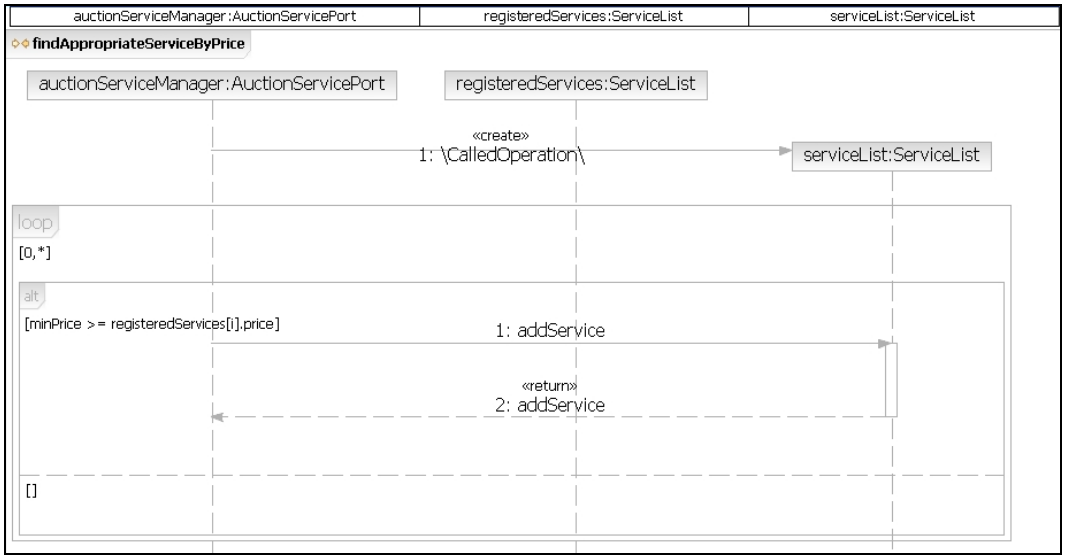

Figure 6 Find appropriate service

This interaction has one parameter "name" which defines what kind of service to look for and it returns a list of services. In most cases there will only be one return value, but if there is more than one service providing the service at the same price, a list is returned. Furthermore, the interaction has three roles collaborating to fulfill the task, auctionServiceManager, registeredServices and serviceList. All of these roles must be defined in the component. The auctionServiceManager is defined as a port in the component. Ports that implement a provided interface acts as a controller objects linking the external interface with the internal parts in a component. On the other hand, a port which is dependent of other components or interfaces is used as a property where we are able to wire components together. Finally, this interaction loops through all services registered and checks their offers for this kind of light bulbs and adds all services to the service list with the lowest price on top.

The other operation described is probe customer level, which is defined in the context of Min-Max component. Probe current level asks the AR service of the current level of light bulbs. This describes an interaction with a required object as described in Figure 7. This operation aims at asserting a promised level of light bulbs for the given car producer.

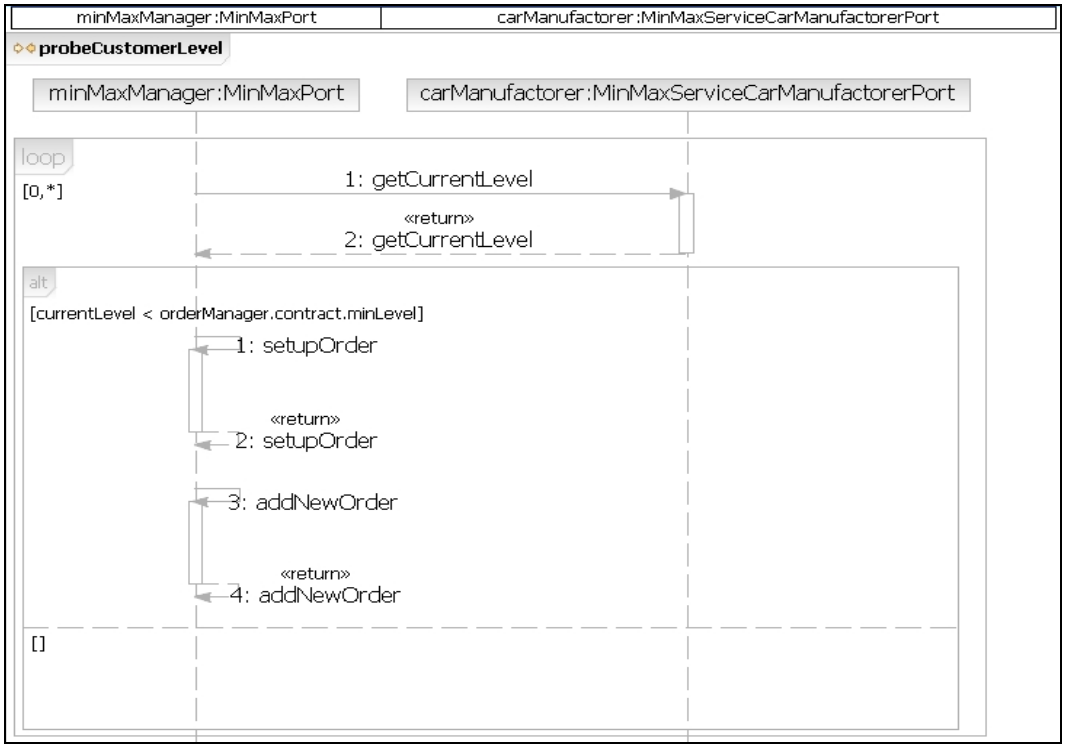

Figure 7 Probe for current level

3.2 Summary

The purpose of this case is to provide a foundation for evaluating existing tools and to execute this case in UMLexe. When evaluating existing tools, this case will be modelled in each tool in order to verify that the tool support the requirements specified in Chapter 4. Furthermore, this cased will be used during the proof of concept chapter for executing and verifying that UMLexe provide a contribution to the field of UML virtual machine.

4 Requirements for UML virtual machine

The goal of this chapter is to describe requirements identified for executing models. These requirements constitute a foundation for comparison of systems aimed at executing UML models. This chapter elaborates requirements for dynamic UML models. Chapter 5 gives an evaluation of existing projects and modeling software tools against these requirements.

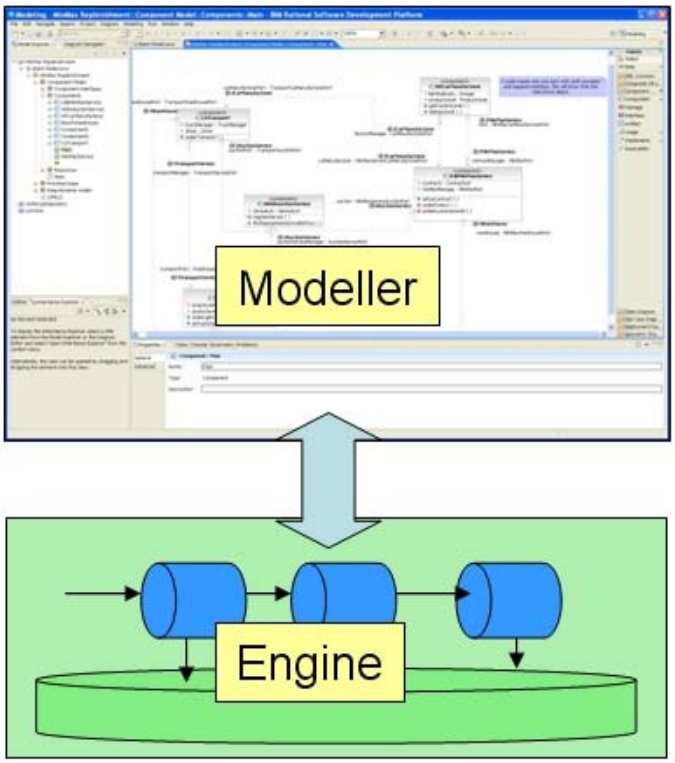

Figure 8 UML virtual machine parts

Figure 8 illustrate two components evaluated in each tool. These two components provide various supports for UML functionality. Therefore a distinction between modeler and engine is necessary. Modeler represents the graphical environment for creating UML models and the engine represents the executable environment for executing these models. In order to create a UML virtual machine nine requirements is declared. These requirements are given identifiers from R1 to R9. Requirement from R1 to R8 are evaluated against both modeler and engine and requirement R9 is fulfilled if requirements R1 to R8 are satisfied.

4.1 UML virtual machine requirements

As described in Chapter 2.1 a UML virtual machine is a program representing an abstract machine for executing UML models. For execution models, nine requirements are declared. Table 8 gives an overview of requirements necessary when dealing with dynamic UML models.

ID	Property	Description
R1	MOF 2.0 support	Support for MOF 2.0 Meta models as part of realizing the reflective architecture.
R2	UML 2.0 support	Support for a minimum set of UML 2.0 syntax and semantics.
R3	XMI 2.0 support	Support for importing and exporting XMI 2.0 UML models.
R4	Support for Modeling tool integration.	Provide application programming interface for interacting with the modeling tool environment.
R5	Support for OCL	Support for handling model constraints with OCL.
R6	Support for abstract action language	In order to precisely define executable UML platform independent models.
R7	Model validation	Support for validation models against UML 2.0 Meta model.
R8	Model execution	Support for executing a UML 2.0 model.
R9	Architecture supporting dynamic models	Support for any changes at each level will be reflected on the lower level at runtime. Requires acceptable support for R1 to R8.

Table 8 UML virtual machine requirements

4.1.1 MOF 2.0 support (R1)

As described in Chapter 2.1, to create dynamic models an open reflective architecture is essential. In the context of dynamic models, a UML virtual machine must support MOF 2.0. MOF provide Meta objects for UML 2.0 and functionality to navigate these models. If the UML virtual machine has support for MOF then it is possible to manipulate elements at the M2 level. This will open the possibilities for creating a dynamic modeling environment not only for UML, but other modeling languages based on MOF as well.

4.1.2 UML 2.0 support (R2)

This requirement declares the minimum set of UML 2.0 elements that ought to be covered in a modeling tool for executing operational behavior within UMLexe.
The goal of this requirement is to assure UML 2.0 support. This UML support must cover both static structure and operational behavior. The minimum is defined to be classifier, class, component, operation, port, interface, interaction, instance specification, instance value and slot. Figure 9 gives an overview of static structure elements declared. Furthermore, supporting operational behavior, a modeling tool should support a minimum set of behavioral elements as well.

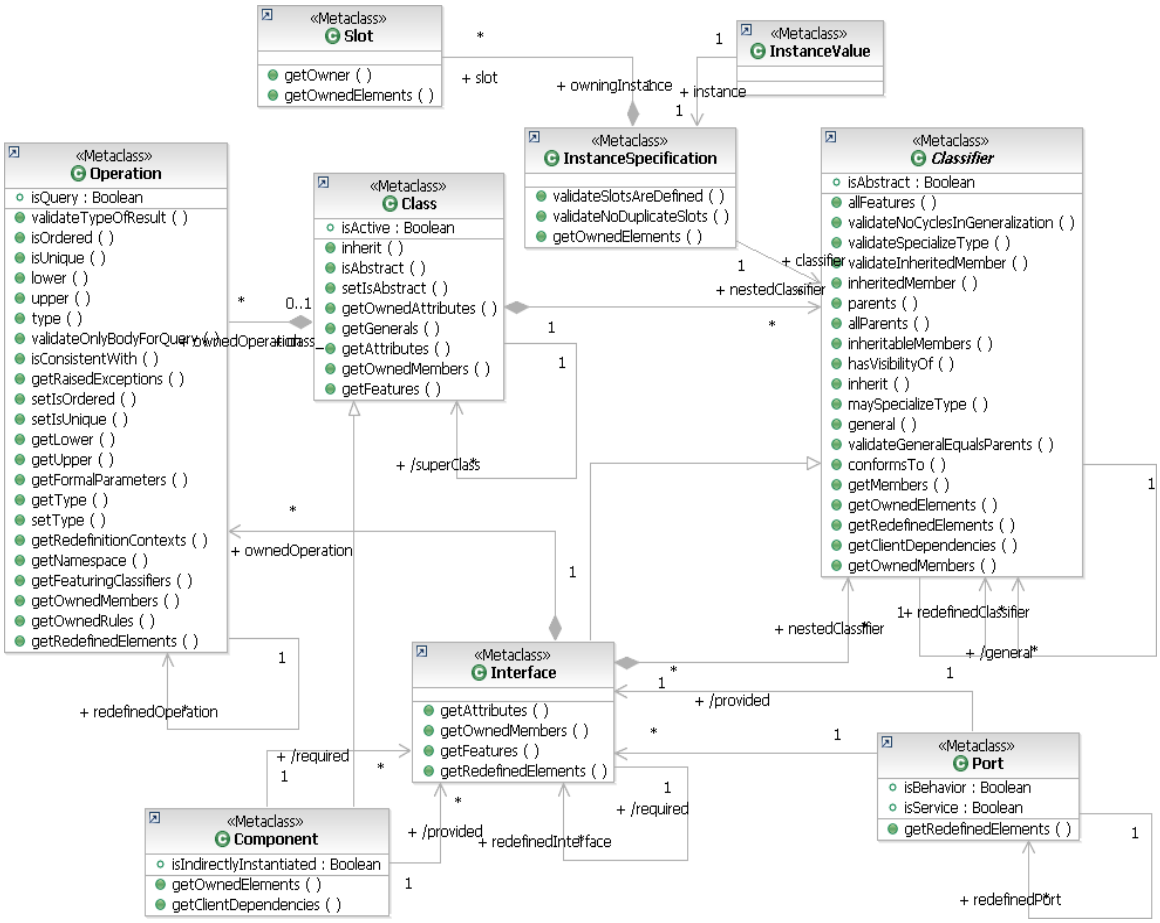

Figure 9 UML 2.0 structural subset

The main focus is the support for components, ports and interfaces. These are necessary to combine the compositional elements and take advantage of the new feature in UML 2.0. Especially is the combination of required and provided interfaces with ports. Since components derive from class it is possible to take advantage of operations linked to a component. This is valuable for specifying operations on components. The port element is a new interesting feature in UML 2.0 for specifying roles a component may play together with their implementation type.

The static structure combined with behavioral specifications such as interaction can be combined to create an executable subset of UML. The behavioral subset which is the main focus within this thesis is interactions. The behavioral subset is described next.

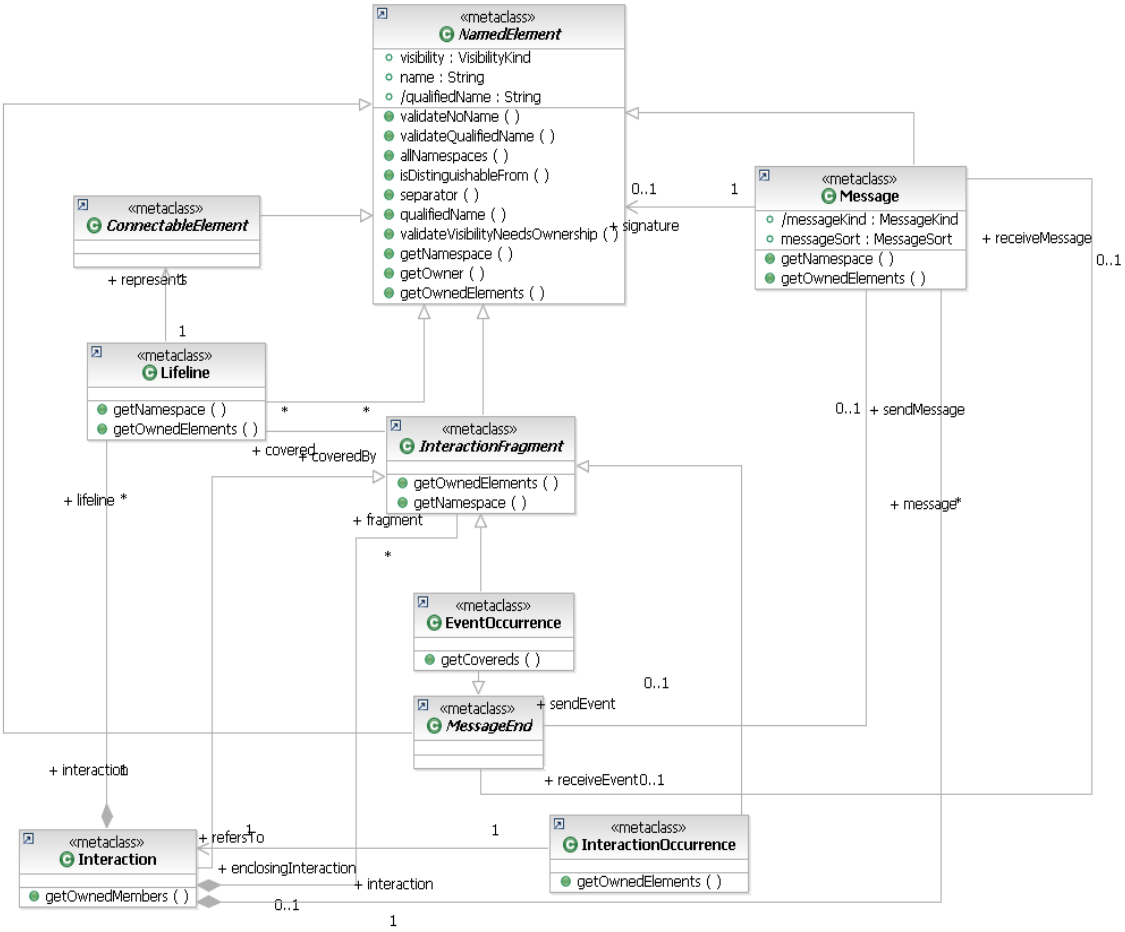

Figure 10 UML 2.0 behavioral subset

Figure 10 presents the essential elements for describing a components behavior. Using interactions give the possibility to specify components operations and the internal collaboration. The interaction contains lifelines, which represents properties, attributes, ports in a component. Specifying operation invocations is achieved with messages and event occurrences. The event occurrences guaranty that a send event happen before an receive event. Furthermore, there is a new feature in UML 2.0 that allow us to model interaction fragments with operators. This gives a greater potential to model more flow within interactions than it was possible with sequence diagrams.

One of the goals of this thesis is to create executable subset of UML and then it is essential to combine both static structure and behavior. Within this thesis the focus on linking operations to interactions and this is described next.

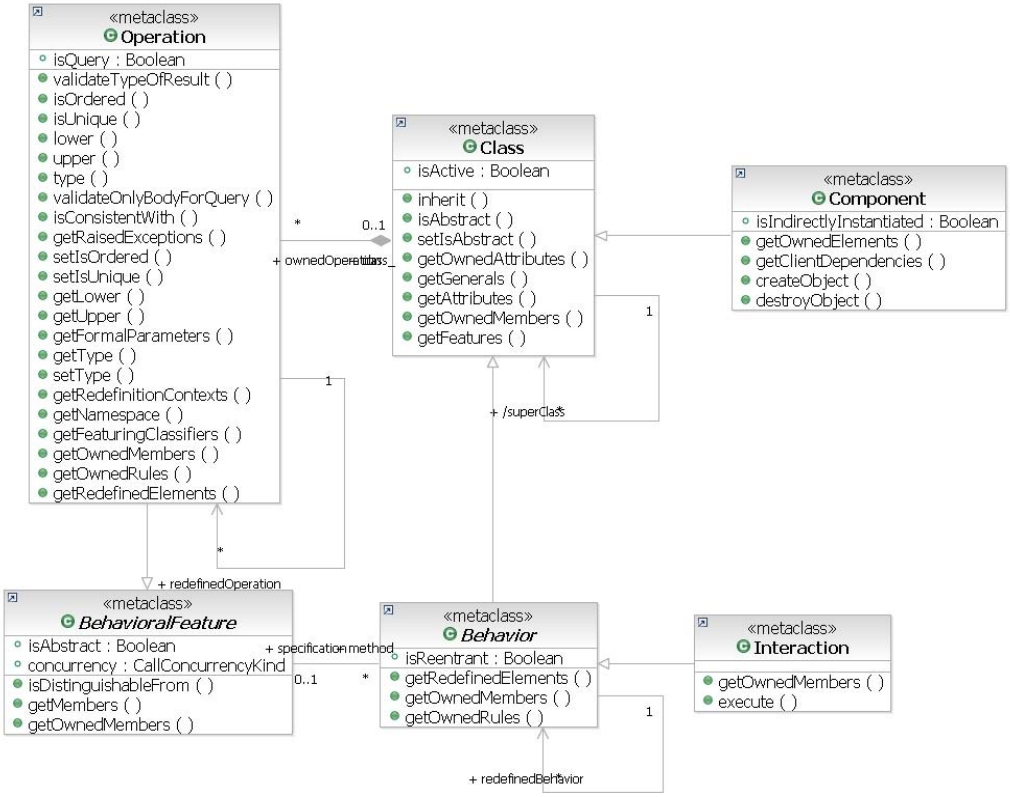

Figure 11 Combining static structure and behavior

Figure 11 describes how an association between operations and interactions is achieved. This figure shows that the interaction is connected to operations through behavior and the specification property within operation is used to let interactions specify operations on components.

When both static structure and behavior is identified, the next step is to identify how the runtime instances should be represented. This thesis has focused on using the existing UML 2.0 Meta model and therefore the instance specification elements are used. The subset of the runtime instances is described next

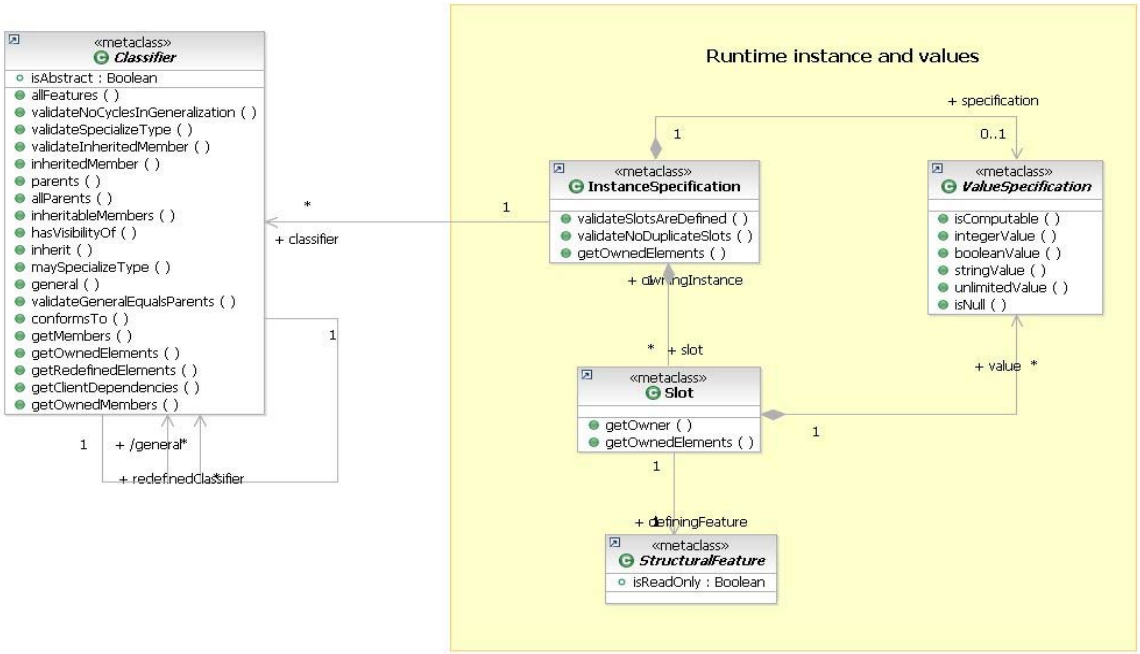

Figure 12 Runtime instances

One aspect in a UML virtual machine is to create instances. Our approach is to let instance specifications represent runtime objects. Figure 12 describe the association between a classifier and an instance specification. This link is important to maintain the reflective architecture described in Chapter 2.1.10.

4.1.3 XMI 2.0 support (R3)

A UML virtual machine must have support for XMI 2.0 to provide an open architecture in terms of interoperability. The XMI standard is well suited for representing the content of object oriented models, which are expressed in UML. XMI is based on MOF, which also is the base for UML. This fact enables the possibility to represent all types of UML elements in XMI and this will ease the serialization of UML objects and interchanging objects between dynamic modeling tools.

4.1.4 Support for modeling tool integration (R4)

This requirement is necessary for an engine to interact with existing modeling tools. Both engine and modeler should provide interface for other tools to connect to. The engine should provide interface for monitoring and deploying models. The modeler should provide interface for engine to update or notify which runtime instances is active. This requirement will provide an open architecture for a UML virtual machine.

4.1.5 Support for OCL (R5)

A constraint is a restriction on one or more values of a model. A UML virtual machine must provide evaluation of OCL statements. As described in Chapter 2.1.7 XOCL can be used to handle operational semantics. So a UML virtual machine must either provide executable OCL statements or add functionality to evaluate abstract action language expressions.

4.1.6 Support for abstract action language (R6)

Executing UML models is not possible without support from an action language. This formalism could be provided by for instance by Java or C#, but an abstract language is preferable because it maintains a platform independent description of behavior in the software system.

4.1.7 Model validation against the UML Meta model (R7)

In order to execute a UML model it is a necessary feature to validate models according to the UML Meta model. This feature ensures that the input model is valid according to the UML 2.0 standard.

4.1.8 Model execution (R8)

This requirement evaluates if model execution is provided. If a tool supports model execution then this requirement is extended to executing models containing components and interactions according to the UML 2.0 standard.

4.1.9 Architecture supporting executable UML 2.0 models (R9)

One of the key aspects of creating a virtual machine for dynamic UML models is the opportunity to provide an open architecture; it should have rich user and programmer interaction, support for Meta objects and ability to reflect changes in Meta objects. The fulfillment of these requirements is partly implied by the fulfillment of R1-R8. A detailing of this requirement is summarized below:

- **Open**: To remain useful, dynamic UML environment, like all software, must evolve. It must be relatively easy to create new system functions and integrate new services. Since a system provides no distinction between the levels of protection for systems and users, we can easily use well-designed system objects [27],[9].

- **Rich user and programmer interaction:** Users should be largely unconstrained with respect to how they interact with the system. For instance, they should not be constrained to a particular style of input/output device or to a particular user interface style (such as UML diagrams). Similarly, programmers and modelers should be allowed to manipulate objects in the system naturally and efficiently.

- **Meta objects:** Each object resides in the context of a collection of meta-objects to handle dynamic system behavior, to deal with transparency, to reduce constraints about distribution; these Meta objects define the environment for computation and constitute a meta-space.

