

# A Design Framework for Service-oriented Systems

Raluca Marinescu, Eduard Paul Enoiu

June 28, 2011

Master's Thesis in Computer Science, 30 credits

Supervisor: Aida Čaušević

Examiner: Cristina Seceleanu

MÄLARDALEN UNIVERSITY  
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING  
721 23 VÄSTERÅS  
SWEDEN



## Abstract

In the context of building software systems, Service-oriented Systems (SOS) have become one of the major research topics in the past few years. In SOS, the smallest functional units are services that can be created, invoked, composed, and if no more needed, deleted on-the-fly. Since these software systems are composed out of different services there is no easy and straightforward way to assure the Quality of Service (QoS). The formal specification of both functional and extra-functional system behaviour, compatibility, and interoperability between different services have become important issues. As a way to address these issues, resource-aware timing behavioural language REMES was chosen to be extended towards service-oriented paradigm with a service specific information, such as type, capacity, time-to-serve, etc., as well as Boolean predicate constraints on control flow guarantees. In this thesis we present a design framework that provides a graphical user interface for behaviour modelling of services based on REMES language. NetBeans Visual Library API is used to display editable service diagrams with support for graph-oriented models. A textual dynamic service composition language was implemented, together with means to automatically verify the service composition correctness. We ensure also an automated traceability between the service specification interfaces, where both modelling levels are combined in an efficient tool for designing SOS.

### **Acknowledgements**

This thesis could not have been done without the great support of our examiner Cristina Seceleanu and our supervisor Aida Čaušević. Thank you for your patience, knowledge, experience, and willingness to help us.

We are very thankful to Lars, Niklas, Ukrit, Li, Andrea, Ivan, Umesh, Gowri, and many more for the great time we have spent together during these two years.

Last but not least, we would like to thank our parents for their enduring support.

Västerås, June 2011



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose . . . . .	2
1.2	Outline . . . . .	2
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Behavioural Modelling of Services . . . . .	3
2.1.1	Behavioural Modelling of CBS . . . . .	3
2.1.2	Behavioural Modelling in REMES . . . . .	4
2.1.3	REMES extension for SOS . . . . .	5
2.2	Graphical Libraries and Frameworks . . . . .	9
2.2.1	Motivation . . . . .	9
2.2.2	Comparison of Graphical Libraries and Frameworks . . . . .	9
2.3	Visual Library API . . . . .	11
2.3.1	Visual Library API Architecture . . . . .	11
2.3.2	Widget Class . . . . .	11
2.3.3	ConnectionWidget Class . . . . .	14
2.3.4	LayerWidget Class . . . . .	15
2.3.5	The Scene: The Root Element . . . . .	15
2.3.6	Model-View and GraphPinScene . . . . .	15
<b>3</b>	<b>Implementation</b>	<b>17</b>
3.1	Tool Architecture . . . . .	17
3.2	Scene hierarchy of elements . . . . .	18
3.3	Models for REMES Service Diagram Manipulation . . . . .	19
3.3.1	Data Model Structure . . . . .	20
3.4	DEV and Display Model . . . . .	22
3.4.1	An Atomic Service Widget . . . . .	22
3.4.2	Parallel Connector Widget . . . . .	24
3.4.3	Composite Service Widget . . . . .	26
3.4.4	AND/OR Mode . . . . .	26
3.4.5	List Widget . . . . .	27

---

3.5	Composite Mode Scene . . . . .	28
3.6	AND/OR mode scene . . . . .	29
3.7	Console View . . . . .	29
3.7.1	Parsing with DOM . . . . .	30
3.7.2	Generating the hierarchical language from the data model . . . . .	32
3.7.3	Generating the data model out of hierarchical language . . . . .	33
<b>4</b>	<b>User Interface</b>	<b>35</b>
4.1	GUI Overview . . . . .	35
4.1.1	Diagram Editor View . . . . .	35
4.1.2	Console View . . . . .	37
4.2	Diagram . . . . .	38
4.2.1	Atomic service . . . . .	38
4.2.2	Composite service . . . . .	38
4.2.3	Sequential Connection . . . . .	39
4.2.4	Parallel Connection . . . . .	39
4.2.5	AND/OR Mode . . . . .	40
4.2.6	Service List . . . . .	42
4.3	Data Model and HDCL . . . . .	42
<b>5</b>	<b>An Illustrative Example</b>	<b>45</b>
5.1	ATM Description . . . . .	45
5.2	Service Repository . . . . .	46
5.3	A service composition . . . . .	48
<b>6</b>	<b>Further Ideas</b>	<b>51</b>
<b>7</b>	<b>Conclusions</b>	<b>53</b>
	<b>References</b>	<b>55</b>