- **Reflection:** To provide an open and self-advancing environment, a dynamic UML environment must provide reflective computing that presents facilities for self-modifying an object with its environment and for inspecting the meta-computing environment of an object.

4.1.10 Summary

As described in this chapter, all requirements declared is essential for creating a UML virtual machine that provides execution of UML models and even makes them interactive for the user. These requirements are derived from studying existing tools for modeling and executing UML models. Furthermore, another important aspect is the study of existing architectures that provide reflective concepts. All requirements described in previous sections are used to evaluate existing tools and solutions for executing UML models. The result of this evaluation is used to derive requirements, design and implementation for a new UML virtual machine.

The evaluation of existing solutions for modeling and executing UML models are described in the following section. This evaluation contains both existing modeling tools and tools which executes UML models.

5 Evaluation of existing tools

This chapter presents the evaluation of existing tools. These tools are based on solutions for modeling and executing UML models. The evaluation is based on the requirements given in the previous chapter. Within this evaluation, the population of tools is grouped into three categories;

- Modeling tools with UML 2.0 support.
- Executable UML tools.
- UML virtual machine.

Modeling tools is chosen because none of the existing tools that execute UML models support UML 2.0 notation. However, as described in the next sections, tools that claim support for UML 2.0 either implements graphical notation on top of existing Meta model or they do not implement all aspects of the standard. In the next category, executable UML, the focus is on the execution engine and how UML is made executable. Tools within this group are the only alternatives today which execute UML models. This approach mainly use the UML 1.x standard and use state machine models with an abstract action language for providing executable UML models. The final group of tools evaluated is UML virtual machine. As mentioned in the introduction this concept is in the early conceptual phase. However, the main focus of this thesis is UML virtual machine for creating dynamic models. One tool which describes a conceptual solution based on Smalltalk is evaluated.

The evaluation is based on the categories identified, the requirements described in the previous chapter, and the Min-max example case as input. For each tool evaluated, an attempt to create models for the Min-Max is done based on the subset of UML 2.0 described in previous chapter. Furthermore, if it is possible to create these models then an attempt to execute those models is accomplished.

All tools are downloaded from internet either as an open source license or free evaluation license. In order to get even more information from this evaluation it was desirable to evaluate more tools in the category of executable UML tools, but these tool vendors would not let me evaluate their tools.

The first part of this chapter evaluates three tools in the modeling tool category, Borland Together, Enterprise Architect and Rational Software Modeler. Then an executable UML, Kennedy Carters iUML, tool is evaluated and finally a UML virtual machine concept, BabyUML, is evaluated. This evaluation gave a valuable input to the specification and implementation of UMLexe.

5.1 Borland Together

Borland Together is developed by Borland Together Technologies [28]. Borland has recently bought Together to supplement their product portfolio. Furthermore, Borland is a commercial product and claims that they are committed to follow the standards for modeling. In terms of Together as a UML modeling tool, they provided early a reflective view of class models into Java code and that is why this tool is involved together in this evaluation. The product which is evaluated is the Borland Together Eclipse version.

By releasing this new modeling tool Borland focused on supporting Model Driven Development (MDD) and the concepts of MDA. This gives them the possibility to execute UML models by transforming them to code. The next section present and validate this product against the requirements for supporting dynamic UML models.

5.1.1 MOF 2.0 support (R1)

The modeler in Together is based on UML models from Java Meta Interface (JMI) [29]. JMI is a Java based implementation of MOF [30], but it is not possible to change or interact with this modeling level. Together do not provide execution of UML models and therefore it is not necessary to evaluate these requirements for the engine.

5.1.2 UML 2.0 support (R2)

Together modeler partly support the minimum subset of UML described in Chapter 4.1.2. Together implements components, ports, connector and interfaces, but it is not possible to add operations to components. There is full support for classes and finally there is support for interactions, but adding gates on interactions is not possible. Therefore is the support for UML 2.0 Meta model is partly implemented.

Components are partly implemented, as it is not possible to add properties and operations to components, and it seems that the UML Meta model is not totally supported. Since a component is a class it should be possible to add operations and properties to a component according to the UML Meta model. Port is a property and should be added to a component as a property. Internal elements are properties and are added to a component as part or attribute. It is possible set types on ports, but it is not possible to connect ports as properties to components. Furthermore, Connector is not supported and therefore it is not possible to create links between components. Interface is supported and it is possible to add these as types to ports. Operation is supported as part of class and interfaces, but as mentioned above it is not possible to add operations to components. In addition, it is not possible to add a specification in form of interactions or activities to operations. According to the UML 2.0 Meta model [2] this should be possible. In order to create data types and value objects classes and structure associations are supported.

Interactions are partly implemented. For instance it is not possible to let lifelines represent properties, meaning it is not possible to set parts or ports to represent a lifeline. On the other hand, it is possible to let the lifeline be represented by a class. Furthermore, a lifeline can be decomposed and represent a part in another lifeline. The vital specification of gates in order to tie model specification together is not supported.

Another new feature of interactions is combined fragment. This feature is partly supported. In fact, as a result of not supporting gates we are not able send messages to and from combined fragments.

As we have seen, there are still a lot to come from together modeler in order to support UML 2.0 and the minimum subset defined in Chapter 4.1.2.

5.1.3 XMI 2.0 support (R3)

Together do not support XMI import and export for XMI 2.0. First of all this tool has support for XMI export for version 1.3, 1.4. Second, Together only allow import for XMI version 1.1 and 1.2. Interoperability with XMI 2.0 is not supported.

5.1.4 Support for modeling tool integration (R4)

Supporting tool integration should be provided by both the modeler and the engine component. The modeler should provide an interface in order to add extensions like for instance an engine. Then, if an engine is provided it should be possible to integrate with a modeler. The modeler in Together does not provide interfaces to provide extension points.

5.1.5 Support for OCL (R5)

Defining constraints on models can be done either by OCL or an abstract action language. An environment for supporting dynamic models must have support for managing executable OCL expressions. The modeler in Together provides support for adding OCL expressions as text strings. Then the modeler have provides functionality for validating these expressions.

5.1.6 Support for abstract action language (R6)

Together modeler provides functionality for adding text strings in order to provide more specifications to a model. This text string can be used to add abstract action language specific expressions. It is then up the modeler to use the right syntax. The modeler does not provide any functionality to validate or execute these expressions.

5.1.7 Model validation (R7)

Together modeler provides functionality to validate models against the UML Meta model, but there are some considerations. First of all, this functionality only validates against the Meta model implemented in Together and as we have seen this only provide support for parts of the UML 2.0 Meta model. As a result, the model validation is only partly supported.

5.1.8 Model reflection (R8)

Borland Together do not provide any functionality to execute or simulate models. First of all, the modeler must provide the ability to create models according the minimum set of UML elements described in Chapter 4.1.2. It must be possible to validate the models and finally there must be an engine, with the possibility of executing these models. Obviously it is not possible to create dynamic UML models with the missing support of the UML 2.0 meta model and this tool does not provide an engine or extension point for adding an engine in order to executing these models.

5.1.9 Architecture supporting dynamic UML 2.0 models (R9)

As mentioned earlier this requirement is fulfilled when requirements R1-R8 is supported by both the modeler and the engine. The architecture supporting dynamic UML 2.0 models is not supported by Borland Together. First of all, Together only provides a modeler and then this modeler only provides support to some of the requirements specified. Second, this tool is proprietary and provides no extension points for third party developers to interact with this tool. As we have seen Borland Together do not have an architecture that supports dynamic UML models.

Pros: Easy and intuitive modeler.
Cons: Lacks of support for UML 2.0 Meta model, proprietary modeler.

5.2 Enterprise Architect

Enterprise Architect (EA) is developed by Sparx systems. EA is intended for use by analysts, designers, architects, developers, testers, project managers and maintenance staff, almost everyone involved in a software development project and in business analysis. It is Sparx Systems belief that highly priced CASE tools severely limit their usefulness in a team, and ultimately to an organization, by narrowing the effective user base and restricting easy access to the model and the development tool.

EA is one of few tools that claims support for UML 2.0. Within this section this tool is validated against the requirements described in Chapter 4 EA mainly focus on the modeler part of UML and do not provide the possibility for executing UML models.

5.2.1 MOF 2.0 support (R1)

One of the criteria for supporting a reflective architecture there must be support for MOF 2.0. MOF 2.0 is not provided by EA. At the moment, EA do not have support for transforming models based on MOF, but provides code generation.

5.2.2 UML 2.0 support (R2)

EA do not support the UML 2.0 Meta model. This was tested by modeling a component structure with ports and role bindings, and trying to link component operations with behavior.

Components are provided and it is possible to add operations to components. EA supports the ability to add ports as properties to a component and it is possible to add provided or required interfaces. In order to link static structure and behavior an attempt to add interaction specification to operations is tried. There is no support for adding parameters and return values to interactions in EA. Interactions is partly implemented. It is not possible to let lifelines represent properties, meaning it is not possible to set parts, ports to represent a lifeline. On the other hand, it is possible to let the lifeline be represented by a class. Furthermore, a lifeline can not be decomposed to represent a part in another lifeline.

The vital specification of gates in order to tie model specification together is supported. Combined fragment is supported but as a result of not supporting gates we are not able send messages to and from combined fragments, which means models are not precisely enough defined.

Operation is supported as part of components and interfaces, but it is not possible to link messages in an interaction to operations on lifeline. This makes sense since it is not possible to let classifiers to be lifelines.

5.2.3 XMI 2.0 support (R3)

In order to validate the support for XMI 2.0 support I looked for this feature in the feature documentation and I tried to import the Min Max XMI 2.0 file. Enterprise Architect does not support XMI 2.0. EA only support XMI 1.1 and 1.2 at the moment and the XMI file is populated with proprietary tags. This makes it difficult to exchange models with this tool.

5.2.4 Support for modeling tool integration (R4)

The only possibility to integrate with this tool is to exchange XMI files. As mentioned earlier this tool only supports XMI 1.1 and XMI 1.2 with proprietary tags.

5.2.5 Support for OCL (R5)

The requirement for OCL support is not provided in EA. It will still be possible to add OCL statements as text strings in for instance combined fragment guards, but there will be no validation of these statements.

5.2.6 Support for abstract action language (R6)

EA modeler provides functionality for adding text strings in order to provide more specifications to a model. This text string can be used to add abstract action language specific expressions. It is then up the modeler to use the right syntax. The modeler does not provide any functionality to validate or execute these expressions.

5.2.7 Model validation (R7)

Model validation is a requirement for make sure that the model is created according to the UML Meta model. EA do not support any kind of validation of the created model.

5.2.8 Model interpretation (R8)

The final requirement is the possibility to interpret UML models. This requirement is not supported by EA. EA is a UML tool only supporting drawing graphical UML model elements. There is no support for UML 2.0 model elements according to the UML 2.0 Meta model.

5.2.9 Architecture supporting dynamic models (R9)

The architecture supporting dynamic UML 2.0 models is not supported by EA. First of all, EA only provides a modeler and then this modeler, which only provides support for some of the requirements specified. Second, it seems that this tool mainly focus drawing the graphical notation for UML 2.0. The Meta model for 2.0 is not supported at all. This tool is a commercial and provides no extension points for third party developers to interact with this tool. Consequently, EA does not have an architecture that supports dynamic UML models.

Pros: Easy and nice implementation for ports, Easy to create UML elements.
Cons: No integration, no execution, no action language, no support for OCL.

5.3 Rational Software Modeler

Rational Software Modeler (RSM) is a new product from Rational that is UML 2.0 compliant and is based on the Eclipse platform. RSM uses the UML 2.0 project and EMF as constituent parts of its implementation. RSM does not ship as a plug-in for Eclipse but comes as a complete application containing the Eclipse platform. It is probable that all existing plug-ins will function in RSM. One can write own plug-ins for RSM using standard Eclipse technology and the documented APIs for the RSM.

5.3.1 MOF 2.0 support (R1)

RSM supports the MOF 2.0 by including the EMF [31] project into their tool. This gives the possibility to extend the modeling tool with ecore models. Ecore models are models based on MOF Meta model and give the opportunity to extend the modeling environment.

5.3.2 UML 2.0 support (R2)

RSM covers UML 2.0 supporting use case, class, component, deployment, interactions, activity, and composite and state machine diagrams. In the evaluation necessary structure elements is added which is defined in the minimum subset of UML 2.0. Then interactions are added and finally a link between a components operation and an interaction is added. The findings show that RSM has the potential to create UML 2.0 Models that are executable.

Components are supported and it is possible to add operations to components. RSM supports the ability to add ports as property to a component and it is possible to add provided or required interfaces. This feature is very user friendly. An extra class diagram with interfaces and classes is added, that either uses or implements this interface. In order to link static structure and behavior interaction specification is added to operations. This was nicely supported by right clicking an operation and only selects the interaction. Interactions are partly implemented, because it is not possible to add gates to an interaction. Another missing feature is the possibility to decompose a lifeline.

RSM supports the UML 2.0 Meta model, but are dependent of the UML2 [32] project. At the moment, UML2 have implemented the current status of the UML 2.0 Meta model specification.

5.3.3 XMI 2.0 support (R3)

RSM supports interoperability by using the UML2 XMI implementation. UML2 supports the XMI 2.0 specification without tagging their XMI files.

5.3.4 Support for modeling tool integration (R4)

Modeling tool integration is supported either by using XMI 2.0 or by the provided UML2 and EMF. RSM do not provide an interface for interacting with the graphical elements in a model.

5.3.5 Support for OCL (R5)

The requirement for OCL support is not provided in RSM. It will still be possible to add OCL statements as text strings in for instance combined fragment guards, but there will be no validation of these statements.

5.3.6 Support for abstract action language (R6)

RSM do not provide an abstract action language for evaluating executable expressions on models, but it is possible write text strings which can contain action language syntax.

5.3.7 Model validation (R7)

Model validation is a requirement to make sure that the model is created according to the UML Meta model. RSM supports this feature by taking advantage of the UML2 Meta model. The model validation operation recursively traverse the model and makes sure it is valid according to the UML 2.0 Meta model.

5.3.8 Model interpretation (R8)

RSM do not provide functionality in order to execute or interpret UML models, but it is potentially the best UML 2.0 tool at the moment for adding support for interpreting UML models. This can be interpreter can be implemented as a plug-in to the Eclipse platform and interacting with the UML 2.0 model

5.3.9 Architecture supporting dynamic models (R9)

RSM take advantage of the Eclipse platform and the existing plug-ins for EMF [31] and UML2 [32].By using these plug-ins RSM implicit supports for the OMG's meta hierarchy, meaning, that there will be a causal connection between a ecore meta object and a UML meta object. Furthermore, there is no support for creating runtime instances from the UML class or component objects.

Pros: Support for MOF, UML 2.0 Meta model and graphical notation.
Cons: No execution, difficult to add ports and specifying provided and required interfaces

5.4 Kennedy Carter- iUML lite

iUML is part of Kennedy Carters product suite. This tool is chosen because of the ability to execute UML models. iUML is tailored to the needs of real projects and it provides support for the xUML formalism [16], including a fully featured Action Language as well as support for model execution, test and debug. iUML is a multi user application development environment that delivers sophisticated support for Executable UML modeling, simulation and code generation [22].

5.4.1 MOF 2.0 support (R1)

One of the most important criteria for supporting a reflective architecture is the support for MOF 2.0. MOF 2.0 is not supported by iUML. This makes sense because they have focused on creating their own Meta model for supporting a subset of UML 1.4.

5.4.2 UML 2.0 support (R2)

The modeler in iUML is proprietary and depends on the UML 1.4 Meta model they chose to implement. iUML do not support UML 2.0 graphical notation except the elements that is equal in UML 1.4 and 2.0. As we have seen there is no support for the UML 2.0 minimum subset described in Chapter 4.1. Furthermore, the main focus in iUML specifying behavior with state machines in order to execute their models.

The execution engine provided in iUML is tight coupled to the modeler and provides only support for executing the UML 1.4 subset provided as the graphical notation.

5.4.3 XMI 2.0 support (R3)

Modeling tools should have the possibility to import and export models in order to exchange models. The preferred file format is XMI 2.0 and the modeler in iUML has no possibility to export or import XMI 2.0 files. Furthermore, the execution engine provided in iUML have do not support import or export of XMI 2.0.

5.4.4 Support for tool integration (R4)

This requirement should be fulfilled in order to either change the execution engine or the modeling tool. The modeling tool do not provide interface for third party developers to interact with the graphical diagrams. In addition, the execution as mentioned earlier is tightly coupled to the modeling environment and does not provide interface either.

5.4.5 Support for OCL (R5)

iUML use their own action language to query and navigate UML models. This action language is used instead of OCL. This action language is used in the modeler as text strings and is interpreted in the engine. There is no support for OCL.

5.4.6 Support for Action language (R6)

One of the main advantages in iUML is the support for an action language. This action language is used in both the modeler and is interpreted in the engine. It is the main resource for iUML to execute UML models.

5.4.7 Model validation (R7)

A modeling tool should provide support for validating models against the UML 2.0 Meta model and as we have seen iUML do not support UML 2.0 at any level. On the other hand for iUML to execute their UML models these 1.4 models are validated against the subset of UML 1.4 before execution.

5.4.8 Model execution (R8)

One of the main criteria for evaluating this tool was the ability to execute UML models. iUML provide an execution engine which is proprietary the tool. This engine executes models based on the action language they have created. In addition, the engine in iUML provides a translator which creates code.

5.4.9 Architecture supporting dynamic models (R9)

The last requirement in order to provide dynamic UML 2.0 models is the architecture. Based on the requirements {r1...r8} evaluated, iUML has no support for creating dynamic UML models. Both the modeling tool and engine fails to fulfill all requirements. As we have seen this architecture is not provided.

Pros: iUML uses an abstract action language, provides and execution engine and simulation as well.
Cons: iUML do not support UML 2.0, have a proprietary modeler and do not provide integration possibilities.

5.5 BabyUML

Another approach to creating dynamic UML models is the BabyUML project [24]. This is an ongoing project created by Trygve Reenskaug and that focuses on the problems with today's programming languages. This is a bottom up approach for raising the level of abstraction, where the underlying runtime engine is implemented in Squeak [13]. Reenskaug is in the finalization of merging Smalltalk and MOF and UML 2.0 Meta model. This is a completely different approach than existing modeling tools used and this project is evaluated against the requirements in order to see how this project provides support for dynamic UML models.

This tool does not provide a modeler, but adds valuable considerations to an execution engine.

5.5.1 MOF 2.0 support (1)

The released version of BabyUML has not implemented support for MOF 2.0, but as mentioned above the next release will implement the MOF 2.0 Meta model as a new root package in Squeak. At the moment there is no support for MOF 2.0.

5.5.2 UML 2.0 support (R2)

The released version of BabyUML has not implemented support for UML 2.0, but as mentioned above the next release implements the UML 2.0 Meta model as the M2 level in OMG's Meta model. At the moment there is no support for UML 2.0. Furthermore, it is not possible to create UML diagrams or graphical elements. As mentioned earlier BabyUML is not a modeling tool, but an environment for executing UML models. A major improvement would be a modeling tool supporting the UML 2.0 syntax.

5.5.3 XMI 2.0 support (R3)

For future implementations of BabyUML there should be provided functionality for exchanging UML models between runtime environments. At the moment BabyUML has no support for XMI 2.0. Since XMI 2.0 is a Meta model based on the MOF Meta model, it could be implemented as M2 level package in Squeak.

5.5.4 Support for modeling tool integration (R4)

Modeling tool integration is not supported. BabyUML do not provide an interface for interacting with the graphical elements in a model or the possibility to exchange models by using XMI.

5.5.5 Support for OCL (R5)

In order to provide support for the UML 2.0 Meta model, BabyUML should implement support for OCL. In this released version the requirement for OCL support is not provided in BabyUML.

5.5.6 Support for abstract Action Language (R6)

BabyUML uses Smalltalk as the execution language and do not provide support for an abstract action language.

5.5.7 Model validation (R7)

Model validation is a requirement for making sure the model is created according to the UML Meta model. As a consequence of not implemented the UML 2.0 Meta model there is support for validation of UML 2.0 models.

5.5.8 Model interpretation (R8)

The open architecture of BabyUML gives the possibility to change or create new system objects like interpreter, object memory and stack for supporting another language like UML 2.0. BabyUML will in a future release provide functionality to interpret UML models based on components and collaborations.

5.5.9 Architecture supporting dynamic models (R9)

BabyUML take advantage of the Squeak environment and therefore provides an open architecture. Squeak is a result of an open source project [13]. Furthermore, Squeak is based on the Smalltalk environment where everything is an object, even the language elements are objects. This model gives an architecture that maps to the OMG's meta hierarchy [9], this makes sense since this builds on the ideas of Meta Object Protocol [20]. Finally, BabyUML provides the possibility to create runtime instances at the M0 level and even reflect the changes from M1 level to the M0 level. This architecture supports the possibility to create dynamic UML models.

Pros: Support architecture for dynamic models, has an execution engine
Cons: No modeler.

5.6 Summary of evaluations

The evaluation of these categories of UML tools is summarized in Table 9. In general, the only tool that has support for a dynamic architecture is BabyUML, but this tool lacks UML specific implementations like MOF Meta model, UML 2.0 Meta model and graphical modeling environment. The modeling tools that have support for UML 2.0 have only partly implemented UML 2.0 Meta model. According to the evaluation the best tool for supporting the minimum UML Meta model is Rational Software Modeler, but there is no tool that has support for executing an UML model based on the minimum subset defined in Chapter 4.1.2. The best tool for executing an UML model is BabyUML and iUML. Finally, there is no tool that supports both a modeling environment and an execution environment for UML 2.0. A tool for the future should provide a modeling environment which has support for both modeling and execution of UML 2.0 models. Furthermore, all existing tools that support execution of UML models only take advantage of state machine models for describing system behavior.

ID	Requirements	Borland Together	Enterprise Architect	Rational Software Modeler	iUML	Baby UML
R1	MOF 2.0 support	JMI	No	Yes, EMF	No	No
R2	UML 2.0 support	Partly Meta model support	No	Yes, UML2	No	No
R3	XMI 2.0 support	No	No	Yes	No	No
R4	Support fort tool integration	No	No	Partly, through Eclipse	No	Yes
R5	Support for OCL	No	No	No	No	No
R6	Support for abstract action language	No	No	No	Yes	Yes
R7	Model validation	Partly	No	Yes	No	No
R8	Model execution	No	No	No	Yes	Yes
R9	Architecture supporting dynamic models	No	No	No	No	Yes

Table 9 Evaluation of existing tools

5.7 Expectations to this approach

This section describes the theoretical and practical expectations to this approach, UMLexe, which is described in the next chapter.

The previous section described the evaluation of existing modeling tools. This evaluation confirmed that most modeling tools support UML 2.0 at the graphical notation level. It was also shown that these tools do not take advantage of UML 2.0 Meta model for operational semantics. Furthermore, tools which execute UML models provide only support for specifying operational behavior with state machines.

One of the main theoretical goals of ours is to offer a subset of UML 2.0 and an operational semantics using other models than state machines. It is expected to implement this operational semantics in UMLexe. If UMLexe achieve a certain implementation level we expect to create dynamic UML models and execute operational behavior specified by interactions, which is not provided by any tool yet.

The usability of a virtual machine for UML models is closely related to raising the level of abstraction. Another aspect is testing system behavior without creating the system in a specific platform. This will provide users with rapid feedback of the specified system.

Operational behavior can be specified by state machines, activities or interactions. Within the scope of this thesis it is not expected for UMLexe to implement semantics to execute activity and state machines. Furthermore, it is not expect to implement an action language. Consequently, our implementation is not aimed at changing objects state but focus on executing synchronic operations on a component.

To summarize, it is expected for UMLexe to provide a virtual machine which is able to execute components external behavior specified by interactions. This will extend state of the art with other aspects of executing UML models.

6 UMLexe – a UML virtual machine

This chapter presents a proposal solution for UML virtual machine, UMLexe, for creating dynamic UML 2.0 models. As described in the evaluation there are no solutions for executing UML 2.0 models today. Another result from the evaluation is that existing execution engines focus on using state machine models to describe operational behavior. This approach would like to take advantage of the new UML 2.0 standard and use other parts of the model than existing solutions. This thesis will try to make a contribution to execute platform independent UML models which consist of component, class and interaction models. Most software systems today are specified using other diagrams than state machine, such as class and sequence diagrams. With the approach described in this thesis it is possible to describe software systems based on components, class and interactions models and possibly execute these models for testing the specification.

The description of this proposal distinguishes between language aspects and the runtime environment. Making UML 2.0 language executable requires both a UML notation and operational semantics. Within this thesis a subset of UML is created and the necessary operations on this subset are described. In addition, a virtual machine must implement the operational semantics in order to execute these models. The process of developing a virtual machine is based on existing software processes, such as Rational Unified Process (RUP) and Component and Model based development Methodology (COMET). These two aspects are further elaborated in the next sections.

In the first part of the chapter a description of the abstract UML notation and the operational semantics is given. Then a platform independent description of the UMLexe is presented and the last part presents a platform specific description of how it is implemented in Java. The description of the UMLexe approach is given in a generic and platform independent way in order to emphasize its general applicability.

As stated in the previous chapter, the expectation to the virtual machine is to execute UML 2.0 platform independent models, aiming to test platform independent specifications for component behavior.

6.1 UMLexe vision and features

The vision for UMLexe is to support dynamic UML models which instantiate, redefine specifications and let the specification be reflected in the runtime environment. UMLexe should create new instances based on component specifications and components behavior should be specified by interaction diagrams.

The main aspect addressed is executing interaction models connected to component operations. Interactions in UML 2.0 give more action support than UML 1.x and UMLexe should take advantage of this. UMLexe should provide features to instantiate models, wire components together and provide objects inspection. Object inspection give functionality to lookup object component specification. Furthermore, querying runtime instances could either be implemented with OCL [18] or if user needs to change the state of an object UMLexe could implemented this by adding an abstract action language [23] or XOCL [3].

These features will be further elaborated in the next sections in order to fulfill the vision of UMLexe, creating dynamic UML models.

6.2 UMLexe notation and operational semantics

This section explains which diagrams, notation and operational semantics needed for UMLexe to execute UML models. The diagrams in UMLexe consist of primary and supporting diagrams. These diagrams are necessary to define executable models for UMLexe. The primary diagrams are component, interface, port types and interaction diagrams. In addition, for gathering requirements to software systems, use case diagrams are utilized. These use cases is further grouped into subsystems for adding functionality to potential components [33].

The detailed notation used in UML 2.0 is described in Chapter 4.1.2. This chapter describes which static structural elements and behavioral elements that are used and the linking between them.

6.2.1 Primary diagrams

Figure 13, gives an overview of the necessary diagrams for UMLexe. Component diagrams specify components and the ports and interfaces. Components are synthesized from one or more ports. These ports handle messages to and from its collaborators. A port is an object in the context of a mediator pattern [4], which means a component instance can play different roles dependent of who the collaborator is. In order to define messages which are sent between ports we add required and provided interfaces. If a port only has provided interfaces it does not know about the other object but acts as mediator which handles messages. On the other hand, a port with required interface has the possibility to send messages to the collaborators port. Both interfaces and port types are created in the port type diagram.

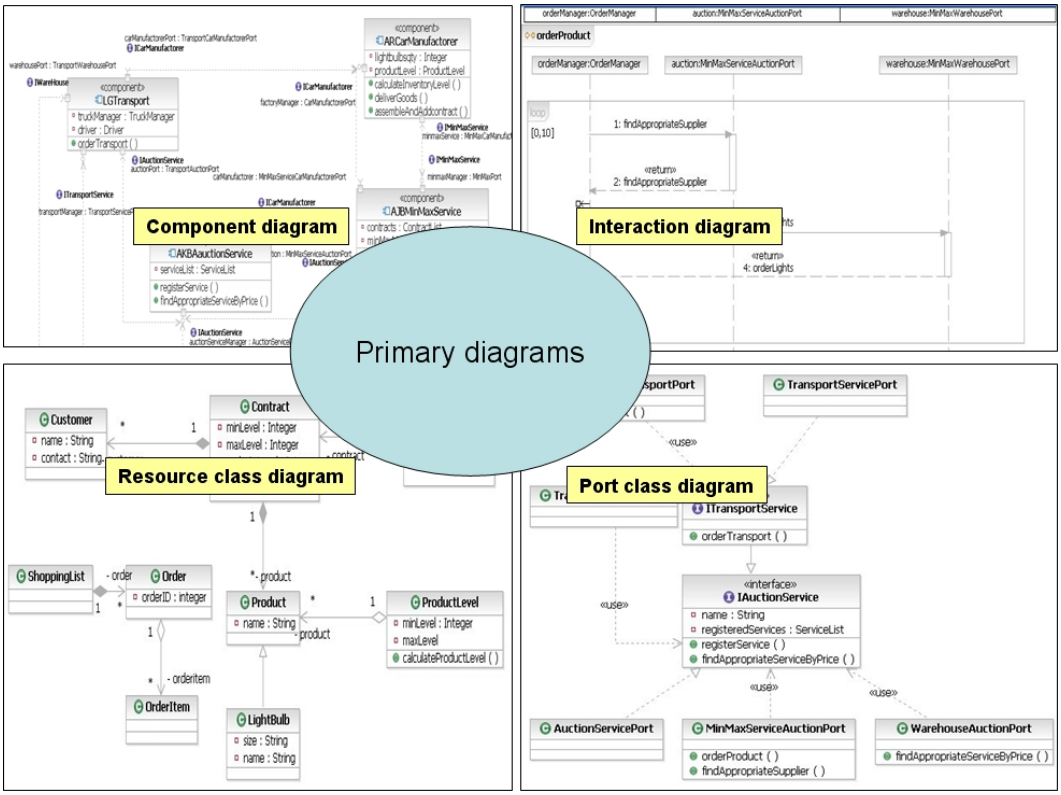

Figure 13 Primary diagrams

When objects receive messages on a port, the specification of the internal behavior is triggered. In order to handle state-dependent behavior we need to add actions for reading and writing values of attributes. State dependent behavior is handled either by XOCL or an abstract action language.

Figure 14, gives an overview of the mapping between component operations and interactions. This mapping links components and interactions that specify behavior with using interaction diagrams. An interaction is bound to the context of the Component object. When the setupContract operation is called, UMLexe looks up the setupContract interaction. This figure only gives a conceptual view of this linking; however Figure 11 describes how this link is achieved in the UML 2.0 Meta model.

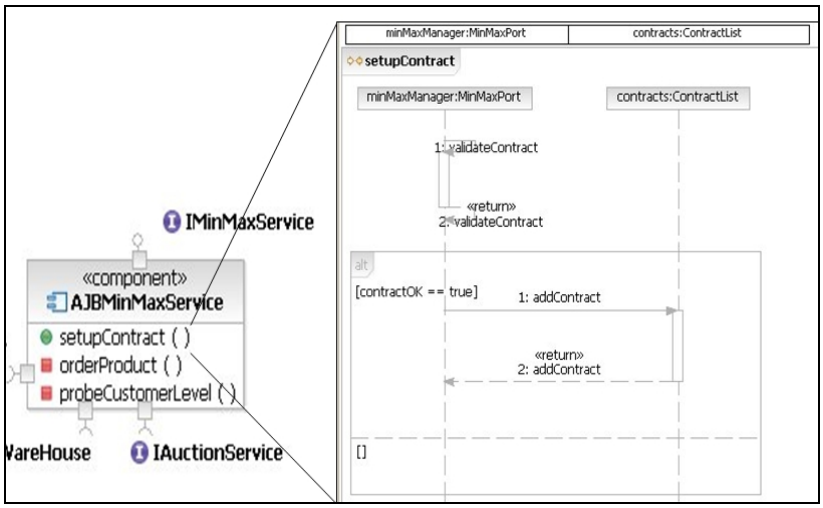

Figure 14 Component operations and behavior specification

This is how the linking between components and interaction is achieved, although all types and parameters must be typed. As described in Figure 14, all ports, attributes and operations are typed element. This is why the notation requires a diagram for port types and resource types as well. This is necessary for controlling slots and value specification when an instance is created.

6.2.2 Supporting diagram

In addition to the primary diagrams, a supporting diagram to capture system functional requirements is necessary. Therefore, use case diagrams with subsystem grouping [33] are added. Grouping use cases into subsystems according to an architectural pattern helps us identifying reusable and maintainable components in the system.

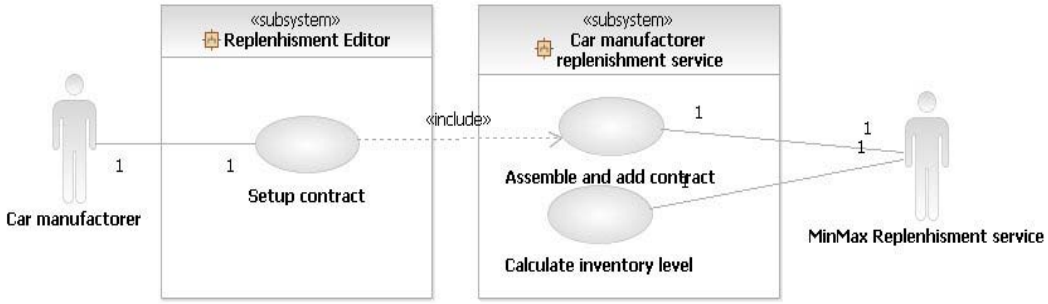

Figure 15 Subsystem grouping

Figure 15, gives a description of a subsystem grouping diagram. This diagram shows use cases grouped into subsystems based on the kind of interface required. If the interface is required from users a graphical interfaces is needed, and if the actor is a service a programming interface is needed. This supporting diagram gives foundation for designing the system and divide systems into graphical and service components.

6.2.3 UMLexe operational semantics

The previous section declared the UML notation which is used. This section adds necessary operational semantics to these models for potentially execute these models. Operational semantics is needed for UMLexe to execute UML models defined with the notation mentioned above. All operations are categorized into three parts; component, instance specification and interaction. This section aims at defining operations needed for executing the UML models. These are operations specified for the UML language level (M2 level in the Meta hierarchy described in Chapter 2.1.4.)

Table 10 describes the component operations needed for handling component instances.

Component operations	Description
CreateObject	Create new Instance specification (runtime object) and add slots based on component structure.
DestroyObject	Remove object from component owned elements.

Table 10 Component operations

Component operations define the context of the execution. To execute an operation a component must be created. Furthermore, it must be possible to remove a component from the memory.

Table 11 describes which operations are needed on component instances to trigger operations on an instance.

Instance specification operation	Description
CreateLink	Sets up a connection between two objects
DestroyLink	Removes link between two objects.
Inspect	Return object classifier.
ReClassifyObject	Changes the "instance of" relation for an object.
ReadVariable	Sets new value on a variable
WriteVariable	Returns variable value.
CallOperation	Looks up an object component and finds the operation. Furthermore, the operations specification is looked up.

Table 11 Instance specification operations

These operations are needed for setting up communication paths between instances and provide a reflective architecture. This will enable changing the type of an instance and look up the current type with the inspect operation.

Table 12 list one operation, execute. This operation executes an instance specification.

Interaction operation	Description
Execute	Execute interaction.

Table 12 Interaction operation

Execute interaction is a composite operation and contains rules for handling all interaction elements, such as transforming messages to operation calls, handling combined fragment with operators such as alt, seq, loop etc.

Table 13 contain one operation eval which takes a textual expression and query the UML model to ensure that the system is in this specific state before executing the following operations.

Guard operation	Description
Eval	Evaluates textual expressions.

Table 13 Guard operation

An eval operation is necessary to control the message flow in an interaction diagram.

In the following section a detailed description of the operational semantics is given. All these operations are specified using XOCL to maintain platform independent descriptions.

All operations are described in the same order as described within the tables above.

```
context Component
 @Operation createObject(o:Object):Element
 Let instance := InstanceSpecification();
 instance.setName(o.getName() + "_" + sequenceNumber());
 instance.setType(o.getName());
 instance.setInstanceOf(o);

 @For attributes in o.getAttributes() do
    self.walkSlot(o,attributes,instance);
  end
  self.setOwnedMember(instance);
 end
```

Table 14 Create object

When create object is called, an instance specification object is created and name, type and "instance of" relation is set. Furthermore, all attributes and properties on a component are added as slots to the instance specification. This operation is recursively called for creating internal parts of a component. Finally an instance reference is added to the component object for maintain connection between M0 and M1 level.

```
context Component
 @Operation destroyObject(o:Object):Element
   let instances := self.getOwnedMembers();
   instances.remove(o);
  end
 end
```

Table 15 Destroy object

Destroying an object is basically involves removing an object from the list of instances in from a component object.

The next category deals with operations on instance specification. These operations should provide instance manipulation.

```
context InstanceSpecification

 @Operation createLink (o:Object, slot:Element):Element
   if checkType(slot.variable, o) then
    slot.setValue(o);
   end
  end
 end
```

Table 16 Create link

Create link add ports and attributes to slot for initializing collaborating instances.

```
context InstanceSpecification

 @Operation destroyLink (slot:Element):Element
    let slots := Self.getSlots();
    let slot  := slots.getSlot();
    slot.setValue(null);
   end
  end
 end
```

Table 17 Destroy link

Destroy link remove reference to a collaborating instance.

```
context InstanceSpecification

 @Operation inspect ():Element
   Self.getClassifier();
  end
 end
```

Table 18 Inspect object

Inspect operation returns the classifier object for the inspected instance.

```
context InstanceSpecification

@Operation reclassifyObject(o:Classifer, mapping:Object):Element
  let classifier := self.getClassifier();
  walkObjectAndTransform(self, o, mapping) ;
  classifier.destroyObject(self) ;
end
```

Table 19 Reclassify object

Reclassify objects change existing classifier for an object. In order to move existing values from one type to another, we chose to introduce a mapping object which tells which attributes in one object is moved to the target object.

```
context InstanceSpecification

@Operation ReadVariable(o:Object):Element
 let slots := self.getOwnedElements();
 let value := slots.getValue(o);
end
```

Table 20 Read variable

Read variable returns value in defined slot.

```
context InstanceSpecification

@Operation WriteVariable(o:Object, value:Object)
 let slots := self.getOwnedElements();
 slots.setValue(value);
end
```

Table 21 Write variable

Write variable set value into a slot defined in instance.

```
context InstanceSpecification

@Operation CallOperation():Element
 let classifier := Self.getClassifier();
 let operation := classifier.getOperation();
 let model := operation.getModel();
 let interaction := model.getElement(operation.getName());
 interaction.execute();
end
```

Table 22 Call operation

Call operation looks up classifiers operation. According to the UML specification an operation do not have direct link to an interaction [2], but we are able to find this operation by looping through interactions in a model and look for the operation object

in the specification field within an interaction. The return value of interaction will be returned to the object calling this operation. This requires that operations and interactions have the same signature.

```
context Interaction

@operation execute(callee:Object):Element
  @For i in self.contents
    //Find message receiver and operation to call
    if i.isKindOf(Message) then
      //getProperty() finds lifeline and the property
      //it represents and return instance to call.
      let receiver := getPropertyInstance(i);
      let op := i.getSignature();
      receiver.callOperation(op);
    else
     i.isKindOf(InteractionFragment) then
      let operator := i.getOperator();
      if operator.isKindOf(alt) then
       let operandsList := operator.getOperands();
       let guard := operand.getGuard();
       @For y in operandList
        if guard.getGuard().eval() then
        @For x in y.getContexlist
         execute(x); //Call recursively on interaction
        end //of for
       end //of if
      end //of if
     end //of if else
    end //of For
end // of operation
```

Table 23 Execute

This operation loops through all messages in an interaction and call operations on receiver of a message. This operation is made simpler for the reader according to different interaction fragment types, but the concept is equal in order to handle other operation types like, assert, sequence, and loop. The difference is how we implement the operator.

Adding these operations to the UML Meta model, the expectation to the implementation is to provide dynamic UML models. This section described operations and semantics for executing UML models in UMLexe. These operations should be added into the UML Meta model in order to execute these models directly, although changing the existing UML model is not wanted and therefore external objects which execute these models is implemented. The design of these virtual machine system components is described next.

6.3 UMLexe - Requirements

With the identified UML 2.0 subset notation and operational semantics described, the next section will elaborate the specification for the runtime environment for executing this subset. As stated in Chapter 2, a UML virtual for executing dynamic UML models requires MOP architecture to support dynamic models. The basic idea of this approach is to take advantage of the MOF Meta hierarchy and create instances from M1 model elements.

As observed from the evaluation chapter there are no executable engines which supporting other diagrams than state machines to describe operational behavior. Within this section a description of a UML virtual machine which is able to use interactions to describe operational behavior. The process of specifying is based on the operational semantics declared in previous chapter and it is further derived with use of the COMET methodology [33]. This section describes the functional and non-functional requirements for UMLexe. The functional requirements derived based on use case diagrams and subsystem grouping. The non-functional requirements are described with text.

6.3.1 Functional requirements

Functional requirements is derived by using use case diagrams together with a study of existing MOP architectures, such as Smalltalk [14] and Squeak [13]. Figure 16 shows use cases identified within UMLexe. These use cases are grouped into subsystems and defines components in UMLexe. UMLexe consist of a modeling editor, an application viewer (Application tool) and the UMLexe engine. The modeling editor tool should include features for making graphical diagrams in UML 2.0. The application viewer should let users inspect runtime objects and stack traces.

- **UMLexe** is the main component that provides dynamics to the objects defined in the UML model. When designing a UML virtual machine there is a difference between operations on the UML language as described in previous section, and system objects which provide functionality to load and control models, control the interpreter and the object memory.

- **UMLexe editor** provide users with functionality to create, save, edit and delete UML models. This functionality will not be developed within the context of this thesis, but we take advantage of an already existing modeling tool. We will provide an interface for existing tools to collaborate with.

- **UMLexe application viewer** provides users with graphical interface for viewing execution of operations and instantiated objects. This component provides possibility for executing operations as well.

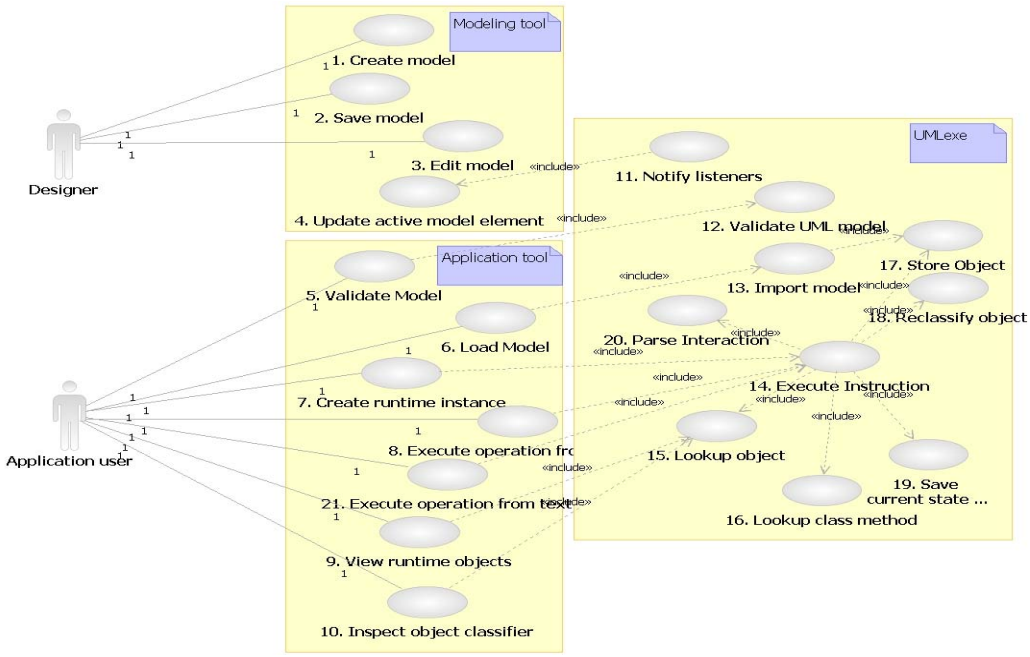

Figure 16 UMLexe use case diagram

Use cases	Description
1. Create model	Feature for creating graphical diagrams
2. Save model	Save model to XMI
3. Edit model	Change existing model
4. Update active model element	Set the current object active in diagram
5. Validate model	Validate model against UML 2. 0
6.Load model	Load model into UMLexe
7. Create runtime instance	Create new instance from component
8. Execute operation	Call operation to execute
9. View runtime objects	Lists all objects in UMLexe memory
10. Inspect object classifier	Returns the class type for an instance
11. Notify listeners	Notify modeling environment for changes
12. Validate UML model	Triggered from application tool
13. Import model	Imports model into UMLexe
14. Execute instruction	Execute operational semantic on model
15. Lookup object	Finds object in memory
16. Lookup class method	Returns instance type
17. Store object	Stores active instance descriptor
18. Reclassify object	Change instance type
19. Save current state	Saves current state of an instance
20. Parse interaction	Parse an interaction to create instructions
21. Execute operation from text	Interpreting for instance XOCL or AS.

Table 24 Use cases

6.3.2 Non-Functional requirements

The main purpose of UMLexe is to provide a partial implementation of a dynamic UML environment, although it should be possible to continue working on UMLexe and in order to provide a complete dynamic UML engine. The main goal of UMLexe (in the scope of the master thesis) is to provide the core functionality of such an engine. Therefore, non-functionality of such as security, performance and availability will not be considered.

6.3.3 User Interface

UMLexe should provide user interface for its users. This interface will enable UMLexe users to use the system described in the use case section. The user interface is divided into two parts: The modeling editor, which provides graphical modeling and tree structures for managing UML models and the user interface for inspecting runtime objects and interaction traces.

UMLexe must provide a user interface for it modelers. This modeling user interface should provide the ability to model UML 2.0 elements defined in the UML subset. Furthermore, UMLexe must provide a user interface for loading UML 2.0 models represented in XMI 2.0.

6.4 UMLexe Platform independent description

This section presents the platform independent description of UMLexe. The platform independent model consist of component structure model, component activity model, component interface model and platform independent types.

This model defines both static and behavioral structure for UMLexe. The design strategy supported by UMLexe is to divide the engine core into two parts, modeling manager and core. The Modeling manager takes care of importing models and provides this model to the core component. The Core component contains object memory and interpreter. The Object memory handles runtime object lifecycle. Finally, the interpreter manages operations on UML language.

6.4.1 UMLexe - Components model

Our design is inspired by Smalltalk virtual machine [34] and the Meta Object Protocol [20]. As stated earlier UML virtual machines should implement MOP architecture to execute models. This will give a more flexible architecture. Smalltalk provide key technology due to reflection and the possibility to reconfigure runtime instances. This is also the key feature that should be supported by UMLexe. Figure 17 gives an overview of the UMLexe architecture. UMLexe is divided into two parts, one conceptual part which is based on the OMGs Meta hierarchy described in Chapter 2 and the physical architecture which implements the dynamics of a UML model. The overview of UMLexe shows that it contains two tools, one for inspecting the runtime and one for designing UML models as described in the requirements section. These are named UML Model Editor and Application viewer.

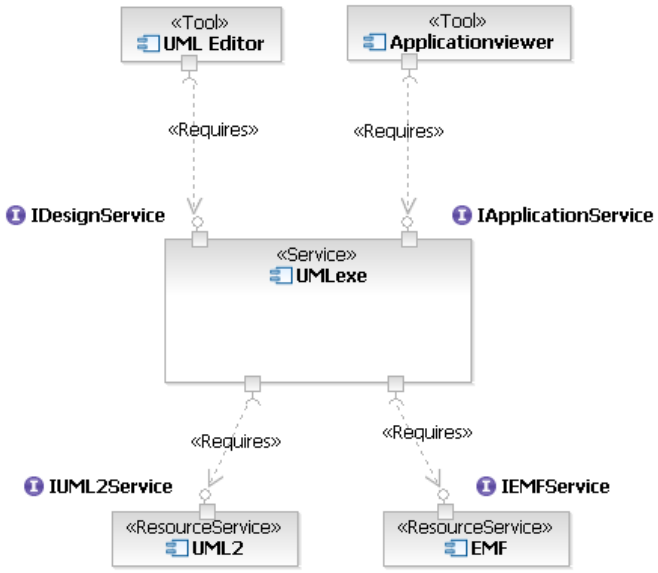

Figure 17 UMLexe Architecture

As described in Chapter 2, the UML Editor should provide support for UML 2.0 minimum subset as described in Chapter 4.1.2. The application viewer is a tool for inspecting classifiers and to show the runtime instances. Furthermore, the application viewer shows the history stack trace, which is a history log of messages sent between runtime instances.

In order to fulfill the requirement for the MOP architecture UMLexe should implement functionality to manipulate UML 2.0 and MOF models. As described in Figure 17, UMLexe requires both UML 2.0 and MOF.

Figure 18, describes internal view of UMLexe and how it is further decomposed into modeling manager and core parts. The Modeling manager is the front end to the core component. This front end dispatches messages to core and interact with modeling editor and application viewer. Furthermore, the core component consists of an interpreter and an object memory. These two components makes the UML models dynamic, in terms of creating new instances from component class objects and to execute interactions between components.

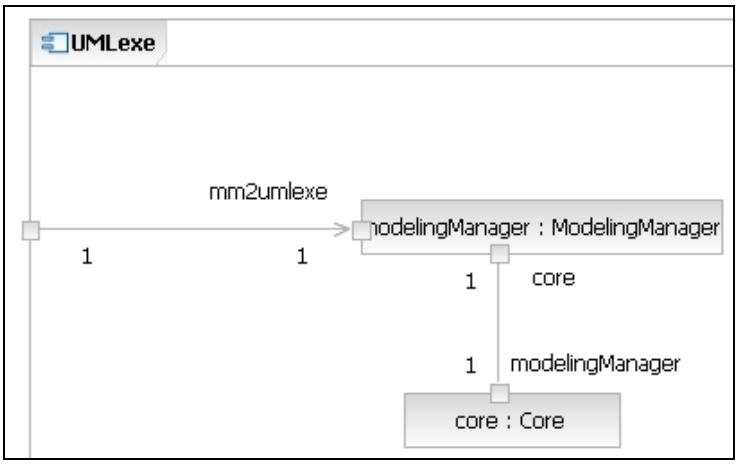

Figure 18 UMLexe Internal view

The front end provides the interface to modeling tools for importing models and providing notifications of which object is active and operations to execute. A simplification which is made is that transforming instructions based on the action semantics provided in UML 2.0 into XML or byte code is not provided. Instead, this version interprets UML model directly.

The Core object, as the name indicates, is the main component in UMLexe and manages instructions. It consists of an interpreter and object memory. The interpreter executes operations and interactions, save the current state of the interpreter in order to switch between interactions and handle messages sent to an object. The object memory provides an interface to the interpreter for creating new objects and looking up objects.

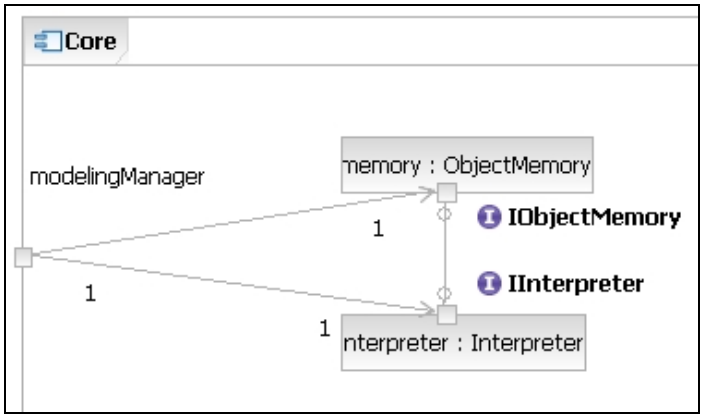

Figure 19 Core component

The Interpreter looks up interactions and the starting point. Due to the missing implementation of interaction gates in modeling tools, another simplification is made. We assume that the first message sent from leftmost lifeline is the starting point. Then is the instruction pointer is set to point to the next instruction in the interaction, which is a message sent from one lifeline to another. The next step is to perform the method specified in the message signature. If there is no signature defined in a message an error is reported to the user. The interpreter should find the appropriate operation to interpret in response to a message by searching a message signature. Then the message signature

is used to lookup the right method in the receiver's component operations. The structure of a component and its associated operations is shown in Figure 9. In addition the interpreter should use component instance specification to determine its instance's memory requirements, such as the number of instance variables defined. This is also known as dynamic type checking in compiler construction [35]. Dynamic type checking is performed during execution. UMLexe should implement type checking according to name equivalence. Name equivalence means that two types are equal if and only if they are either the same simple type or are the same type name [35]. Furthermore, the interpreter should load and unload objects into the active context. When an object is loaded it changes state from passive to active. This is part of the mechanism to switch between active objects in the runtime environment.

The Object memory object manage model changes and when create object action occurs it stores new objects into memory. Furthermore, it is possibly to destroy object in object memory although a garbage collection mechanism could be implemented to support this. Garbage collection is a mechanism provided by most virtual machines. It handles reference to objects and removes objects from memory when there are no clients to this object. Then object memory should provide the interpreter with an interface to all objects defined in the UML model, which means all objects and Meta objects. When creating a new object the object memory associates each object with an object reference to a classifier object, which makes up the type declaration. Type checking ensures that the classifier is defined accessible in the model name space. This feature makes it possible to create objects and inspect what kind of type an object is created from and lookup the specification of operations to interpret.