# List of Figures

2.1	A REMES mode . . . . .	5
2.2	A service modelled in REMES . . . . .	7
2.3	NetBeans Visual Library Architecture as a Class Diagram . . . . .	12
2.4	Simplified widget hierarchy . . . . .	13
3.1	The design framework workflow . . . . .	18
3.2	Display and Data Models Lifecycle . . . . .	20
3.3	Data file Structure for Edge, Scene Control, and Pin element . . . . .	21
3.4	Data file Structure for Node element . . . . .	21
3.5	ConnectScene Structure as a Class and datatype diagram . . . . .	23
3.6	Composition of the Connection Layer in the ConnectScene . . . . .	24
3.7	Construction of an atomic service widget . . . . .	24
3.8	Attributes widget composition inside an atomic service widget . . . . .	25
3.9	Mode properties widget composition inside an atomic service widget . . . . .	25
3.10	Entry and exit widgets composition inside an atomic service widget . . . . .	26
3.11	Parallel widget composition . . . . .	26
3.12	Construction of a composite service as a widget . . . . .	27
3.13	Construction of an AND/OR mode as a widget . . . . .	27
3.14	Construction of a list as a widget . . . . .	27
3.15	Construction of a SubModesScene for composite service widget . . . . .	28
3.16	Composition of entry, init, write, and exit widgets inside the composite service widget . . . . .	29
3.17	SubModesScene structure as a class diagram . . . . .	30
3.18	Composition of ConnectionLayer in a SubModesScene . . . . .	31
3.19	Construction of the AND(OR)ModeScene for AND/OR mode. . . . .	31
3.20	Composition of entry, init, write, and exit widgets inside the AND/OR mode widget . . . . .	31
3.21	XML Data file as a class diagram . . . . .	32
4.1	Diagram Editor View . . . . .	36
4.2	Additional Menu . . . . .	36



4.3	Console View . . . . .	38
4.4	Atomic Service . . . . .	39
4.5	Composite Service . . . . .	40
4.6	Sequential connection . . . . .	41
4.7	Parallel Connection . . . . .	41
4.8	AND Mode . . . . .	42
4.9	OR Mode . . . . .	43
4.10	Service List . . . . .	43
4.11	HDCL manipulation inside Console View . . . . .	44
5.1	Service oriented ATM system . . . . .	46
5.2	Login service modelled as a REMES composite service . . . . .	47
5.3	Balance service modelled as a REMES atomic service . . . . .	47
5.4	Money withdraw service modelled as a REMES composite service . . . . .	48
5.5	Error service modelled as a REMES atomic service . . . . .	48
5.6	Logout service modelled as a REMES atomic service . . . . .	49
5.7	HDCL and the generated check command for the ATM system . . . . .	49
5.8	A service composition for the ATM system modelled in the design framework . . . . .	50

# List of Tables

2.1	REMES interface operations. . . . .	8
2.2	Libraries compared . . . . .	10
2.3	Visual Library API abstract packages . . . . .	11
2.4	Visual Library API widgets subclasses . . . . .	12
2.5	Widgets Actions Types . . . . .	14
3.1	Container (Scene) representation definition . . . . .	19
3.2	Displayed element's properties . . . . .	19



# Chapter 1

## Introduction

Over the last years, software systems have increased in complexity, which emerged from a need to integrate and connect different applications and resources into new system scenarios, requiring new paradigms to address these new research challenges.

Service-orientation can be seen as a new paradigm that provides the basis to deal with application and resource integration and composition by exploiting a loosely coupled and autonomous software entities called *services*. In literature we can find many definitions for the term "service" with different meanings depending on the field of research [16] [24]. Broy et al. define a *service* as a "set of functions provided by a software or system, usually accessible through an application programming interface" [8].