6.4.1.1 Application viewer internal class design

Figure 20 gives an overview of the internal class design for the application viewer. The application viewer component is represented as a package and the internal view of this package is shown below. The main component which is loaded into the platform is named UMLexePlugin. All linking between these components is configured in an xml file that is parsed by the platform when loading this plug-in into memory.

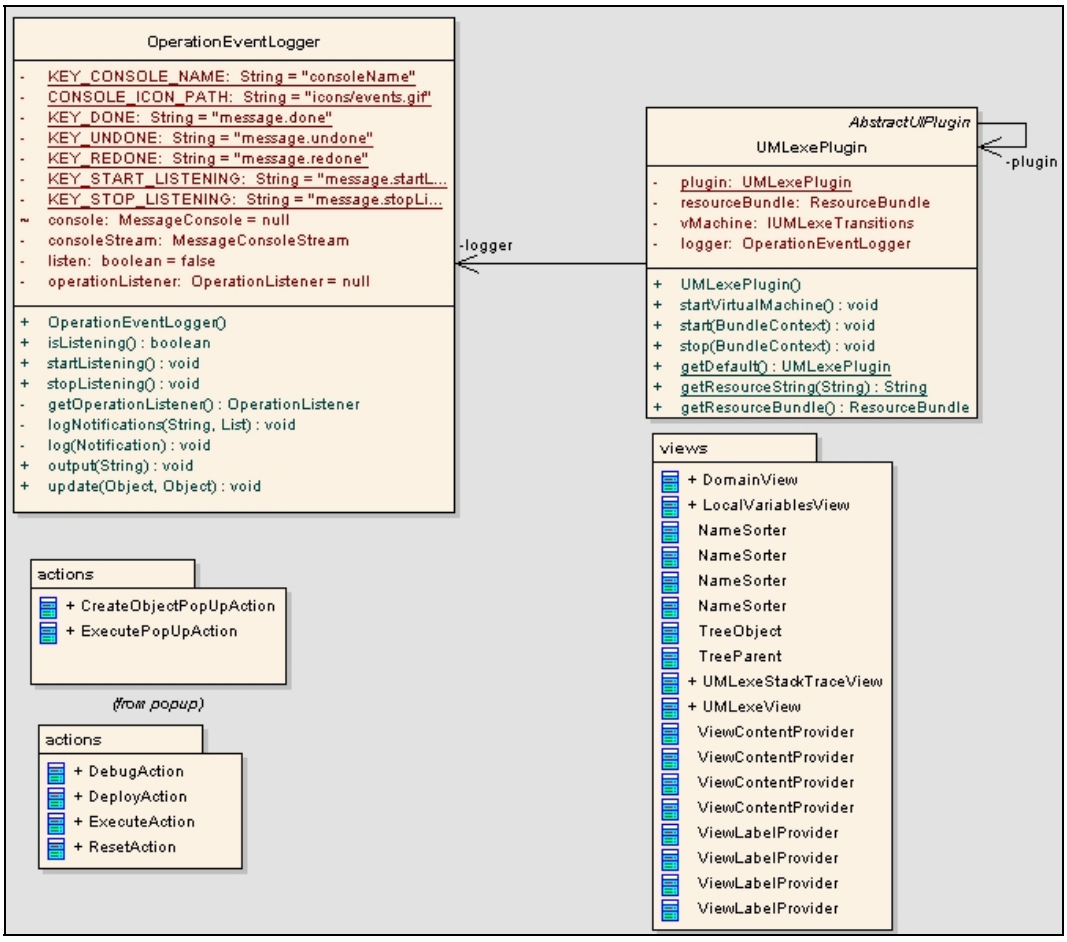

Figure 20 Application viewer internal class view

The application viewer component is the bridge between user interface, modeler and the UMLexe engine.

6.4.1.2 UMLexe internal class design

Figure 21 gives an overview of the internal class design in the UMLexe component. This component is divided into one package for UMLexe, the core engine and the model manager. The UMLexe component use the singleton pattern [36] to guarantee that it is only one instance of UMLexe.

"Ensure a class only has one instance, and provide a global point of access to it."

This design pattern is used because it is important that it only exist one UML virtual machine to interact with.

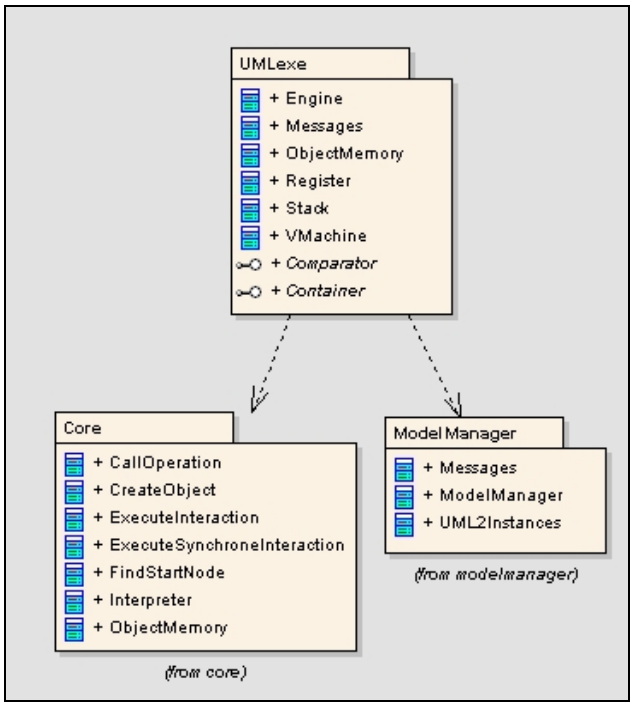

Figure 21 UMLexe internal class design

The core package contains the rules and the interpreter that looks up the right strategy [36] when the engine is triggered to execute a statement.

"The Strategy pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses."

The strategy pattern defines a family of algorithms that encapsulate each one of them and make them exchangeable. This pattern lets the algorithm vary independently from clients that use it.

6.4.2 UMLexe Component interaction

This section describes how the UMLexe works and this is described by using an activity diagram and then the activities are explained with XOCL expressions and text. Finally, the internal component interaction is described with interaction diagrams.

Figure 22 describes the situation of executing models in UMLexe. The internal work flow of UMLexe shows that model manager imports an XMI model and creates a data structure in memory, which represents the model.

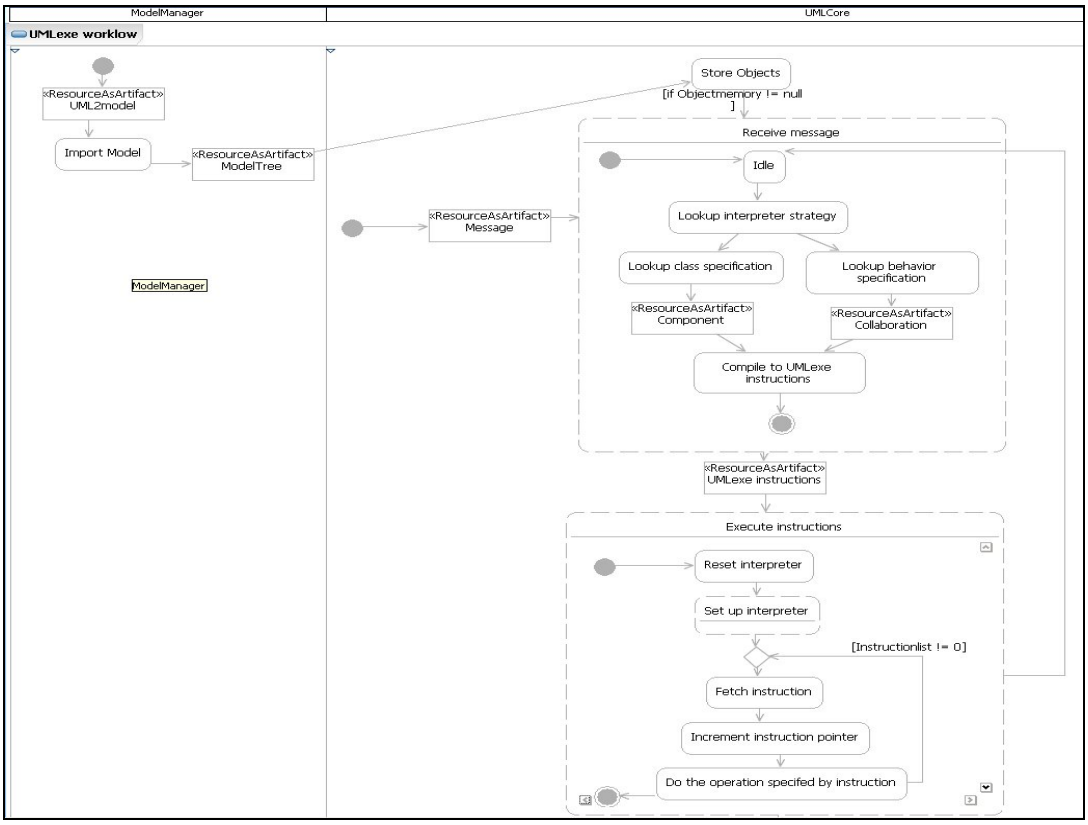

Figure 22 UMLexe internal work flow

The next section describes the activities within the figure above. Table 25 import model, describes how the UML 2.0 model is imported into UMLexe and added to the model manager object.

```
context ModelManager
@Operation importModel(o:Element):Element
  Let tree := walkXMIFile(o);
 end
end
```

Table 25 Import model

Input parameter o, represents a XMI 2.0 file structure. This file is parsed in the walkXMIFile method which returns a graph structure with all the model elements. This data structure is assigned to the tree attribute in model manager.

The next activity is to store this graph of objects into the object memory in UMLexe. In order to achieving this, the data structure is delivered to Core object for storing it in Object memory. When this model is stored into object memory UMLexe is ready to receive messages to manipulate this model. Table 26 specifies the storeObject operation.

```
context UMLCore
@Operation storeObject(o:Element, arg:Element)
  let mem := findObject(arg);
  mem.uml2Objects := o;
 end
end
```

Table 26 Store Object

The storeObject operation looks up the memory (with parameter arg) object and sets the graph (parameter o) to the object memory in UMLcore. When the model objects are set into the object memory, UMLexe is ready to receive operation calls.

When UMLexe receives an operation call, for instance, to execute an operation, it is specified like this:

```
context Person
  let p := Person();
  let x := p.calculateSalary();
 end
```

Table 27 Salary example

UMLexe call operation createObject, specified in Table 14, to create a new instance. Then the operation execute instruction in Table 23 is called to execute interaction within the context of runtime instance p. A simplification is made and that is no support for asynchronous call. This is addressed to future work.

As described in Table 28, there are a difference of interactions and instantiating new objects and this is controlled by checking the type of these objects.

```
context UMLCore
 @operation getStrategy(o:Element):Element
   if o.lof() = Interaction then
    let strategy := getInteractionStrategy();
   else
     let strategy := getInstanceStrategy();
  end
end
```

Table 28 Get strategy

UMLexe looks up the right strategy and returns it to Core object. A strategy is defined as a class object and is returned to interpreter for executing the right rule.
To illustrate the interaction between components how the calculate salary example above is handled; an interaction diagram is given next.

Figure 23 describes the interaction between internal elements in UMLexe when the example in Table 28 is executed.

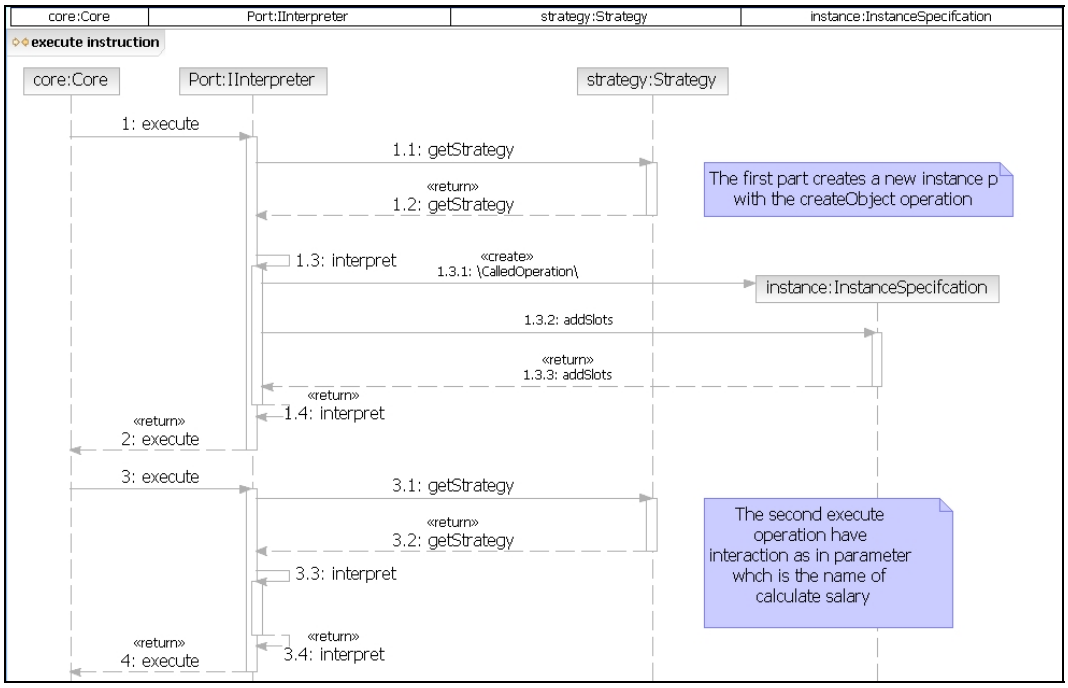

Figure 23 Component interaction

As described in the tables there are two instructions, one that create an instance p from the Person class, and one which executes an operation on this object. This is achieved by sending in two execute instructions to the interpreter. The interpreter finds the correct strategy rules for interpreting the object. The first execute statement returns a createObject strategy in order to create a new instance and add the necessary slots based on the class attributes. The next statement to execute is an interaction for calculating the salary. Based on the operation it finds the interaction strategy to execute the interaction object.

This section has described the system objects and language objects which are necessary to execute UML models. Within the design of the UMLexe the necessary operational semantic is moved from the UML language to external objects within UMLexe. This is necessary since the UML 2.0 language is still a moving target.

6.4.2.1 Information model

This section describes which value objects UMLexe can manage. For executing a components operational behavior UMLexe deal with the UML 2.0 subset described in Figure 9 and Figure 10. These subsets consist of Meta objects supporting both static and behavior objects. How UMLexe operate on the UML models is described in section 6.2.3. However, in the next section a few notes are given on the component elements we consider.

The structure of a component
A component represents a modular piece of a logical or physical system whose externally visible behavior can be described much more concisely than its implementation. Components do not depend on other components but on interfaces that the component supports. A component can be replaced by an instance of any component that supports the same interface. Components are the "real" objects and developers should focus on objects roles and add these roles to components [9].

As described in Figure 9, a component is derived from class and have required and provided interfaces. These interfaces represent dependent or provided behavior. Furthermore, components have attributes, representing components state. Finally, wiring components together are done by nesting the provided and required interface.

As described in Table 14 we traverse the components structure to create an instance specification and add necessary slots. Furthermore, type checking is also important in order to wire components together. UMLexe checks if the type of required and provided interface is equal by looking up specified type name are in the UML2 model imported into the object memory. If the types are not equal an error is reported to the user.

Operations
A component's operation is specified by interaction diagram. An interaction describing an operation must contain properties or attributes within components context. If the object does not exist within the context an error is reported to user.

Inheritance
Inheritance is not supported in this version of UMLexe, but should be implemented in the future.

6.4.3 UMLexe Component interfaces

Figure 24 gives an overview of interfaces provided and required for UMLexe. In order to interact with UMLexe, two interfaces are identified, IModelerService and IApplicationService. IModelerService defines an interface for the modeling tool in order to notify the graphical user interface about which object is active and which instruction is executed.

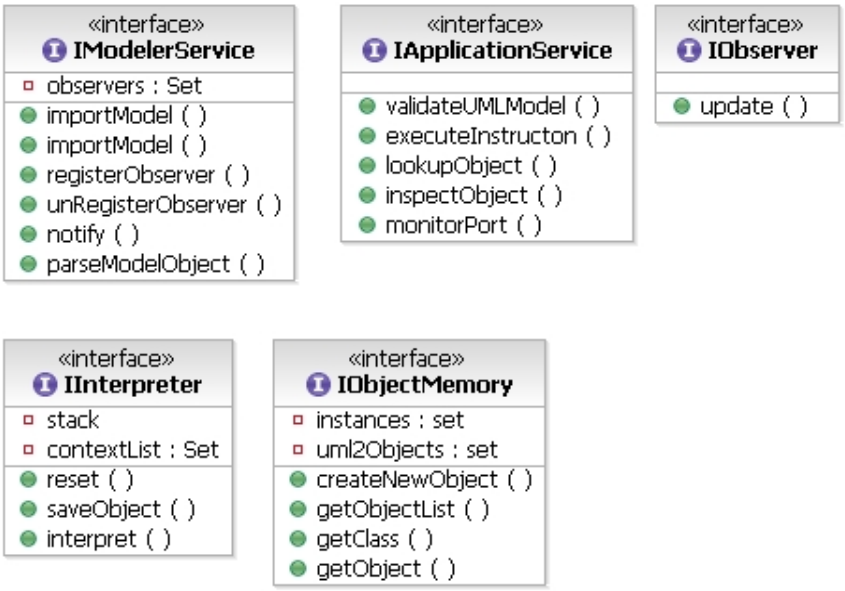

Figure 24 UMLexe interface

Furthermore, IApplicationService defines operations for loading and executing operations on model elements. For, instance it should be possible to create a new instance based on component specification. All these interfaces and operations are derived from the use cases described in section 6.3. As described earlier these interfaces is implemented by components and classes which makes up the UMLexe engine.

Here, a specific operation will be elaborated in detail as an example, the executeInstruction operation in IApplicationService.

```
context Core
@Operation executeInstruction(o:Element):Element
 Case(o)
  o.of() = self.Interaction do
    self.fetchInstruction(o);
  end
 end

@Operation resetInterpreter():Element
 let interpreter := self.find("Interpreter");
 interpreter.activeContext.stack := null;
 interpreter. activeContext := null;
 end
end

@Operation setupInterpreter(o:Element):Element
 let interpreter := self.find("Interpreter");
 interpreter.activeContext := o;
 end
end

@Operation fetchInstruction(e:Element,o:Element):Element
  resetInterpreter();
  setupInterpreter(o);

  Case(e)
   e.of() = self.caseEventOccurrence do
    if e.sendMessage() != true then
      interpreter.activeContext.stack.push(e);
    else
      interpreter.activeContext.stack.pop(o);
  end
 end
end
```

Table 29 Execute instruction

Table 29 gives details of how the interpreter executes instructions in an interaction specification. An operation call invokes this routine and fetches the instruction to look up the interpreter instance and sets the active component. Then these messages are added to the context stack. In addition it is necessary to parse the combined fragment operators for handling different flow mechanisms. These operators contain guards which are populated with XOCL or abstract action language expressions. A simplification which is made is that parsing XOCL or an abstract action language is not implemented. This reduces the potential for handling the flow mechanism in interactions. This will be addressed in future work.

6.5 UMLexe Implementation

This section describes the implementation of UMLexe on the Java platform. This platform was chosen because it is very widely deployed and provides a good number of open source modeling frameworks.

The execution mechanisms described in the design section are implemented on top of existing execution mechanism in Java. Furthermore, these mechanisms are extensions to existing modeling frameworks implemented on this platform.

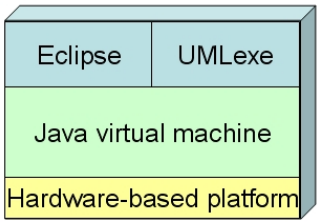

Figure 25 Architecture

UMLexe application tool is implemented as a plug in to the Eclipse platform. This makes it possible to take advantage of existing plug-ins deployed into Eclipse. Eclipse is an open source platform-independent software framework for creating rich-client applications. So far this framework has typically been used to develop IDEs (Integrated Development Environments). However, it can be used for other types of client application as well.

"Eclipse is a kind of universal tool platform - an open extensible IDE for anything and nothing in particular" [37]

The first section describes the main parts of the implementation and how this diverts from the generic approach described in the previous section. Integrating with existing modeling editor has some special issues and is addressed in the next section.

6.5.1 Main implementation

This section describes the main parts of UMLexe. The main objective of the implementation has been to execute operational behavior on components which match the state-of-the art UML model execution methods of today.

Both the main implementation, UMLexe, and the application viewer take advantage of existing modeling frameworks implemented in Java. These two frameworks are Eclipse Meta Modeling Framework, also known as EMF [31] and UML2 [32]. EMF is a generic modeling framework. This is not a MOF implementation, but is aligned with MOF and provides us with necessary Meta model mechanisms. The other framework used is UML2 which is a Java implementation of UML 2.0 Meta model specification. This project is derived from EMF and adds necessary mechanisms for manipulating UML 2.0 models. Both of these frameworks are implemented as plug-ins to the Eclipse platform [37]. As described earlier, it is necessary to implement MOP architecture to

execute dynamic models and these frameworks provide MOP architecture to a certain level.

The Eclipse workbench is implemented by a group of plug-ins, with the user interface in a UI plug-in and the non-UI infrastructure in a separate core plug-in. This separation of UI and non-UI code allows the Eclipse workbench core infrastructure to be used in GUI-less configurations of the Eclipse Platform, and by other GUI tools that incorporate Java capabilities. Figure 26 illustrates key connections between the Eclipse workbench and the UMLexe plug-in.

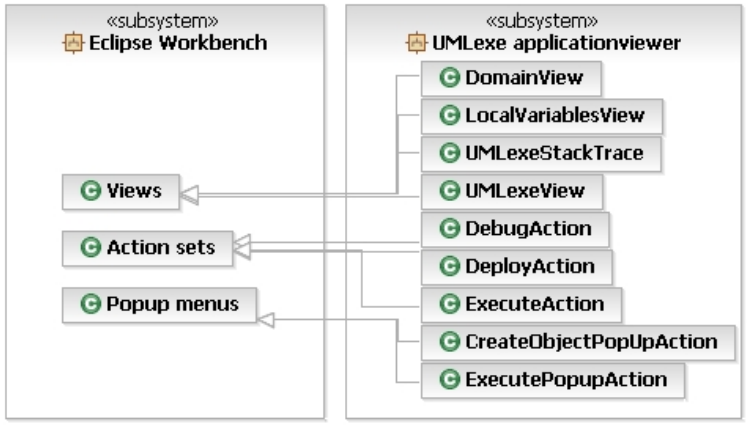

Figure 26 UMLexe plug-in views

UMLexe plug-in adds extra functionality to the Eclipse platform in three extension points. UMLexe add four extra views, four action sets and two extra popup menus. This is how the UMLexe plug-in collaborate with the Eclipse platform. Furthermore, does the plug-in act as a dispatcher to the UMLexe engine. It routes messages between the engine and the user interface.

Figure 27 gives an overview of the application viewer. This Eclipse workbench shows the Rational model explorer which takes advantage of existing UML2 functionality to browse UML2 models and provide us with functionality to select which UML elements to interact with. In center of Figure 27 Rational modeler adds functionality to create UML 2.0 diagrams. When it comes to our plug-in we have added a menu for loading, executing, debugging and reset models in UMLexe. Furthermore, we add a viewer which reflects instances within UMLexe and interaction traces after executing operations.

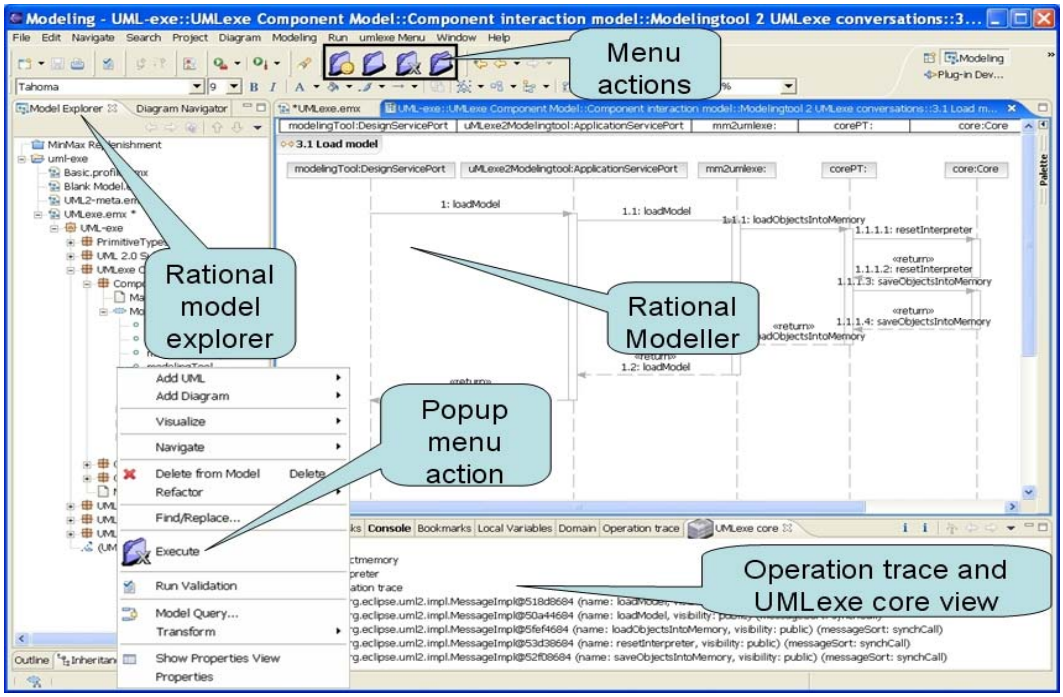

Figure 27 UMLexe workbench

This implementation contains buttons for load model, execute, debug and reset actions. Deploy, execute and reset are implemented and debug will be implemented in the future. These actions implement the org.eclipse.ui.IWorkbenchActionDelegate interface and are loaded in the Eclipse environment when UMLexe application viewer plug in is deployed to Eclipse. This interface contains operations for executing menu action, initialize the action, disposing allocated resources and handle selection changed events. As described these actions will be part of a menu category which is added to the workbench.

The popup menu action implements execution of interactions and in a future release we plan to implement this action on operations as well. Popup menu action implements the org.eclipse.ui.IObjectActionDelegate interface. This interface contains an operation which captures active element and trigger action chosen by a. user. As described in Figure 27 this action is loaded and added to the pop up menu in Eclipse.