In the context of system design, Service-oriented Systems (SOS) provide separation of service interface from the service behavioural description. Service interface information such as capacity, time-to-serve, type, etc., are available and visible to potential service users. This information is used by users to find and invoke on-the-fly services, most suitable for their needs. On the other hand, service implementation is hidden from users, but available to service developers. Service implementation provides information about a service functionality, enabled actions, resources used, and interactions with other services [11]. Based on the behavioural description, the developer can improve a service with respect to required Quality of Service (QoS) or functionality and ensure that the correctness of the service behaviour remains unchanged. The service behavioural description enables the developer to understand the service inner functionality, to interconnect services in a correct manner, and provide a better way to analyse service extra-functional properties, such as availability, accessibility, integrity, performance, reliability, etc. To ensure a design-time analysis of the service behaviour description, a formal model of a service should be provided. In many cases, the systems lack their formal behaviour description. The QoS reflects not only functional, but also extra-functional properties of the service/system. The concept of internal service description is similar in characteristics (e.g. reusability, composition) with previously existing paradigms (object-oriented and component-based), more concretely to their smallest functional units, objects and components. Underlying concepts for both Component-based Systems (CBS) and SOS are modularization (i.e., modules are made from splitting up the system functionality) and composition (i.e., modules are used in order to get the overall system behaviour). In CBS all components and connections between them are decided at design-time, before the actual system is provided to users.

Based on the fact that SOS and CBS are similar to each other in many ways, it could be beneficial to use a unified behavioural model for both paradigms [11]. Seceleanu et al.

have introduced a behavioural modelling language, called REMES, which is a state-machine language for analysis of resource-constrained behaviour of a system [28]. Čaušević et al. have provided an extension of the REMES language [10] towards SOS. The extension brings service-oriented features, such as service interface description (type, capacity, time-to-serve, status, pre-, and postcondition), Hierarchical Language for Dynamic Service Composition (HDCL), as well as means to check service compositions, making REMES suitable to behavioural modelling and analysis of SOS. The benefit of having pre-, and postconditions lies in the fact that the constraints imposed during the system development, can be easily proven by checking the Boolean expressions for a given service setup. Also, by using service operations one can create, add, delete, or replace a service, but also connect services sequentially, or in parallel using HDCL. The requirements can be proven by checking the correctness of service compositions (Boolean relation between invoked services are being checked).

In the design framework developed within this thesis work, we have considered behavioural description provided in the language REMES. The constructs provided by REMES have enabled us to build a graphical environment for behavioural modelling of services and support for easier manipulation and reasoning about system requirement specification using formal techniques. Our framework comprises specialized cross-platform standalone Java tools and components to enhance the potential of SOS by focusing on intuitive service manipulation. The dynamic and scalable aspects of service composition are integrated as a textual description language, providing the user with an easy mechanism to address on-the-fly manipulation of services.

## 1.1 Purpose

The purpose of this thesis is to provide a service-oriented design framework, by developing a Graphical User Interface (GUI) for a service behaviour modelling based on the resource-aware timing behavioural language, called REMES. The thesis also contributes with an automated way of ensuring traceability between visual models and the textual description language, which intends to facilitate efficient design.

## 1.2 Outline

The thesis is divided in six chapters. In Chapter 2, we provide the background for the basic concepts of REMES language, its extension towards SOS and the description of the library used for displaying visual models. Chapter 3 describes relevant key concepts of implementation such as architecture, internal structure of the tool, composition of the internal visualization model. Chapter 4 comprises a detailed description of the GUI. In Chapter 5, we describe some ideas for possible further extensions of the provided framework. Chapter 6 discusses the achieved results and concludes the thesis.

## Chapter 2

# Preliminaries

In this chapter, we provide some preliminary explanations of a service behavioural modelling, tools and concepts used to develop this framework. Also, in order to choose the most appropriate underlying graphical management system, we have evaluated some popular available technologies and chose the one suitable for our purposes.

### 2.1 Behavioural Modelling of Services

When designing software systems one has to respect functional and non-functional requirements. They should be the main driver and impact on the software architecture under construction. There exist a number of approaches used to describe SOS architectures [3, 4]. Many of these approaches end up at the level of individual services without providing the detailed internal behavioural description of a service.

In SOS exists a clear separation between a service interface- and behaviour-level description. The aim of the interface information is to make a service visible and available to service users. It specifies relevant informations, such as capacity, time-to-serve, pre-, and postconditions, etc. On the other hand, service behavioural description is hidden from the service user, but available to service developers. One may say that this approach is beneficial in terms of efficiency when a service is changed to fit with newly given user requirements. Also, developers can reuse existing services, significantly reducing time to develop new systems.