The operation trace view gives a stack trace output after executing an interaction. Operation Trace view object extends the org.eclipse.ui.part.ViewPart in Eclipse and collaborates with UMLexe. This integration is achieved by implementing the IUMLexeListener[5], which is an observer pattern [36] that notifies listener throughout execution in UMLexe.

The view part which provide one operation, create part control that register the view in Eclipse and sets up the internal structure. This plug-in is added to the workbench during initialization of Eclipse. UMLexe provide read only possibility to the internal structure of UMLexe component. This view is extended from the org.eclipse.ui.part.ViewPart as well. This component shows the Object memory runtime instances, the active object stack and the stack trace for the previous execution.

With creation of static structures in place, the UML virtual machine becomes more complex and should provide us with dynamic UML models. With a two way communication with the Rational modeler could provide us with a graphical monitoring tool which colorizes a stack trace.

UMLexe is divided into two main packages representing Application viewer component and UMLexe component as described in design section. Application viewer is implemented as a plug-in to Eclipse and take advantage of existing plug-ins implemented in this platform, such as workbench, menus, EMF and UML2. We choose not to implement UMLexe as a plug-in to Eclipse, but as pure Java implementation. This made our implementation usable for other containers as well, such as Spring [38]. Spring is a lightweight component container which could provide extra functionality to our virtual machine.

[5] org.sintef.eclipse.umlexe.vm.IUMLexeListener

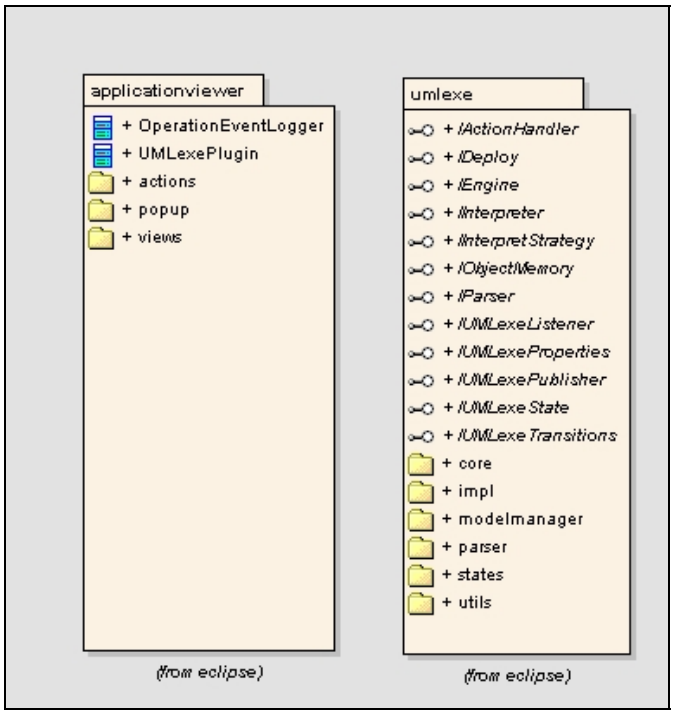

Figure 28 UMLexe packages

Figure 28 gives an overview of packages and interfaces currently implemented. As described, components are assembled into packages with interfaces and implementation classes. Description of all class diagrams is given in Appendix 11.

In addition to the Application viewer and UMLexe we earlier introduced a modeling tool. Our implementations focus on executing models and therefore we have integrated UMLexe with an existing modeling tool. We have chosen to integrate with Rational Software Modeler (RSM) because this modeling tool is implemented as a plug in to Eclipse as well. RSM provide us with necessary API in order to collaborate with this tool. As described in Chapter 5 RSM takes advantage of UML2 [32]. Our implementation could then straightforwardly integrate with this tool.

Before we can execute models within UMLexe we must ensure that models specified are created according to UML 2.0 Meta model. This functionality is provided within UMLexe with existing features in UML2[6] by pre-processing models before they are loaded into object memory.

A model is loaded into UMLexe with functionality provided by EMF[7] and the instantiated model structure is handed over to object memory. When this model is loaded, UMLexe is ready to execute interactions specified within this model. As described earlier creating new objects based on components is not fully implemented yet, but UML2 provides us with necessary functionality to achieve this. The interface UML2AdapterFactory from UML2 can produce instance specifications and if we add parsing mechanisms the component structure we believe that we can control the

[6] org.eclipse.uml2.util.UML2Validator
[7] Java package: org.eclipse.emf.ecore.resource.Resourceset

“instance of” requirement specified in Chapter 2.1.10. UML2AdapterFactory is based on the factory pattern [36] and handles the OMG's UML Meta level and runtime instances. Object memory provides an interface to the interpreter for looking up objects by querying the UML model and the list of runtime instances. Then, Object memory should support the possibility to reclassify the runtime instance by re-factoring the classifier. This feature is a model re-factoring [39] algorithm and one possible approach is to add rules for re-factoring classes, such as, add class, remove class, rename class, add attribute, remove attribute and rename attribute. These rules, which handle reflection on runtime instances, could be implemented with the strategy pattern [36], where the actions from the modeling environment choose the right algorithm to use. These rules handle reflection on runtime instances. One problem of reflecting these changes directly to runtime instances is that the current state of the runtime instance will be affected and cause unwanted changes to the system state. One approach for managing this could be to copy the existing class object structure and let the existing runtime instance refer to original class object. Then the new runtime instances are created by using the new re-factored object. This would probably lead to a great number of objects and unless there is a garbage collection [35] mechanism deleting the old class object when all runtime instances are deleted. Another approach is to restructure the internal structure of the runtime instance and not allow any changes in the interface. The interface is the contract wiring runtime instances together and not allowing the contract to change keeps the consistency between collaborating components. The last approach is the one preferred in order to realize reclassify object feature.

Executing operational behavior in UMLexe is handled by the interpreter[8]. As described in the design section we need to add functionality for parsing XOCL or an abstract action language for handling textual expression in guards. Our approach will implement this according to XOCL specification, but implementing an interpreter for XOCL is not possible within the timeframe for this thesis. When specifying component operation we can use interactions, activities or state machines. UMLexe implements rules for looking up interactions, but by using the strategy pattern we can extend UMLexe with rules for handling activities and state machines. Executing interactions is implemented with a tree iterator and a switch class[9] provided by UML2[10]. We used these classes and created rules for fetching messages specified in an interaction.

6.6 Summary

This chapter described our proposal to execute models. The first part described the notation and operational semantic which is the foundation for the implementation of the solution. The UMLexe implementation is tested in the proof of concept chapter and shows how the environment interacts with Eclipse.

[8] org.sintef.eclipse.umlexe.core.Interpreter
[9] org.eclipse.uml2.util.UML2Switch
[10] org.eclipse.emf.ecore.util.EcoreUtil

7 Proof of concept

In this chapter we will step through a UML 2.0 model execution with UMLexe. The base model for this execution is the case presented in Chapter 3. The purpose for this proof of concept prototype is to prove the feasibility of executing system models described with UML 2.0 containing composite structures and interactions.

The overview of the process of executing a model in UMLexe is described in Figure 29.

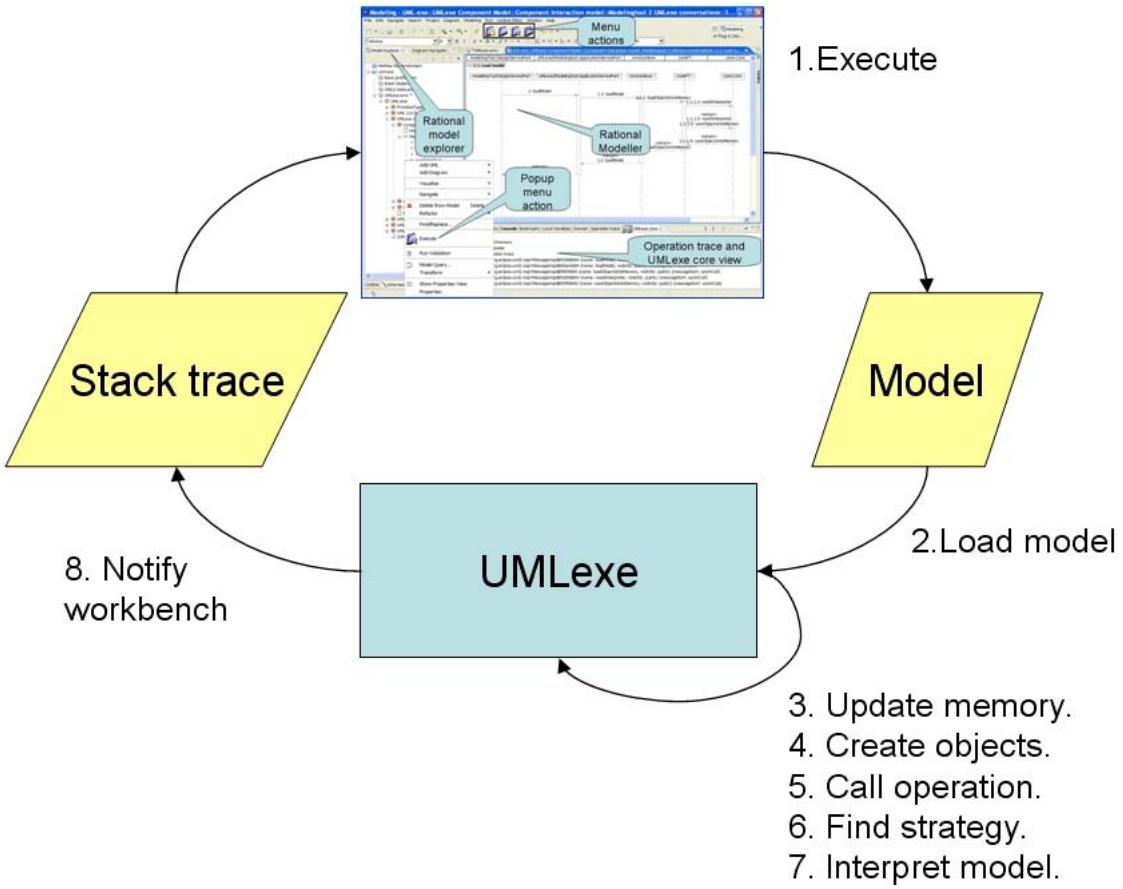

Figure 29 Process overview

The steps we are going through are: First, we load the model into the workspace. Then we execute the model. This triggers load model with the model as input to UMLexe. Then UMLexe run an initialization process for creating instances from components and wire these components together. Furthermore, we execute a component instance operation and return the stack trace to the workbench, which updates its view. As we have described UMLexe provides initiation of components, triggers operations and executes the state independent collaboration between components. The details of what is happening in UMLexe are described next.

Figure 30 illustrate how the workspace looks like when an UML 2.0 model is loaded. This is a two-step process. First we to import model into the modeler and then we load this model into the UMLexe engine. Loading a model into the modeler is happened as follows:

1. **Start** Rational Software Modeler.
2. Choose **file** and then **import**.
3. Select **Existing Project into Workspace.**
4. **Browse** for the min max project catalog.
5. **Click** the finish button.

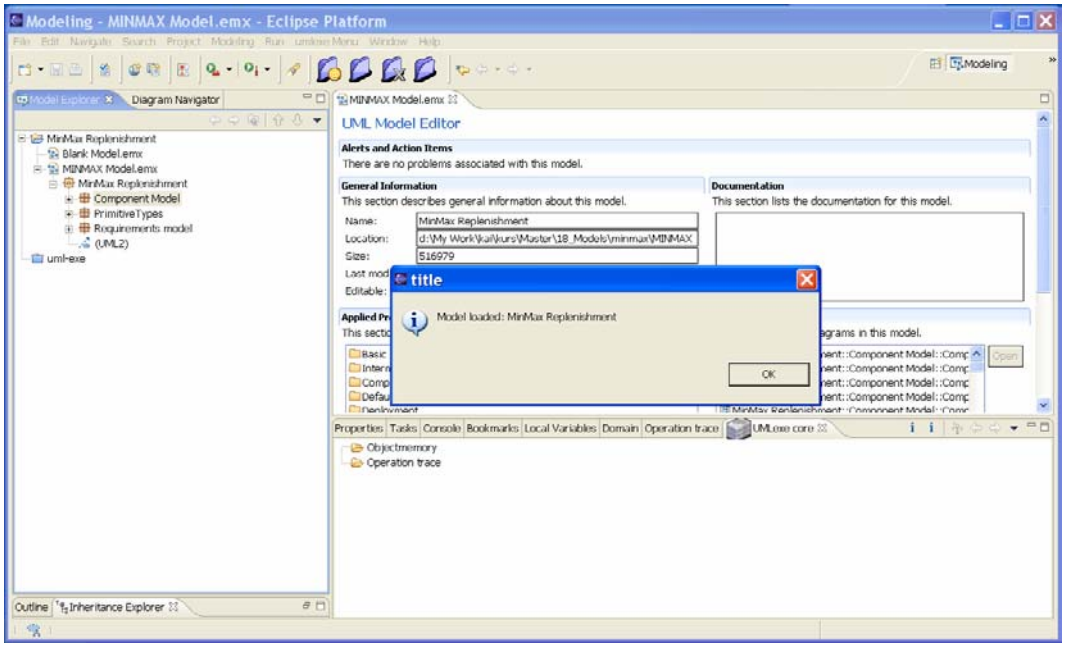

Figure 30 Load model

The next step is to load the min max model into UMLexe virtual machine. This is done by selecting the min-max replenishment node in the Model browser and then we click the deploy button.

When the model is loaded UMLexe change state and Figure 31 describes the internal structure of UMLexe after loading the model. This view is a snapshot of the object browser provided to the Java virtual machine. Since we have not implemented an object browser for our virtual machine, we use the object browser within Java. The top level, named VMachine, contains slots for holding property values. The property current state is set to model loaded and the property uml2Model specify which model is loaded.

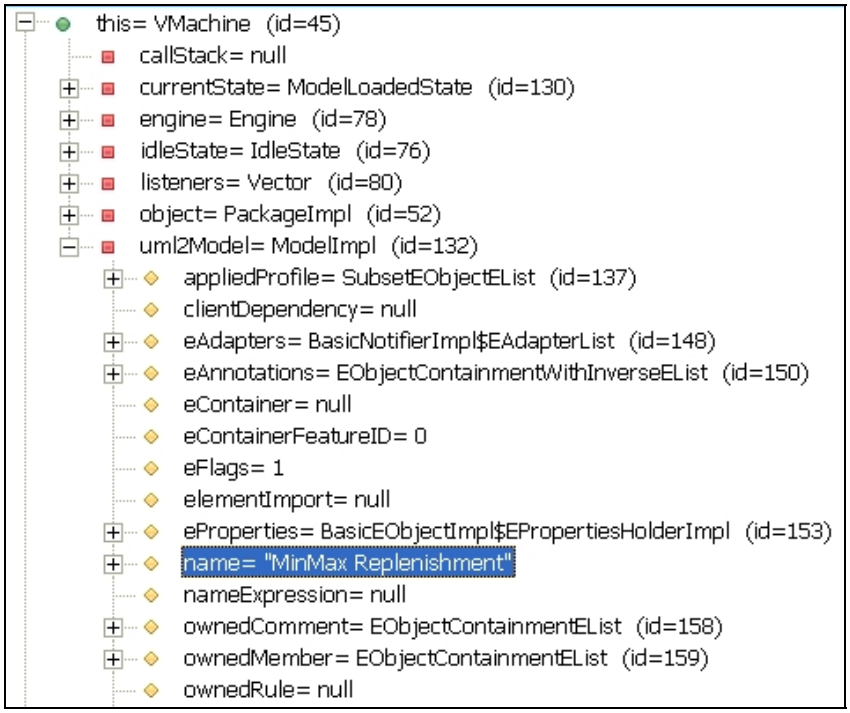

Figure 31 UMLexe when model is loaded

Figure 32 illustrates the context of UMLexe when two instances are created, AJBMinMaxService and AKBKAuctionService.

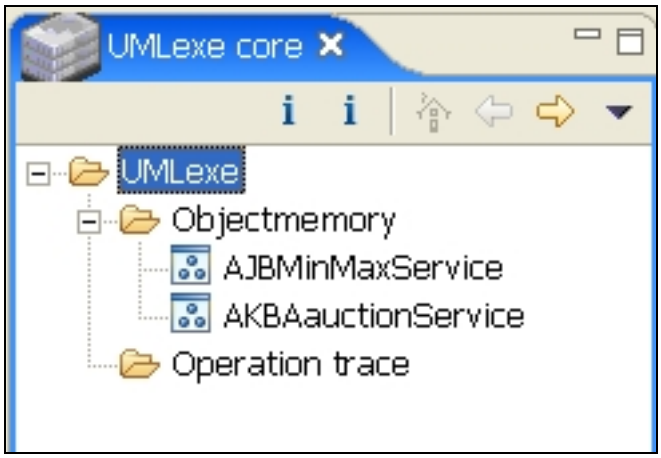

Figure 32 UMLexe with two instances

When these two instances are created the memory structure is described in Figure 33. As illustrated objectMemory contain two InstanceSpecificationImpl objects, although only one is expanded.

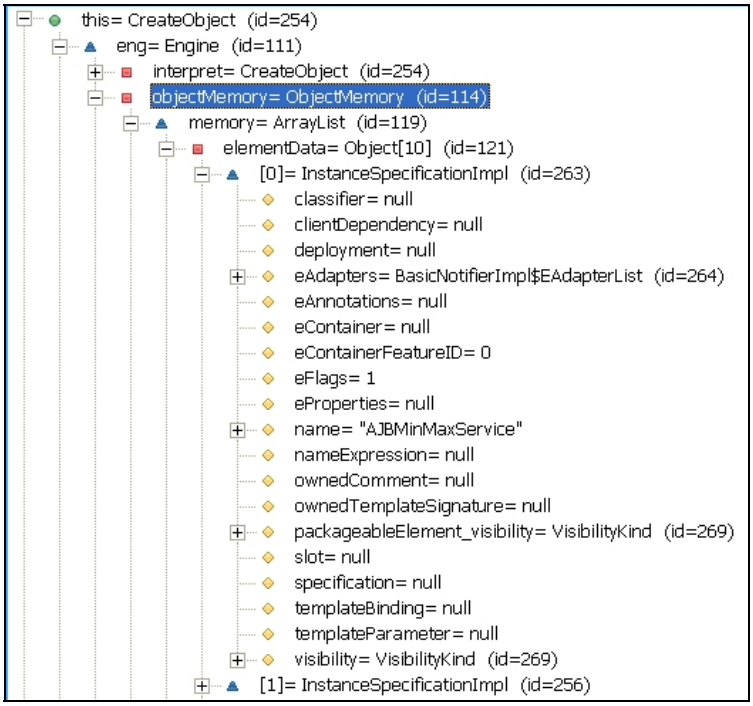

Figure 33 Memory with two instances

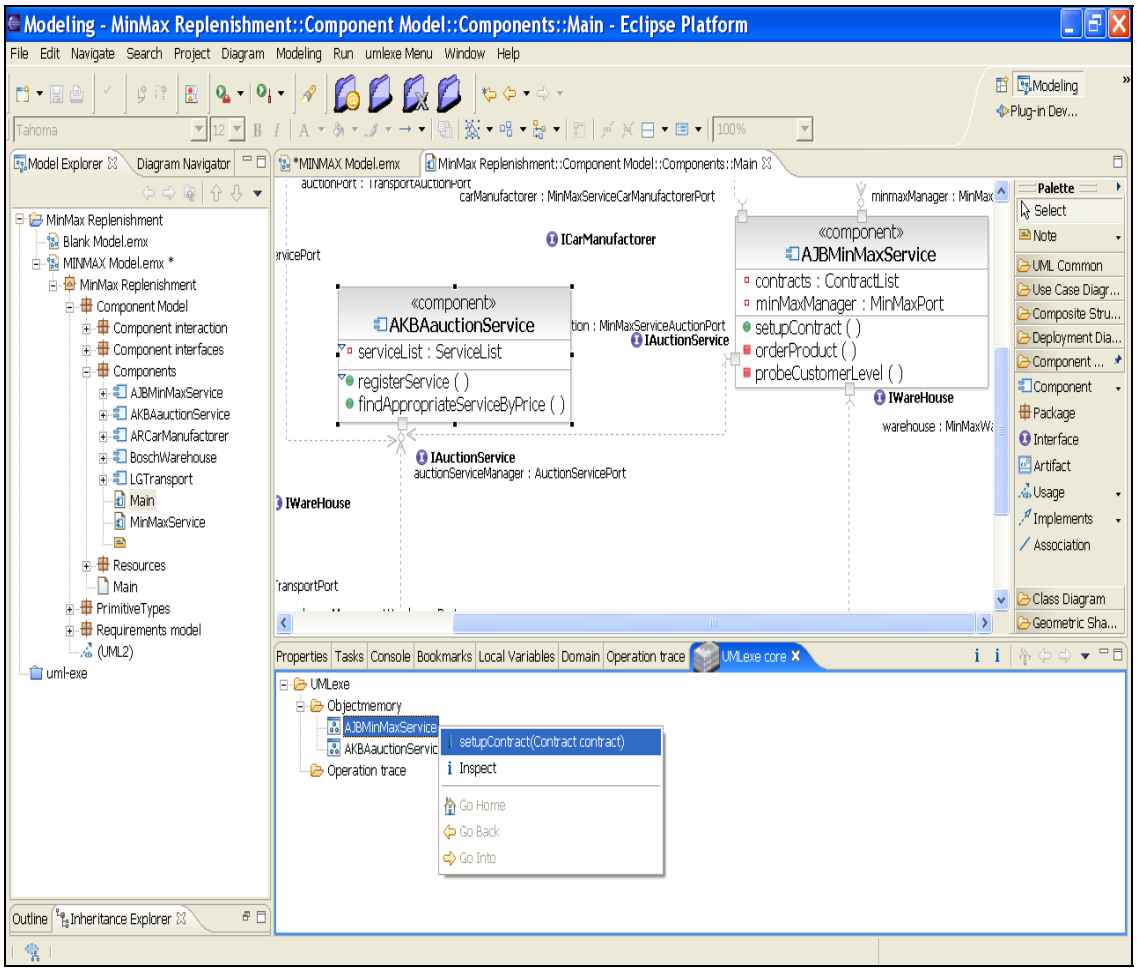

Figure 34 Execute model

Figure 34 illustrates how we trigger execution of an operation on a component instance. In the UMLexe view we drill down to the AJBMinMaxService in object memory and right click this instance. Then a pop up menu appears with available operations to activate. As illustrated, the only operation available is “setupContract” which requires a contract parameter. As mentioned earlier we are not able to add any values to variables in this version of UMLexe. The next thing to do is do click on this setup contract. This triggers the operation to execute and the result of this operation is described in Figure 35.

When setup contract is triggered, it finds interaction associated with this operation. Setup contract interaction represents two properties of AJBMinMax component, minMaxManager and contractList. The minMaxManager is a port and handle all incoming messages on AJBMinMaxService. As described in Figure 35 there are two messages sent, validate Contract which is handled by the minMaxManager and add Contract which is handled by the contracts object. In order to handle these methods we need to add an abstract action language to UMLexe. As we have seen we have achieved executing component operations, but there is still work to be done in order to handle state changes and asynchronous calls.

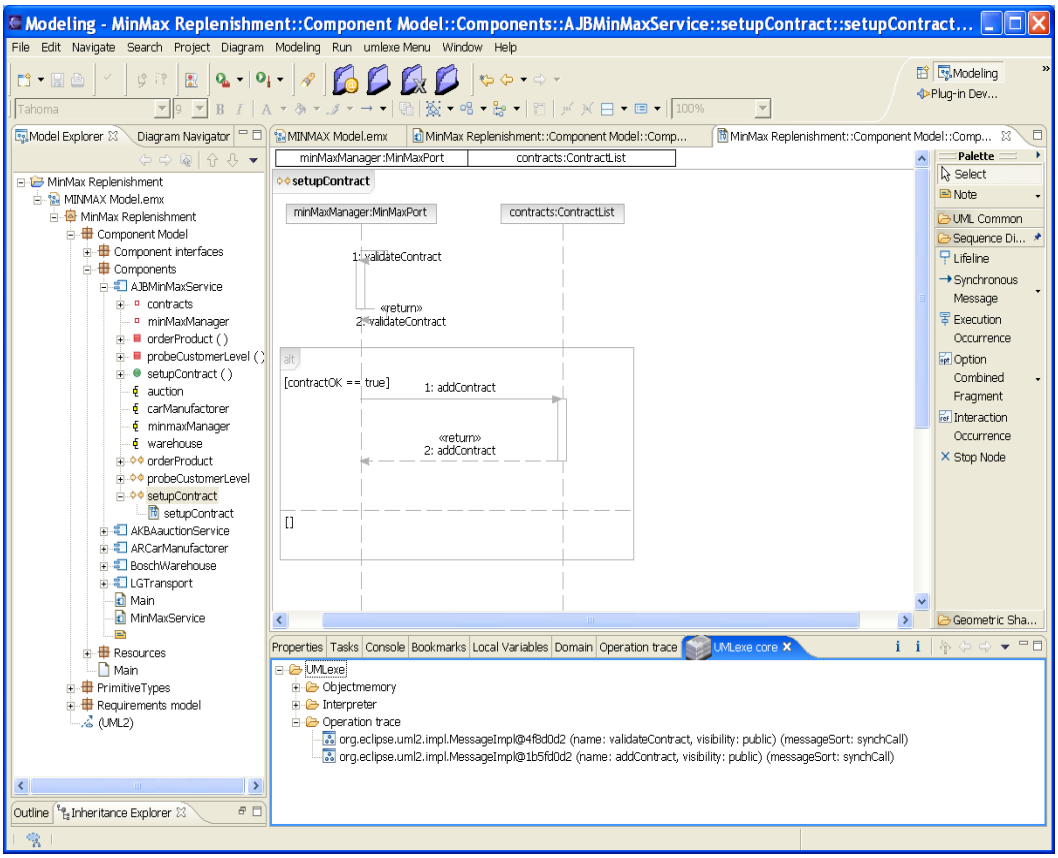

Figure 35 Stack trace

When UMLexe is finished executing a model it returns a stack trace to the workbench. This triggers the workbench to update its views. Within this example the operation trace is returned from UMLexe with two operation calls. In Appendix 13 there is a stack trace log which describes an execution of a more detailed example described in Appendix 12.

8 Discussion

In this chapter we discuss how our own approach, UMLexe, perform in comparison with state of the art described in previous chapters. The discussion is centered on the requirements for UML virtual machine specified in Chapter 3.2. Given the description and results of the previous chapters, it should come as no surprise to the reader that in this chapter we argue that UMLexe match or exceed the state-of-the-art in the context of requirements defined. Based on the platform independent description we argue that UMLexe will provide us with dynamic UML models that are able to execute operational behavior when all parts are implemented.

8.1 Evaluation of UMLexe

UMLexe as described in Chapter 6 implements only parts of the requirements described, although we have implemented interpretation of collaborations between components in interactions. This version of UMLexe integrates the virtual machine with Rational Software Modeler for creating graphical UML 2.0 models, although there is only one way interaction between these tools, actions sent to the model only goes from the modeling tool to UMLexe. Notifying the graphical diagram is not implemented but is leaved to further study on how to interact with the graphical elements in Rational Software Modeler.

8.1.1 MOF 2.0 support (R1)

UMLexe has support for MOF 2.0 by using the EMF 2.0 but with the limitations of not having an “instance of relationship” between MOF and UML Meta level; it is not possible to reflect the changes between these layers except when we recompile the UML 2.0 level. If we change the UML Meta model we have mechanism to look for changes in the Meta layer and we need a set of rules to handle the changes to update the lower level correctly. If we had direct relation between MOF level and UML level we would automatically been notified when changes occurred. For the main goal of executing UML models we include EMF to support model navigation.

Another nice feature UMLexe implements is the ability to import a subset of a model. This is done by defining Ecore objects as input parameters in the UMLexe provided interface UMLexe using the deploy method.

8.1.2 UML 2.0 support (R4)

UMLexe has support for UML 2.0 by including UML2 Meta classes. It means it has support for the subset of UML 2.0 according to the current standard. For instantiating and sending messages between roles within a component it is crucial to support UML 2.0 components, interactions and runtime instances.

Components are supported and it is possible to bind provided operations on components to internal specifications like interactions, activities and state machines. Furthermore, in UMLexe components are instantiated and act as objects. The components operation behavior is specified by interactions.

Interactions define a set of roles as lifelines. These roles are mapped to defined objects or components roles a component. This is achieved by the “represents” attribute in a lifeline. The object “represents” must be defined in the component. In this version of UMLexe it is only possible to send synchronous messages between roles in an interaction. UMLexe supports the minimum requirements for the specification part of dynamic UML models.

In order to specify a component’s operation it is possible to combine interaction by sending messages to gates and link interactions together. In this version of UMLexe I have used Rational Software modeler and this tool does not have support for UML 2.0 gates graphically.

Runtime instances represent components and their internal structure in object memory. UMLexe provides the possibility to create new instances from a component specification. The instance specification has a reference instance to the classifier. Furthermore, the components operations are specified by interactions and these are looked up when a message is sent to a component. UML2 do not provide features for creating instances from a specification directly, but this is provided by UMLexe by adding feature for linking runtime instances to component specification and creates runtime instances based on these specifications.

UMLexe supports the minimum subset of UML 2.0 specification in order to create dynamic UML models. It is possible to create runtime instances from components and it is possible tie behavioral specifications like interactions to a components operation.

8.1.3 XMI 2.0 support (R4)

In order to load and save UML 2.0 objects UMLexe has support for loading XMI models into object memory and saving the current model to an XMI file. This is achieved by using the EMF facilities for loading and storing XMI files.

8.1.4 Support for modeling tool integration (R5)

UMLexe support one-way communication with Rational Software Modeler, meaning it is not possible to interact with the UML diagrams. This feature should be implemented in the next version in order to colorize the diagram showing the model execution. On the other hand UMLexe have a two way communication with the application viewer. The application viewer is an Eclipse plug-in showing the stack and object memory in UMLexe.

To summarize, UMLexe do not provide full support for dynamic UML models in order to synchronize with UML diagrams, but changes in modeling tool can be loaded into UMLexe and this will reflect changes in the system behavior.

8.1.5 Support for OCL (R6)

OCL is not provided by UML2 and therefore not supported in this version of UMLexe. It is possible to use libraries from existing tools as the one provided by Rational Software Modeler. As an effect of using a Rational provided library UMLexe will be dependent of the development of Rational Software Modeler which is not the intention of UMLexe. UMLexe provides an open source development strategy and should thus include or implement an open source OCL or XOCL library.

8.1.6 Support for abstract action language

UMLexe do not provide an abstract action language for evaluating executable expressions on models. Today there are three different textual notations for an abstract action language and we should implement parsing mechanisms for one of these proposals in a later release. As consequence, UMLexe implementation is not able to control combined fragments. This must be considered in a future release.

8.1.7 Model validation (R7)

UMLexe take advantage of the model validation feature provided by the UML2. This feature is included in the model loading use case in the engine. It ensures that the model is valid according to the UML 2.0 Meta model. If the model is not valid it is not loaded into UMLexe. Furthermore, model elements must be validated when UMLexe interprets the model according to the execution semantics in the UMLexe engine. For instance, if we try to send a message to a component operation, there must be an interaction specified and linked to this operation. If this is not the case, the model is not valid according to the rules in UMLexe. This feature is not supported in this version of UMLexe but is added to the development task list.

8.1.8 Model execution (R8)

UMLexe environment has an engine which interprets a system model. It is possible to create new component instances and its internal structure based on the UML specification. Furthermore, the active component is loaded into the interpreter when the interpreter receives a message on a components operation. This message looks up the interaction, which defines the internal interaction of a component and the roles which fulfill the task requested.

8.1.9 Architecture supporting dynamic models (R9)

One of the main goals of supporting executable models is to implement a reflective architecture. UMLexe partly implements a dynamic architecture, meaning it is possible to change model elements in M1 level of OMG's Meta hierarchy, having these changes are reflected to the runtime instances. Changing model elements at M2 and M3 are not supported in this version of UMLexe. In order to do changes in these levels we need to recompile the java classes representing these levels. MOF M3 level is indirectly supported by using the EMF library in order to query a UML model, but as mentioned earlier there is only a conceptual connection between these layers, because there is no reference between a UML class object and an EMF class object.

Table 30 shows a summery of the UMLexe evaluations.

ID	Requirements	Borland Together	Enterprise Architect	Rational Software Modeler	iUML	Baby UML	UMLexe
R1	MOF 2.0 support	JMI	No	Yes, EMF	No	No	Yes, EMF
R2	UML 2.0 support	Partly Meta model support	No	Yes, UML2	No	No	Yes, UML2
R3	XMI 2.0 support	No	No	Yes	No	No	Yes
R4	Support fort tool integration	No	No	Partly, through Eclipse	No	Yes	Yes
R5	Support for OCL	No	No	No	No	No	No
R6	Support for abstract action language	No	No	No	Yes	Yes	No
R7	Model validation	Partly	No	Yes	No	No	Yes, UML2
R8	Model execution	No	No	No	Yes	Yes	Yes
R9	Architecture supporting dynamic models	No	No	No	No	Yes	Yes

Table 30 Evaluation summary

8.1.10 Summary of tools providing execution mechanism

In this section we summarize the evaluation of the tree tools supporting execution mechanism. As described in Table 30 UMLexe take advantage of the new UML 2.0 standard and their alignment with MOF. As described in state-of-the-art existing tools use state machines to describe system behavior, which is the most precise notation for specifying behavior in UML 1.x. With UML 2.0 we can combine structural elements such as components and behavioral elements such as interactions to execute system behavior. With new elements in interactions such as flow mechanisms, we can describe and execute operational behavior with interaction models. With this execution engine pattern it seems that it should be possible to implement a general execution engine for a domain language which is based on MOF and the reflective architecture. Defining an open architecture where the compiler and interpreter is objects part of the architecture will give the opportunity to define the operational semantic for each new domain language. In addition, this could be the architecture for transformation tools as well, where transformation objects is part of the objects within this architecture.

9 Conclusion

The introduction the problem of executing UML models has received a lot of academic and public attention. Especially after the introduction of Model Driven Architecture [21] and tools which supports this concept. We believe that our results constitute a significant contribution to this field, results which we summarize and conclude upon in this chapter. We will also point out areas for future work.

9.1 Concluding summary

In this thesis, we initially established requirements for a UML virtual machine supporting dynamic models. With these requirements we evaluated existing tools and state-of-the-art resulting in functional requirements for our approach. Then we proposed a solution for a UML virtual machine, called UMLexe. It contains an executable subset of UML and necessary operations for the UML virtual machine. UMLexe was described in terms of a platform independent description showing how the virtual machine would execute UML models. The implementation of UMLexe is explained and even though UMLexe was not fully implemented, the core parts of the mechanism were implemented and provided us with valuable information for future work.

The vision of UMLexe was to provide the possibility to execute operational behavior on components specified with interaction diagrams. UMLexe has partly created architecture in order to support dynamic models and depends on the architecture of Java. Therefore it is not possible to dynamically create new M2 level elements without recompile M2 classes and reflect these classes into the Java virtual machine. Still it is possible to create dynamic UML models from the minimum UML 2.0 subset described in Chapter 4.1.2.

With the current version of UMLexe we used Rational Software Modeler to create UML 2.0 models. This choice caused some problems concerning interaction with the graphical elements in a diagram, because as far as we has experienced, this tool do not provide interfaces for manipulating diagram elements. Consequently, UMLexe do not provide visualization in diagrams of the execution.

Our proposal combines static structure and behavioral elements to define an executable subset of UML which executes operational behavior on components. However, this pattern of dynamic models can be used on other executable models as well, such as business process models or other domain specific languages. The next step should be to create a virtual machine for models in general. Having an open architecture and the potential to change the behavior of the virtual machine could facilitate in achieving this. A kind of virtual machine could be created from the ideas of dynamic UML models.

The conclusion concerning the UML virtual machine and dynamic UML models is mainly based on a theoretical comparison, since there is no tool supporting dynamic UML 2.0 models except BabyUML, which is based on the Smalltalk language. We are, however satisfied with the progression of the implementation of UMLexe, considering the fact that the final UML 2.0 specification is not released yet.

9.2 Future work

The most obvious area of future work is to fully implement UMLexe in such a way that all described requirements are handled. It would then be natural to carry out substantial empirical tests. The next sections are concerned with different aspects of how to fully implement extensions and improvements to UMLexe.

9.2.1 Architecture

As a foundation for our work we used the ideas from Squeak [13], babyUML [9] and the work from Dirk Rhiele [25] in order to support an dynamic architecture. This work showed that the Java runtime environment has constraints on interacting with the runtime system objects. It is only possible to have a conceptual connection between M2 and M3 level objects. In order to change the language and environment at runtime there must be an instance of relationship between M2 and M3 [9]. UMLexe contributes with the instance of relationship between M0, M1 and M2 in the OMG's Meta hierarchy. A dynamic architecture also requires an instance of relationship between M2 and M3 level. Furthermore, we should be able to access the system objects of the runtime system. For instance we should be able to change the interpreter's behavior and the language it interprets. An alternative way of implementing a dynamic architecture is to create a Squeak environment [13] with Java that supports the OMG's Meta hierarchy. Then UMLexe should provide an UML interaction compiler and byte code instructions m which the runtime objects are able to execute in order to speed up the execution.

9.2.2 UMLexe core

In UMLexe it is implemented support for interpreting interactions. These interactions represent operations in components and I found that it should be easy to link activities and state machines to operations as well. UMLexe contributes with the ability to instantiate and execute UML static and behavioral specifications, but the UMLexe interpreter should be extended with rules for interpreting activity and state machines.

9.2.3 Modeling tool integration

UMLexe have integrated with the Eclipse environment and Rational Software Modeler. In order to view runtime instances and the stack trace I have created an application viewer. I have full control over the communication with this tool, but I am not able to interact with the graphical diagram elements in Rational Software Modeler [40]. In order to interact with a graphical diagram a provided interface is needed from the modeling tool. UMLexe contributes with two-way communication with the application viewer and one-way communication with Rational Software Modeler. An alternative way of creating graphical UML models is to create a modeler ourselves. A good base for this work could be seDI [41]. SeDI is an open source initiative supporting interaction diagrams. This tool is created with Java and is based on UML2 and EMF as well. We could integrate UMLexe with this tool in order to create colorizing and show the state of the system graphically.

9.2.4 UML 2.0

UMLexe uses the UML2 implementation of the UML 2.0 Meta model and focus on components and interactions in order to create real objects [9] and specify their provided methods and internal behavior by interactions. UMLexe provides runtime

system objects for creating instances and interpret interactions, but in order to execute statements like 1 + 2, UML needs to add support for primitive methods. The implementation of primitive methods like, arithmetic is not supported in this version of UMLexe. A major improvement to UMLexe would be support for primitive methods in UML 2.0 Meta model in order to send messages to arithmetic or logical operators [14].

The next improvement is to add functionality to use activity models to describe behavior. It is possible to add extra strategy classes to handle activity models the same way as the interactions are handled and the same could be achieved with state machines as well.

The final improvement we would like to propose is the ability to create textual statements that could be parsed and handled by the engine. This could be achieved by adding for instance a XOCL or an action language parser into the engine. Doing this could give the possibility to write statements and send them to the engine which returns the result based on the query. This could for instance be applied within an SQL client where we are able to add statements and send them to the database that returns the query result. This would add valuable features to the UML virtual machine.

9.2.5 Other improvements

UMLexe is built as a standalone single process system. As mentioned earlier this is only a prototype for testing the ability to create an adaptive system based on UML2, EMF and Java. In the future UMLexe should have support for interacting with other systems on a network. UMLexe should implement support for sending messages to web services in order to collaborate with other systems. This will require UMLexe to run as a server process and listen on the network protocol.

10 Bibliography

1. Systemator, *Systemator*. 2005. http://www.genera.no/2052/tilkunde/09.04/data/pages/page_display.asp?doc_id=94
2. OMG, *UML Superstructure*. 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02
3. Clark, T., et al., *Applied Metamodelling A Foundation for Language Driven Development*. 2004. http://albini.xactium.com/content/index.php?option=com_remository&Itemid=28
4. Reenskaug, T., *Working with objects : the OOram software engineering method* ed. P. Wold and O.A. Lehne. 1996, Greenwich, CT: Manning. 1-884777-10-4, 0-13-452930-8.
5. Rumbaugh, J., I. Jacobson, and B. Grady, *The Unified Modeling Language Reference, second edition*. 2004: Addison-Wesley Professional; 2 edition (July 19, 2004). 752 pages. 0321245628.
6. Gelernter, D., *Truth, Beauty, and the Virtual Machine*, in *Discover Magazine*. 1997. p. 72. http://www.webopedia.com/TERM/v/virtual_machine.html
7. Rhiele, D. *Architecture of a UML virtual machine*. in *OOPSLA*. 2001. Tampa Florida.
8. OMG, *Request for proposal on executable UML*. 2005.
9. Reenskaug, T., *A rudimentary UML virtual machine as a Smalltalk Extension*. 2002.
10. Breton, E. and J. Bazivin, *Towards an understanding of model executability*, in *Proceedings of the international conference on Formal Ontology in Information Systems - Volume 2001*. 2001, ACM Press: Ogunquit, Maine, USA. p. 70-80.1-58113-377-4.
11. Chang, J.P., D. Tolbert, and D. Mellor, *Common Warehouse MetaModel developers guide*. 2003: Wiley publishing. 0-471-20243-6.
12. JVM, *Java Virtual Machine*. 2005. http://www.java.com/en/
13. Squeak, *Squeak*. 2005. http://www.squeak.org/
14. SMALLTALK, *Smalltalk*. 1980.
15. Sunye, G., A. Le-Guennec, and J.M. Jezequel. *Using UML action semantics for model execution and transformation*. 2002: Elsevier.
16. Mellor, S.J.M., *Software-platform-independent, Precise Action Specifications for UML*. ZDNet UK, 2004. http://whitepapers.zdnet.co.uk/0,39025945,60093279p-39000377q,00.htm
17. MOF-2.0, *MOF 2.0 Query views and transformation specification*. 2003. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF
18. OCL-UML-2.0, *UML 2.0 OCL Specification*. 2003, OMG.
19. XMI-2.0, *XML Metadata Interchange (XMI) Specification*. 2003, OMG.
20. Forman, I., *Putting metaclasses to work*. 1998: Addison Wesley Longman. 300. 0201433052.
21. Kleppe, A., *MDA Explained*. 2003: Addison-Wesley Professional; 1st edition (April 25, 2003). 192 pages. 032119442X.
22. Carter, K., *iUML*. 2005. http://www.kc.com/download/index.html

23. Alcatel, et al., *Action Semantics for the UML*. 2001. p. 263. http://www.omg.org/docs/ptc/02-01-09.pdf
24. Reenskaug, T. *IS21C Information Systems for the 21st Century*. in *Roots 2004*. 2004. Bergen. http://heim.ifi.uio.no/~trygver/2004/rOOts-04/IS21cHandout.pdf
25. Rihle, D., et al. *The architecture of a UML virtual machine*. in *OOPSLA*. 2001. http://www.riehle.org/computer-science/research/2001/oopsla-2001.html
26. UMLVM, *UMLVM*. 2005. http://umlvm.cs.umb.edu/index.html
27. Yokote, Y., et al., *The muse object architecture: a new operating system structuring concept.* SIGOPS Oper. Syst. Rev., 1991. **25**(2): p. 22-46.
28. Together, B., *Borland Together Eclipse edition*. 2005. http://www.borland.com/together/pdf/tgr_techview.pdf
29. JMI, *Java Metadata Interface*. 2005. http://java.sun.com/products/jmi/
30. Gerber, A. and K. Raymond, *MOF to EMF: there and back again*, in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*. 2003, ACM Press: Anaheim, California. p. 60-64.
31. EMF, *Eclipse Modeling Framework*. 2005.
32. UML2, *EMF-based UML 2.0 Metamodel Implementation*. 2005.
33. Berre, A.-J., et al., *Component and Model-based development methodology*. 2004, SINTEF. p. 193.
34. Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation* ed. Addison-Wesley. 1983: Addison-Wesley.
35. Louden, K., *Compiler construction, Principles and practice*. 1997: Thomson.
36. Gamma, E., et al., *Design Patterns, elements of reusable object oriented software*. 1998: Addison Wesley.
37. Eclipse, *Eclipse*. 2005.
38. SPRING, *SPRING*. 2005. p. Java framework for component oriented programming.
39. Zhang, J., L. Yuehua, and Y. Gray, *Generic and Domain-specific Model Refactoring using a Model transformation engine.* 2004.
40. RSM, *Rational Software Modeler*. 2005.
41. Limyr, A., *seDI*. 2005.

11 Appendix UMLexe model platform specific models

Figure 36 describes the UMLexe component and the application viewer and how it is implemented in Java. The interfaces make up the façade for the UMLexe component. These interfaces are implemented in the packages impl, core and modelmanager.

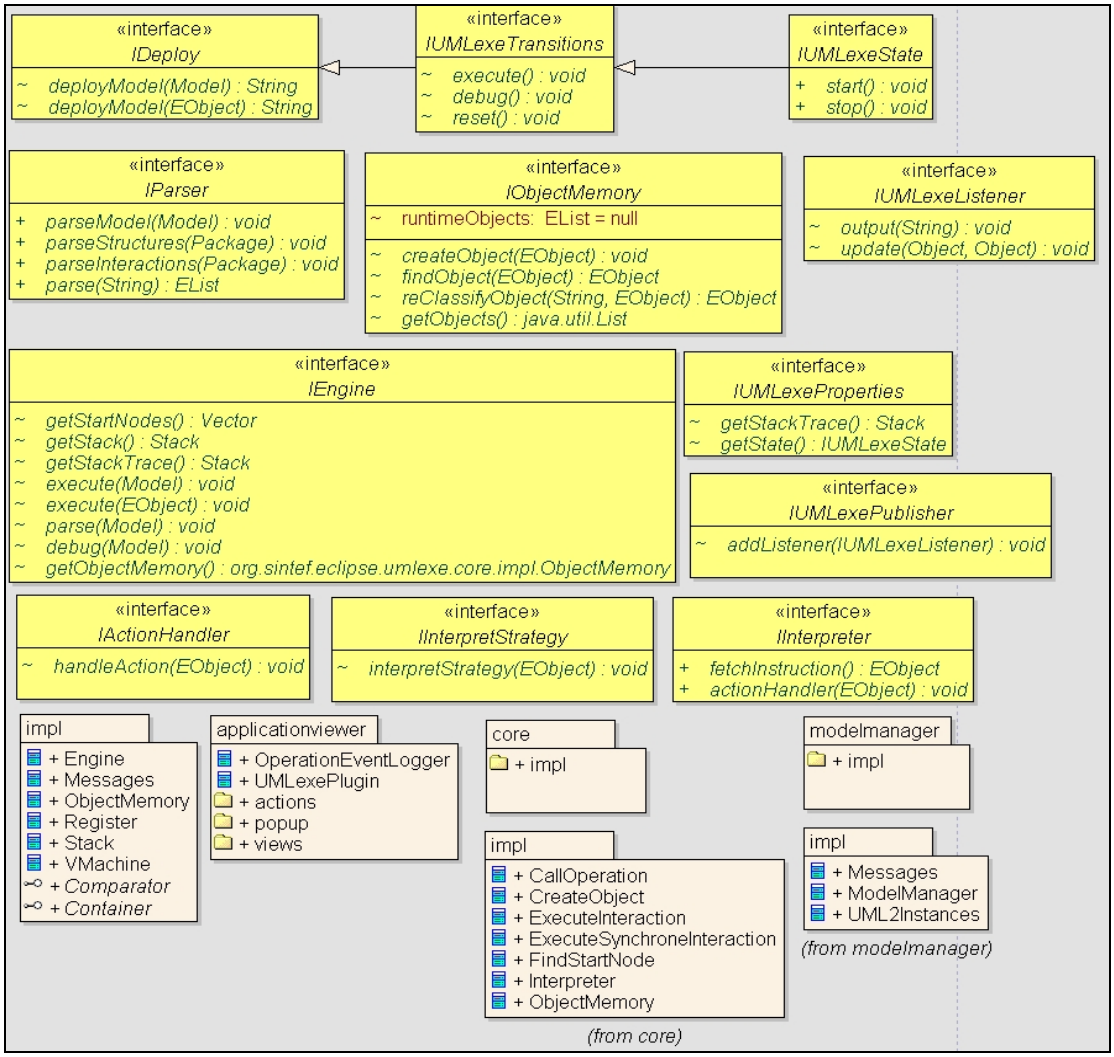

Figure 36 UMLexe model

The application viewer is the client application that interacts with the Rational Software Modeler and Eclipse. This tool is implemented as a plug-in to Eclipse and is derived from Eclipse's abstract plug-in class. The application viewer package is further described in the next section.

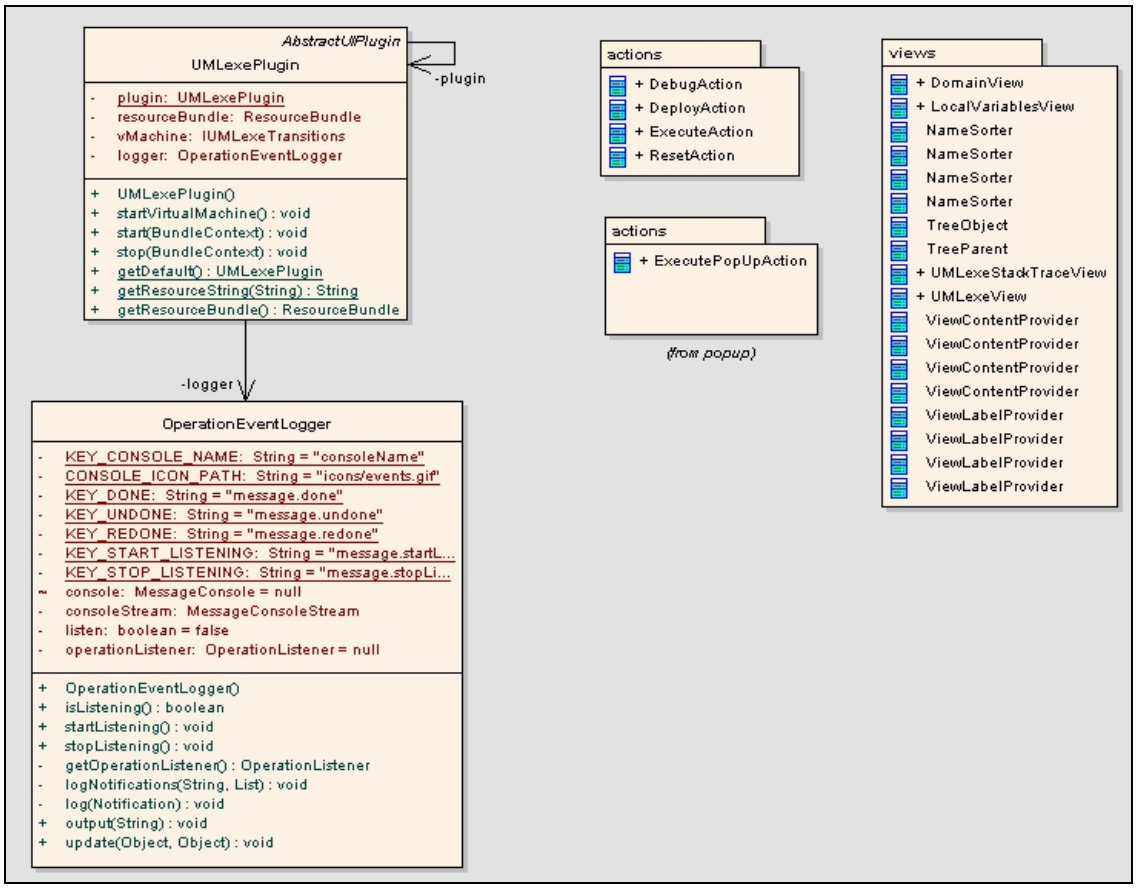

Figure 37 Application viewer model

Figure 37 shows the classes and packages which make up the application viewer. The UMLexe plug-in is derived from the abstract plug-in from the Eclipse platform. The abstract plug-in is loaded into the Eclipse environment by traversing the plug-in directory in Eclipse. All plug-ins within this directory are loaded by reflection when Eclipse starts. The same routine loads the menus, pop-up menus and views during start-up as well. The OperationEventLogger is logging component handling logging within the UMLexe plug-in.

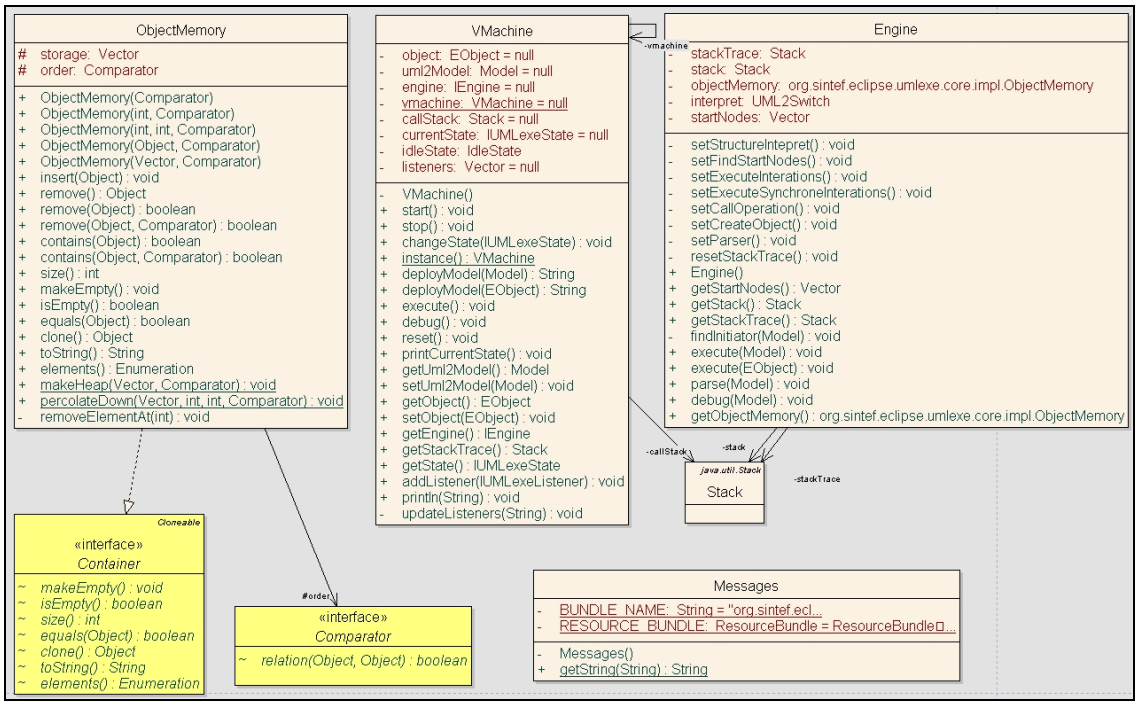

Figure 38 UMLexe impl model

Figure 38 describes the internal classes in the UMLexe component. The ObjectMemory handles all model and runtime objects within UMLexe. VMachine is the façade implementation of UMLexe. This class is the port of the UMLexe against its clients. VMachine implements the interfaces for interacting with the internal parts of UMLexe. The engine executes instructions given from the model manager.

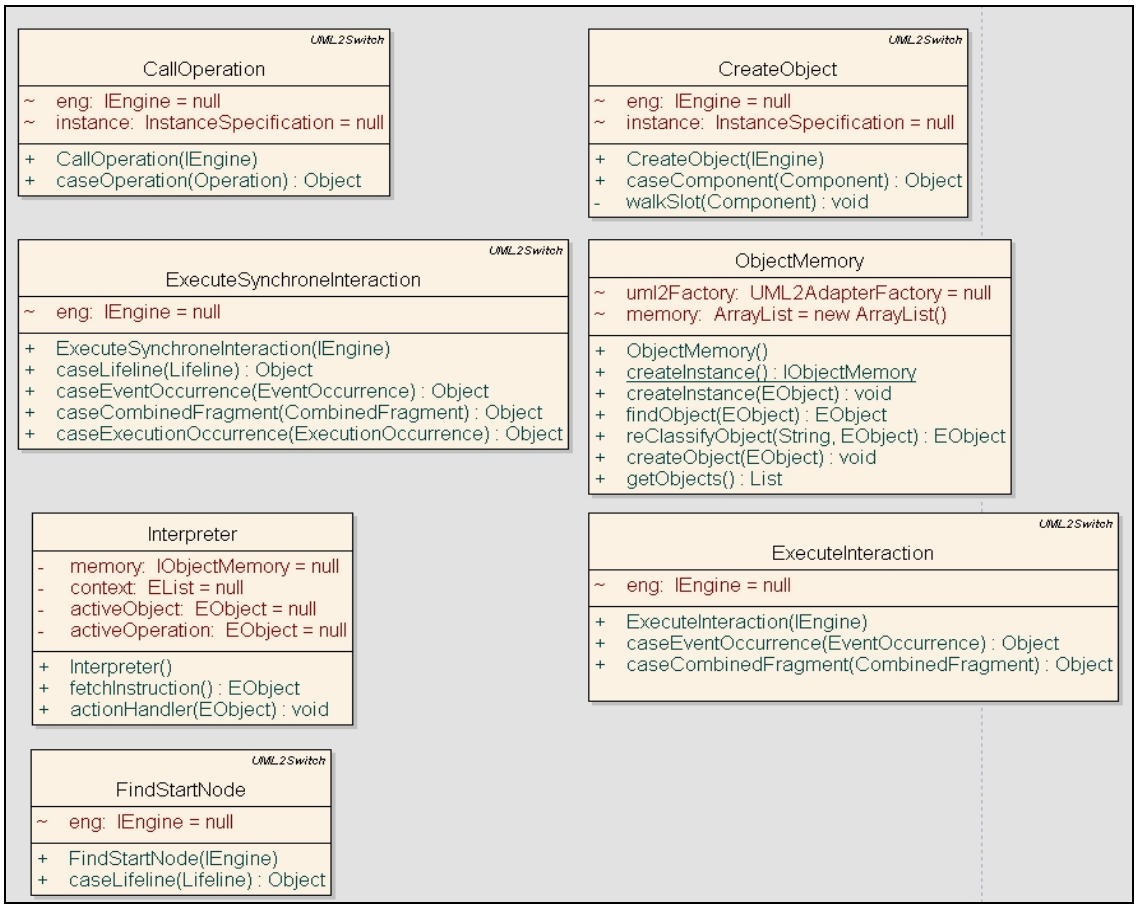

Figure 39 Core impl model

Figure 39 gives the details about the interpreter and the strategy pattern used for parsing the components and interaction objects. The interpreter is implemented as a strategy pattern. At the momemt there are rules for executing interactions with synchronous messages, Create Object and CallOperation.

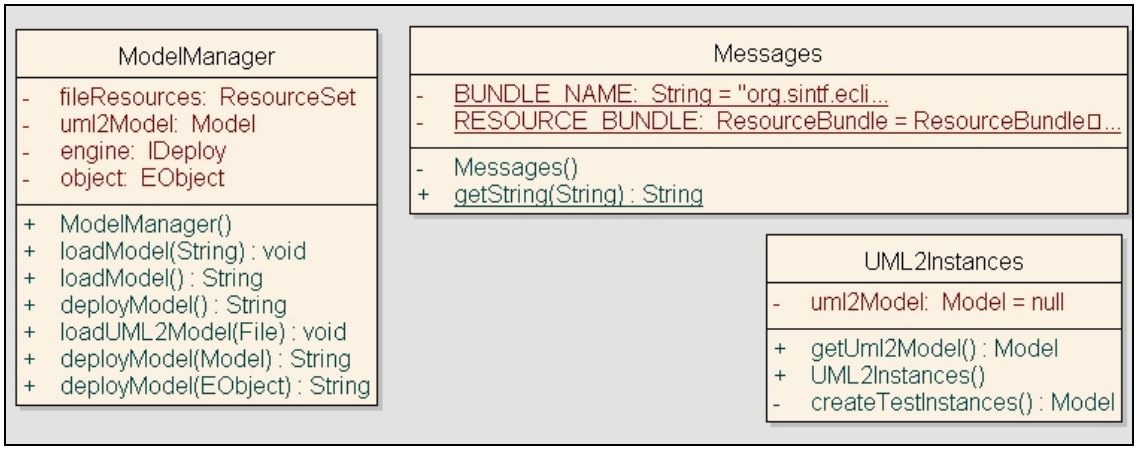

Figure 40 ModelManager impl model

Figure 40 described the model manager and the operations for importing UML models and stores it into UMLexe and the core object. The ModelManager class acts as a mediator between a client and the UMLexe engine. This class handles both models as XMI files and as EObject type, which is an abstract class within the EMF framework containing the model graph.

12 Appendix Min-Max replenishment

This appendix describes one of the roles represented in the min-max replenishment case. This model describes the internal and external collaboration for the AR car manufacturer.

12.1 Business model

The business model describes the context for the AR car manufacturer. As in put to this model have been the scenario description presented earlier and a process description.

12.1.1 Scoping statement

12.1.1.1 Context statement

AR car manufacturer produce sports cars with high quality and in the high end market area. This company is divided into four departments and an effective collaboration between these departments is the key aspect for giving customer the right service. In addition, the company focus on improving relations to external actors to achieve the company's strategically goals.

To achieve the operational goals, some parts of the production process needs to be automated. Especially administrative warehouse routines and order processing will be handled within this case. The business process is divided into four departments:

- Market department, calculates the optimal trading stock. In addition, this department has the responsiability to update the key accounts for the external actors that order light bulbs on behalf of AR.
- Order department, knows the order status and stores orders, freight letters et. In the accounting system
- Warehouse department, receives goods, checks the inventory against the order and notifies the order department when there are differences between order and delivery.
- Production department consumes light bulbs during the car production.

AR will be part of a supply chain management program which follows the principles within Software Oriented Architecture (SOA). This will give the opportunity to exchange information independently of platform, programming language and data format. This collaboration is achieved by implementing interfaces for communication between external actors. AR must provide interfaces to the AJB min-max service in order for the min-max service to monitor AR's warehouse. Furthermore, it should be a two-way communication and the min-max service must provide an interface for AR to setup a contract and to change existing contract.

The vision for change chapter describes technologies that is provided today and the necessary changes to achieve automated business processes both internal and externally.

12.1.1.2 Context diagram

Figure 41 describes internal and external actors involved in AR's business process. These roles are further detailed in the table below.

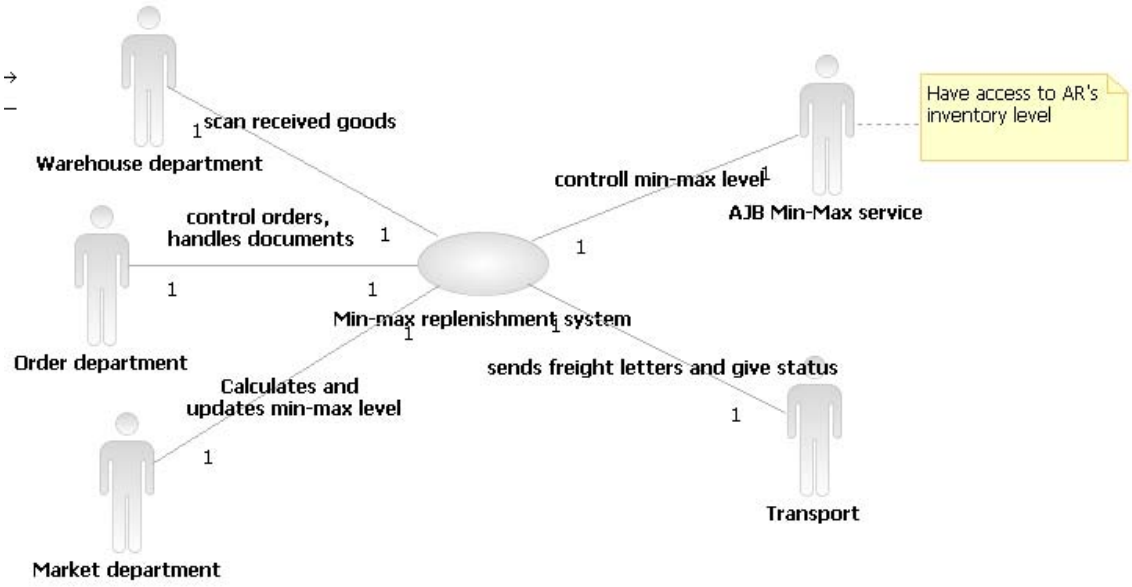

Figure 41 AR context diagram

Actor	Description
Warehouse department	This department have the responsibility for receiving and controlling goods. This is achieved by scanning deliveries and handling differences.
Order department	This department is responsible for invoicing, and need to handle differences between ordered goods and the delivered goods. This is important function because the Min-Max service is dependent of correct inventory information
Market department	This department calculates the need for light bulbs based on prognoses within the market. This information is re-estimated all the time in order to achieve the best min-max level.

External actors are dependent of interfaces from AR for the supply chain management program to work. In this case it is the transport companies and the AJB min-max service.

Actor	Description
AJB Min-Max service	This service buy's light bulbs from light producers. AR is a registered customer which gives the min-max service access to the AR's warehouse.
XE Transport	This service is responsible for delivering goods for the light producer to the car manufacturer.

12.1.1.3 Vision for change

This information system will provide AR with the necessary help to achieve their goals. This system will change the most aspects within warehouse and order department.

For AR it is decided to use bar code scanners and personal data assistant (PDA) to support the business processes. It is assumed that the freight letter is transferred electronically from the transport company. In addition, there will be more focus on using PDA's for validating deliveries and to calculate market prognoses. The following areas can be improved:

Activity	Workflow
Problem	Too much resource is used to exchange documents. These documents might get lost and there is a lot of manual typing which can produce errors.
Improvement	Order department, warehouse and market department will have access the same documents all the time.
How	Orders, freight letters and updated min-max level regularly.

Activity	Warehouse replenishment
Problem	Difficult to achieve acceptable minimum warehouse level.
Improvement	The min-max service handles effective replenishment.
How	The min-max service has access to the warehouse through a web service interface.

Activity	Updating warehouse
Problem	Manual typing can produce errors or the information can be forgotten to be registered.
Improvement	Inventory database is automatically updated with scanning bar codes when new deliveries are arrived.
How	All received goods are scanned and this will give a real time updating of the warehouse information.

12.1.2 Goal model

The goal model describes which business goals this system will support.

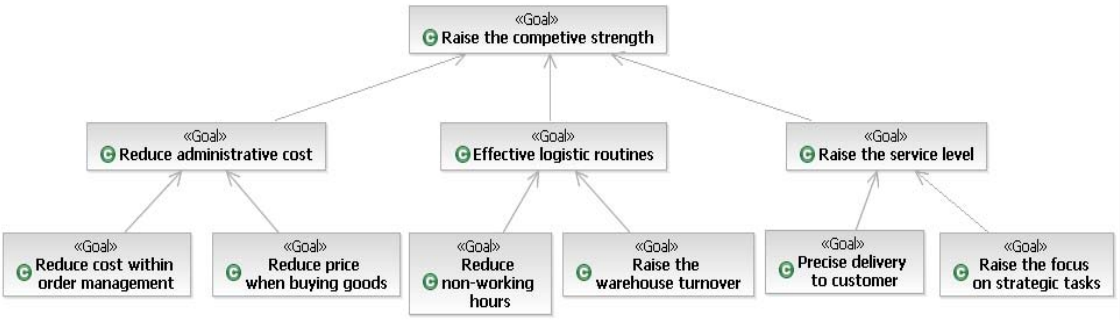

Figure 42 AR goal model

Acceptable integration between the departments is necessary to achieve the sub goals reduce administrative cost, effective logistic routines and raise the service level. Introducing this new information system will raise the collaboration between the departments as a result of more precise information and improved access to this information.

As a result of automating the business processes, will give AR opportunities to focus on strategic tasks. For instance, will the non-working hours in the warehouse be reduced and within the order department will electronic information mean less manual typing.

AR desire higher service level to the customer and it is expected that the car suppliers will get faster access to new cars.

A successful introduction of the supply chain management program will give the car producers support with these areas:

- Delays in the ordering process will be reduced.
- Reduce risk for unnecessary products in the warehouse.
- Right price when buying
- Faster order processes.

12.1.2.1 Goal and process model

Figure 43 describes how the business process support the goals defined. Automating order management can help us with achieving our goals. Good routines for receiving goods can improve the access level for the warehouse. Continually deliveries and consuming of goods will raise the warehouse turnover. The advantage for continually and more predictive replenishment is that the production process will never stop.

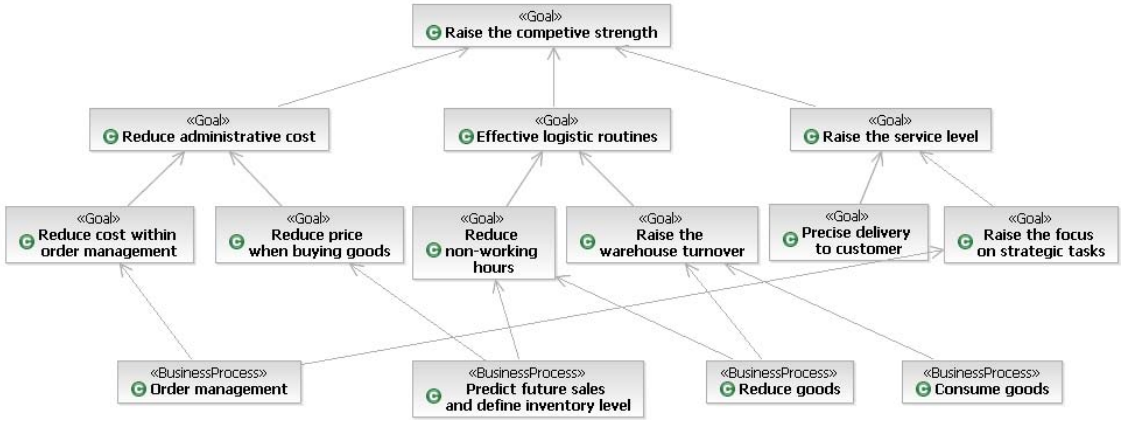

Figure 43 Goal and process model

12.1.3 Community model

12.1.3.1 Business resource model

Figure 44 describes the concepts in the min-max replenishment domain. These concepts are resources the system use and the type of information it provides. The AJB min-max service is a central resource and contains contracts received. The inventory database gives the warehouse level which is connected to the min-max service. Order is another resource which is linked to the light bulb resource. In addition, it is relations between order, invoice and freight letter as well. These resources are stored in AR's database and have because of that a relation to the Accounting system. AR consists of four departments and this is modelled with composites relations. All departments will receive notifications about order and transport status.

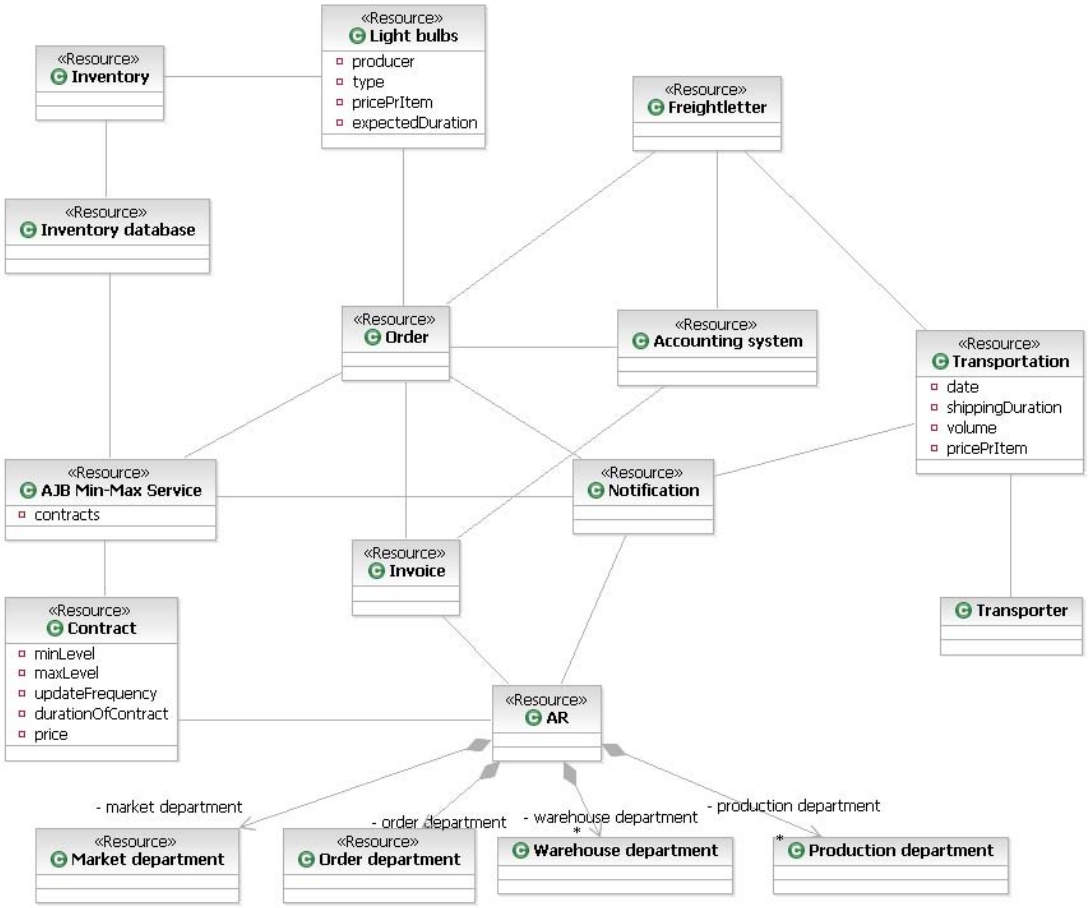

Figure 44 Business resource model

12.1.3.2 Business process and role model

Figure 45 describe the overview of the business process. First, the market department calculates the future sale and setup a contract with the AJB min-max service. This contract contains minimum and maximum level of light bulbs within the warehouse. The min-max service creates orders when it is needed. The last activity describes how the light bulbs are transported and received at AR. The consume activity may happen in parallel and is independent of the other three activities.

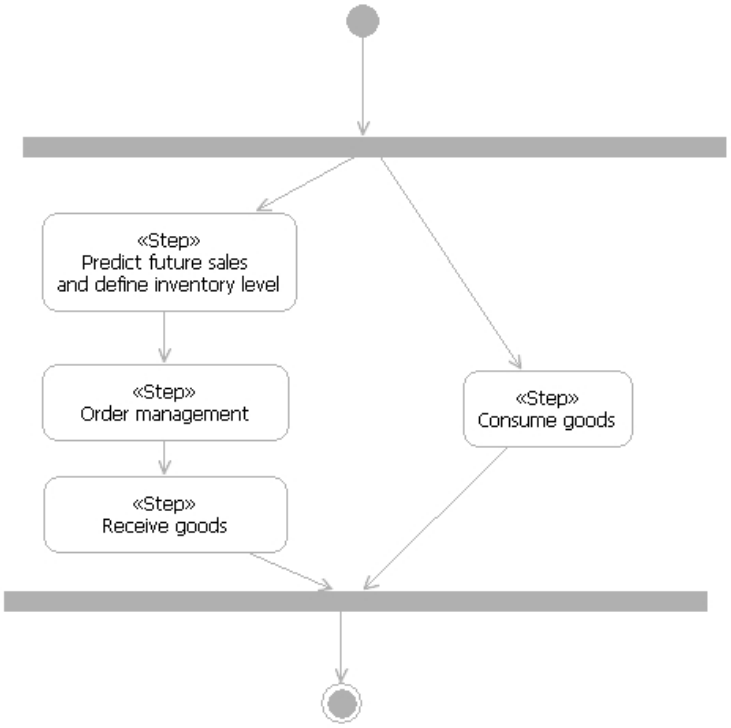

Figure 45 Business process model

12.1.3.3 Work element analysis (WARM)

This section describes the flow within these four activities in Figure 45. These activities are refined using the Work element analysis technique.

12.1.3.3.1 Predict future sales and define inventory level.

Figure 46 describes the first activity "Predict future sales and define inventory level" which is done by an employee in the market department. This is an activity done by a human and it is not necessary done with a computer tool. Whereas, we may assume that this activity is done with help from a computer tool to gather information and do the calculations. That is why this activity is marked with the tool step stereotype.

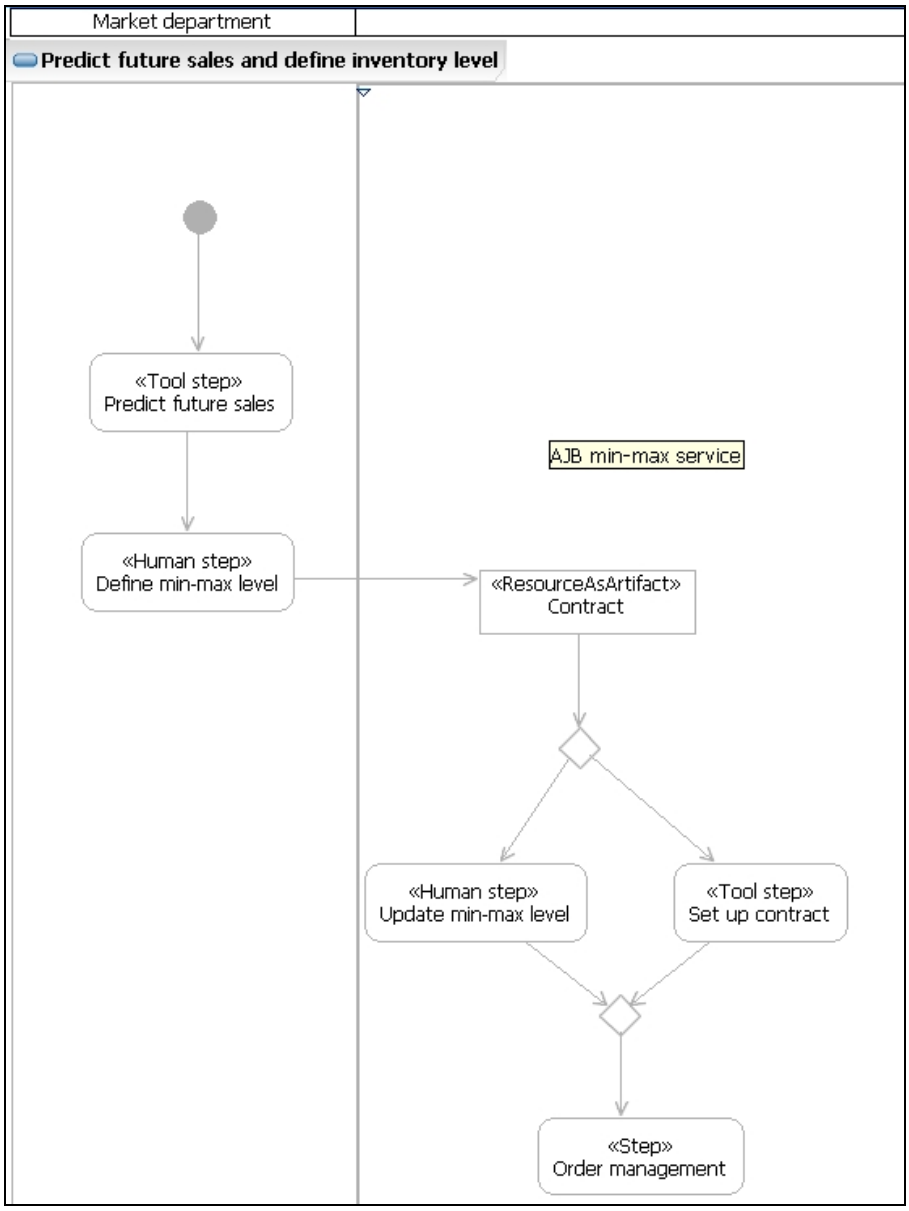

Figure 46 Predict future sales and define inventory level

12.1.3.3.2 Order management

When the contract is setup and the min-max level is defined, the min-max service will check the inventory level in regular intervals. When the inventory level is below minimum level the min-max service create new orders. The min-max service uses the action service to find the most appropriate supplier for light bulbs. When the order is registered, will the order department at AR receive this automatically and store this within the accounting system. When the light bulbs are ready for delivery, the "Receive goods" activity will be executed.

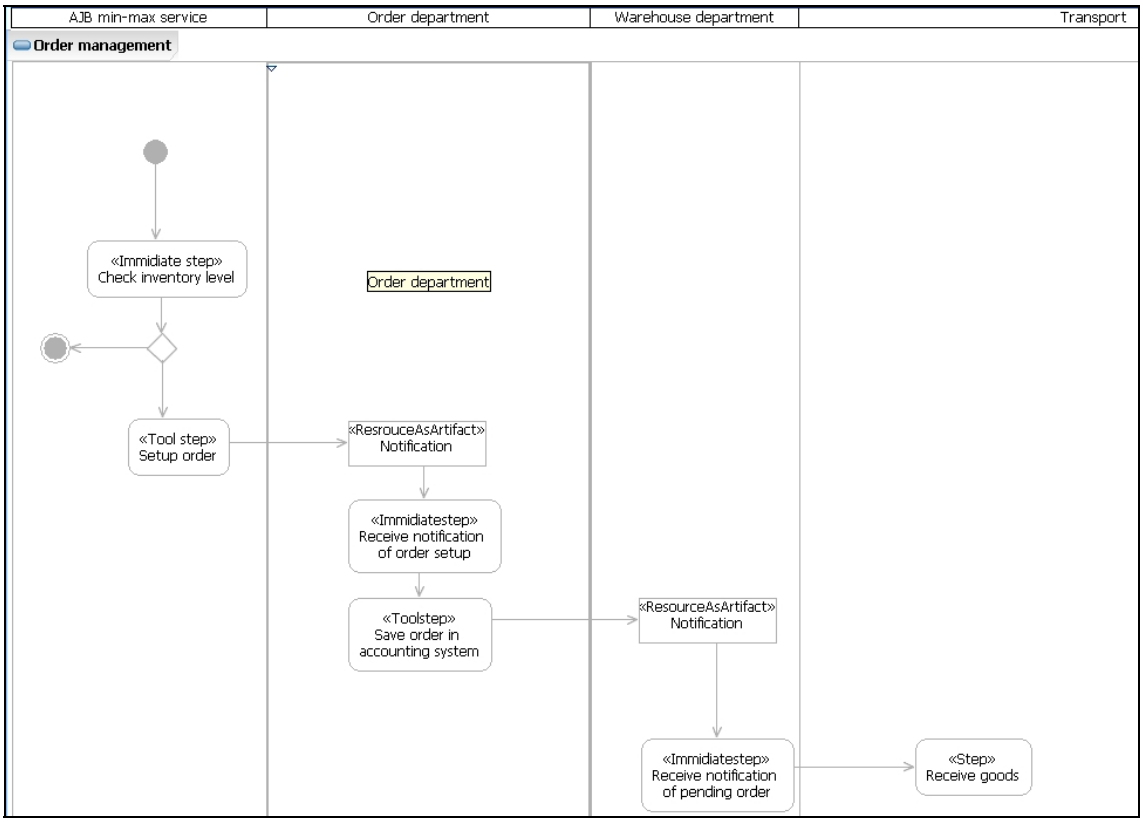

Figure 47 Order management

12.1.3.3.3 Receive goods

The order department is automatically notified when an order is ready for delivery. This message is sent to the warehouse department as well. The warehouse receives goods and the freight letter as well. The freight letter is transferred from the transporters PDA and it is automatically validated in three steps. The information scanned from the barcode is validated against the electronic freight letter and against the order.

If the freight letter is correct, it will be digitally signed and the transporter receives a receipt on the PDA. If there is no order which belongs to the freight the process will terminate.

When the database is updated, it will automatically notify the order department about the order status.

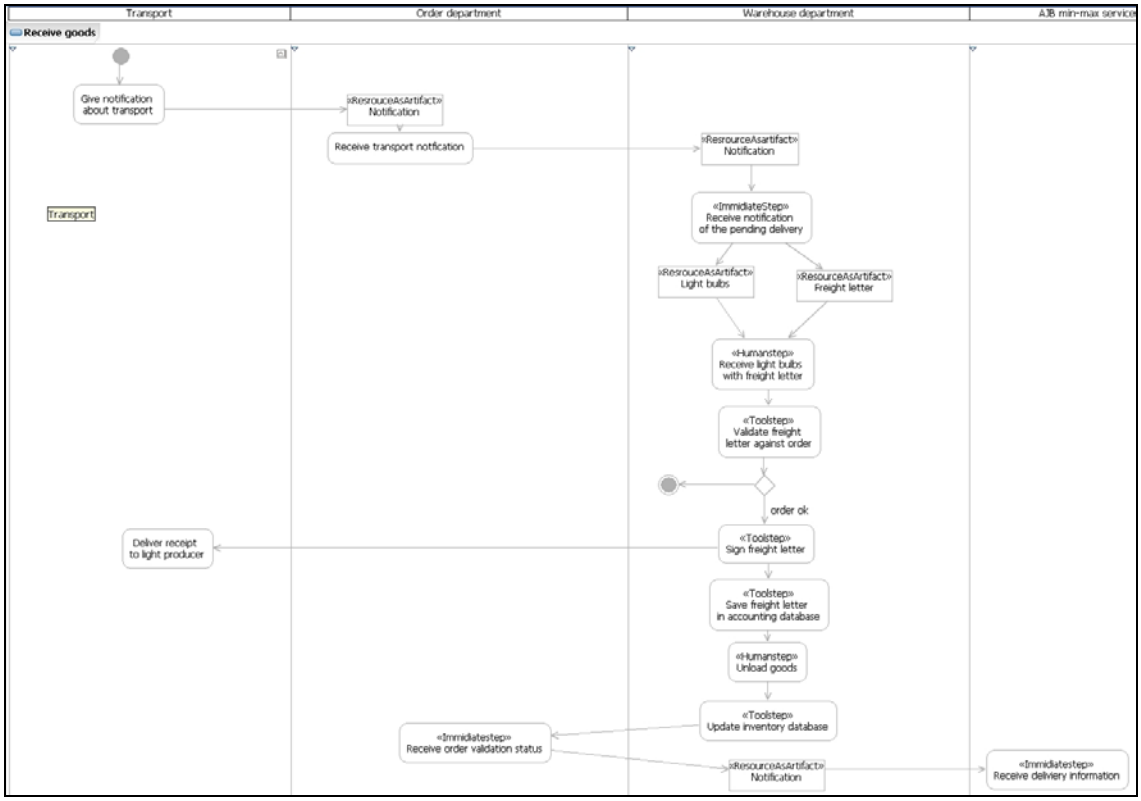

Figure 48 Receive goods

12.2 Requirement model

The use diagrams in this chapter describe how the actors use, maintain, get or give information to the system.

12.2.1 System boundary model

The system boundary model describes the identified actors and use cases. On the right side of the diagram two system actors identified. These actors generate reports to the users of the system.

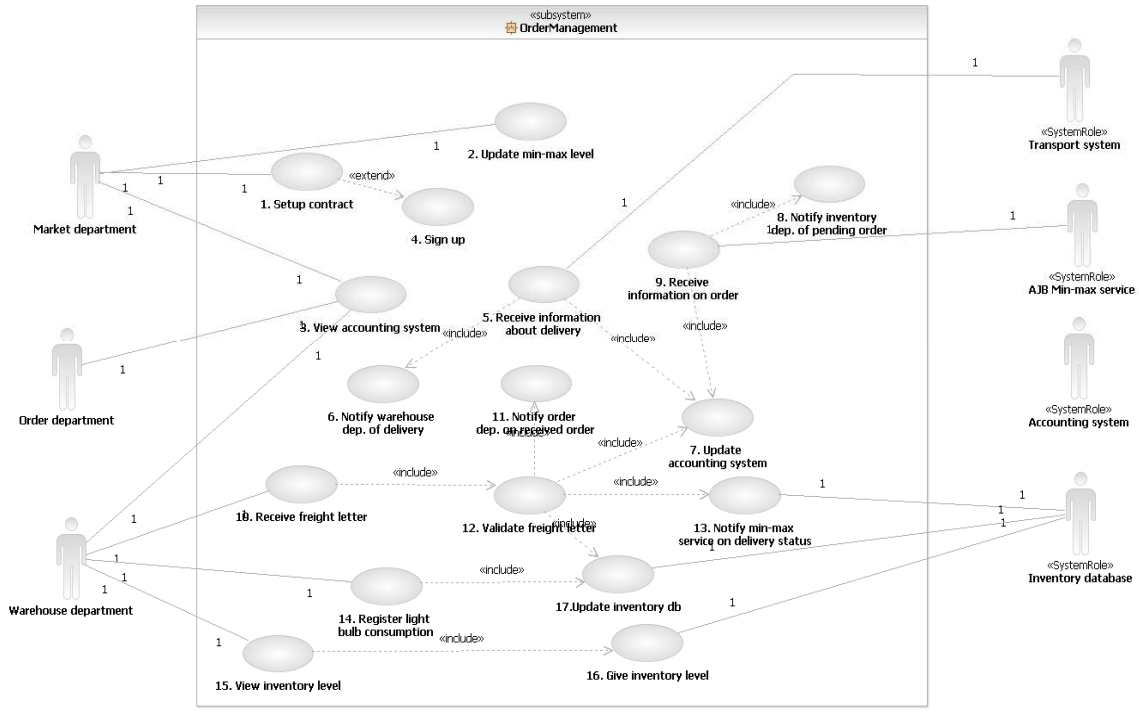

Figure 49 System boundery model

Nr.	Name	Description	Actors
1	Setup contract	For each product AJB min-max service maintain for AR it will be produced a contract. The contract gives the interval for probing the inventory level, min-max level and duration.	Market dep., AJB min-max service
2.	Update min-max level	Market department change the min-max level.	Market dep., AJB min-max service
3.	View accounting system	Interface to monitor documents which is stored in Accounting system	Market, Order, warehouse dep., Accounting system
4.	Sign up	AR need to register as customer with min-max service	Market dep.
5.	Receive	External notification. As soon as light	Order dep.,

	information about delivery	producer has found a transport company for the delivery. This company notifies AR order dep.	transport
6.	Notify warehouse of delivery	Internal notification, Warehouse is informed about the ordered products.	Order, warehouse dep.
7.	Update accounting system	All order information such as freight letter and notifications is stored.	Accounting system, order dep.
8.	Notify warehouse of pending order	Internal notification, Warehouse is informed about the order.	Order, warehouse dep.
9.	Receive information on order	External notification. When min-max service have created an order, an order is sent to the accounting system	Order dep., accounting system
10.	Receive freight letter	The driver transfer the freight letter from the PDA to AR order system	Warehouse dep., Transport
11.	Notify order dep. On received order	Internal notification. Order dep. Receives a message that the order is arrived and is registered in the system. The invoice is ready to be paid.	Warehouse, order dep.
12.	Validate freight letter	When the product is received, it has to be controlled	Warehouse dep.
13.	Notify min-max service on delivery status	External notification. AJB min-max service receives a confirmation that the order product is delivered.	
14.	Register light bulb consumption	The inventory level is updated.	Warehouse dep. Iventory db.
15.	View inventory level	Interface in order to monitor the warehouse inventory level.	Warehouse dept.
16.	Give inventory level	Interface for AJB min-max service to get information about current inventory level.	
17.	Update inventory db	Updates inventory db	

12.2.2 Reference architecture analysis

This model groups the use cases into different services. It is totally seven services.

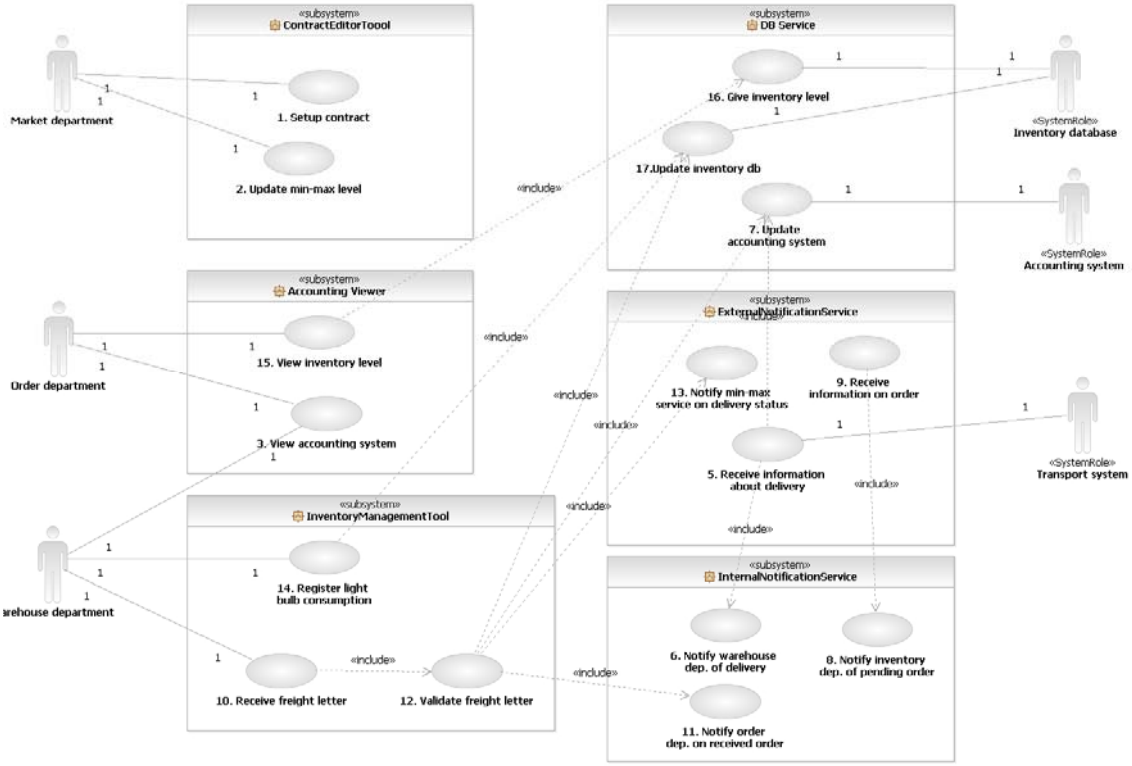

Figure 50 Subsystem grouping

12.3 Service model

The service model describes the system architecture and its components and services. The collaboration between components is modelled with interaction and interface diagrams. The service model describes both services static structure and behaviour. The static structure describes components internal structure and their dependencies. This is a platform independent model.

12.3.1 Component structure model

The component structure model describes components on a higher abstraction level and its dependencies.

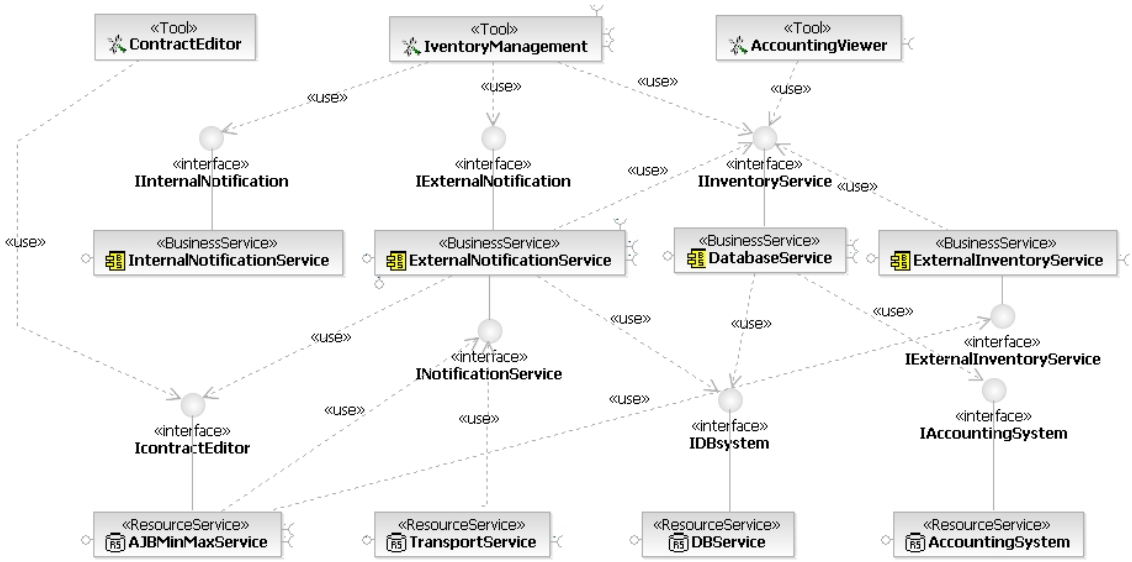

Figure 51 Component diagram

ContractEditorTool
ContractEditorTool is a tool component and contain all use cases which is handled by the market department. "Setup contract", Register as customer" and "update min-max level" will be a web client or a rich client who require access to internet and the AJB-min-max service. This tool is created for the market department.

AccountingViewer
AccountingViewer contain use cases for the order and warehouse department.

InternalNotificationService
InternalNotificationService gives internal notifications.

ExternalNotificationService
ExternalNotificationService is a business component and contain use cases for automatically notify external actors.

ExternalInventoryService
ExternalInventoryService implements the interface to the AJB-min-max service. This provide access to AR's warehouse.

InventoryManagementTool
InventoryManagementTool is a tool component which is used by the warehouse department.

DatabaseService
DatabaseService is a business service component which is used by the warehouse department and order department. This component abstracts the database layer.

12.3.2 InventoryManagement Tool description

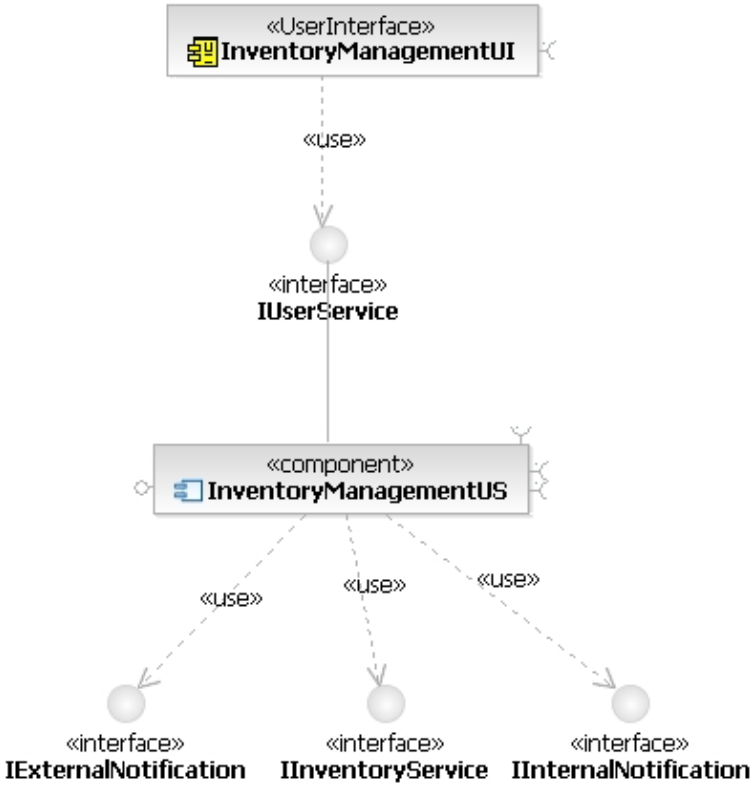

Figure 52 Inventory management tool

Figure 52 describes the internal components within Inventory management tool component. InventoryManagementUI contains user interfaces, while InventoryManagementUS act as a façade which provides the necessary implementation of the use cases. This component is dependent of three external components.

12.3.3 BCE analysis diagram

The inventory management tool is further mapped into classes which implements the necessary elements for this tool.

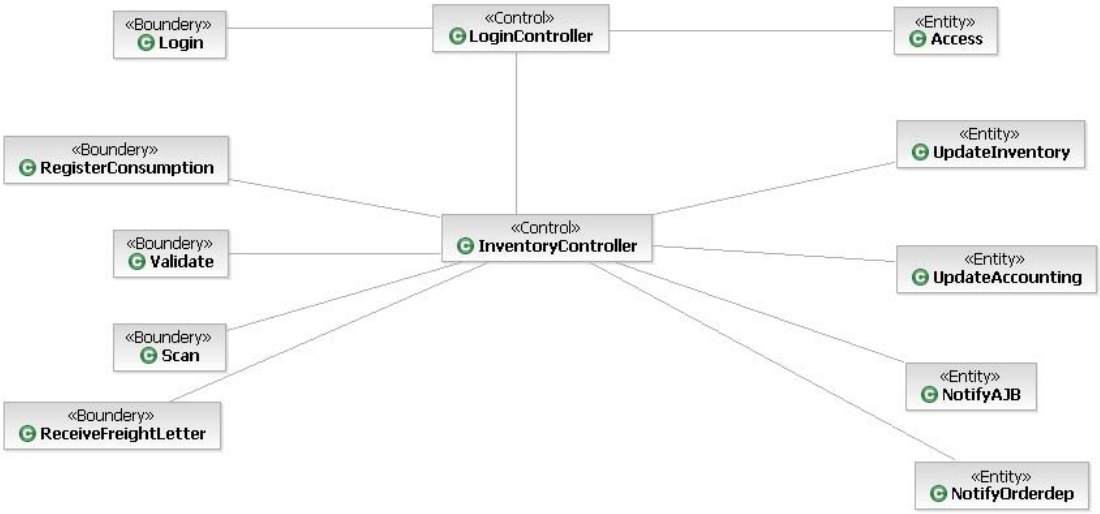

Figure 53 Inventory management tool BCE

12.3.3.1 Interaction diagram

Figure 54 shows the internal interaction within inventory management tool handles the validate order use case.

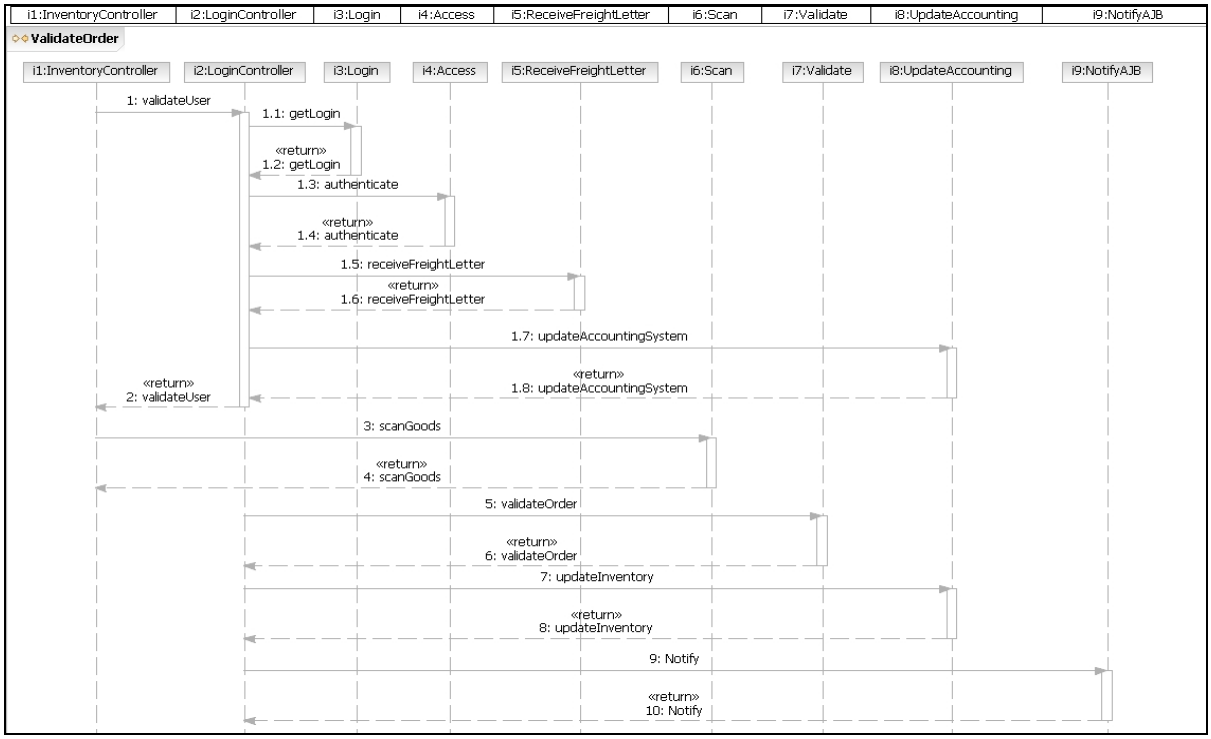

Figure 54 Validate order interaction

13 Appendix Validate order execution

```
STATE=> IDLE, event = deployModel,[guard= EObject != null],
action=setUml2Model
STATE=> MODEL_LOADED, event = execute,[guard= uml2Model != null],
action=doParse()
============== EXECUTE MODEL ========================
START EXECUTING
EVENT:
PUSH:validateUser
EVENT:
EVENT:
PUSH:getLogin
EVENT:
EVENT:
POP:getLogin
EVENT:
EVENT:
PUSH:authenticate
EVENT:
EVENT:
POP:authenticate
EVENT:
PUSH:receiveFreightLetter
EVENT:
EVENT:
POP:receiveFreightLetter
EVENT:
PUSH:updateAccountingSystem
EVENT:
EVENT:
POP:updateAccountingSystem
EVENT:
EVENT:
POP:validateUser
EVENT:
EVENT:
PUSH:scanGoods
EVENT:
EVENT:
POP:scanGoods
EVENT:
EVENT:
PUSH:validateOrder
EVENT:
EVENT:
POP:validateOrder
EVENT:
PUSH:updateInventory
EVENT:
EVENT:
POP:updateInventory
EVENT:
PUSH:Notify
EVENT:
POP:Notify
```