SOS is a new paradigm that evolves from the object-oriented computing and CBS paradigms, which makes it appropriate to try to adapt and extend existing approaches to fit with concepts of SOS. Also, as pointed out in [11] there are practical means for relating SOS and CBS in the context of behavioural modelling with support for formal analysis. Čaušević et al. [10] have proposed an extension of the existing behavioural modelling language, called REMES. Since the work in this thesis is based on the REMES language, that was initially introduced for CBS, in the following we first describe the REMES language in context of CBS.

#### 2.1.1 Behavioural Modelling of CBS

The basic units of CBS are component models. Component models are used in the development of component to describe their interfaces, dependencies, and composition mechanisms. CBS aims at supporting component reusability and efficient software development. In most CBS, the design description usually ends up at the detailed description of behaviour in

terms of interfaces, connectors, and protocols [30]. Also, in CBS components and connections between them are composed in advance, before the actual system is provided to users.

An approach that treats specification and modelling in component-based fashion is ProCom (A Component Model for Distributed Embedded Systems) [29]. The approach defines models as a group of hierarchically interconnected components. The components have well defined interfaces that provide a separation between data and control flow. To enable a formal behavioural modelling and analysis of systems designed in ProCom component model, Seceleanu et al. [28] have introduced a Resource Model for Embedded Systems (REMES) that provides a model and a state-machine based language for analysing resource-constrained behaviour of a system. In addition, REMES is used for specification of both functional and extra-functional behaviour of the components (timing, resource usage, reliability etc.). REMES models can be transformed into the formal Timed Automata (TA) [27] or Priced Timed Automata (PTA) [6], and analysed with UPPAAL<sup>1</sup> or UPPAAL CORA<sup>2</sup> tools. In the next section we provide more details about REMES.

### 2.1.2 Behavioural Modelling in REMES

The purpose of REMES behavioural language is to provide a meaningful foundation for timing and resource analysis within embedded systems. The language provides support for different types of resources, discrete such as memory, or continuous such as energy. Initially the language was introduced for the specification, modelling, and analysis of CBS.

The component behaviour is described by a REMES mode. A mode can be *atomic* (does not contain any sub-mode(s), see *Mode a* and *Mode b* in Figure 2.1) or *composite* (it contains an arbitrary number of sub-mode(s)). For transferring data between modes, REMES uses a data interface that consists of global and local variables of type Boolean, natural, integer, or array. The timing properties are expressed via a clock variable, a special kind of variable that evolves at rate 1 (variables  $k$  and  $v$  in Figure 2.1).

The control flow in a mode is defined by a set of directed lines called edges that connect modes at control points. The execution of a composite mode is performed by using a sequence of discrete steps. An action,  $A = (g, S)$  (e.g.,  $(v == x, y := z)$ , in the Figure 2.1), is a statement  $S$  (in this case  $y := z$ ), preceded by a Boolean condition  $g$  (in this case  $v == x$ ), which must hold in order for the action to be executed and the outgoing edge taken. If the mode is decorated with *invariants* (e.g.,  $v < x$  on *Mode b* in Figure 2.1), then one can say that its execution is bounded from above. In order for the mode to be executed *invariant* must hold. When it stops holding the mode is immediately exited through the exit point.

A mode can be defined as *urgent* (decorated with the letter  $U$  in Figure 2.1), meaning that no delay is allowed within a mode. Also it is possible to define a *conditional connector* (depicted as a circle decorated with the letter  $C$  in Figure 2.1) that enables the selection of one edge out of two or more, based on a Boolean condition that guards the action. An action can be taken only if the guard evaluates to true. In Figure 2.1 one of the statement actions,  $k > c$  and  $y == p$  or  $y < c$ , can be chosen for execution.

One of the main features of REMES is the possibility to specify, model and reason about resource-wise continuous behaviour of systems. Every mode can be annotated with corresponding resources. The *consumption* of a resource represents the resource usage that occurs instantaneous in time, whereas the *utilization* of a resource is the rate of *consumption*

<sup>1</sup>The UPPAAL tool is available at <http://uppaal.com/>

<sup>2</sup>For more informations on UPPAAL CORA tool, visit <http://www.cs.aau.dk/~behrmann/cora/>

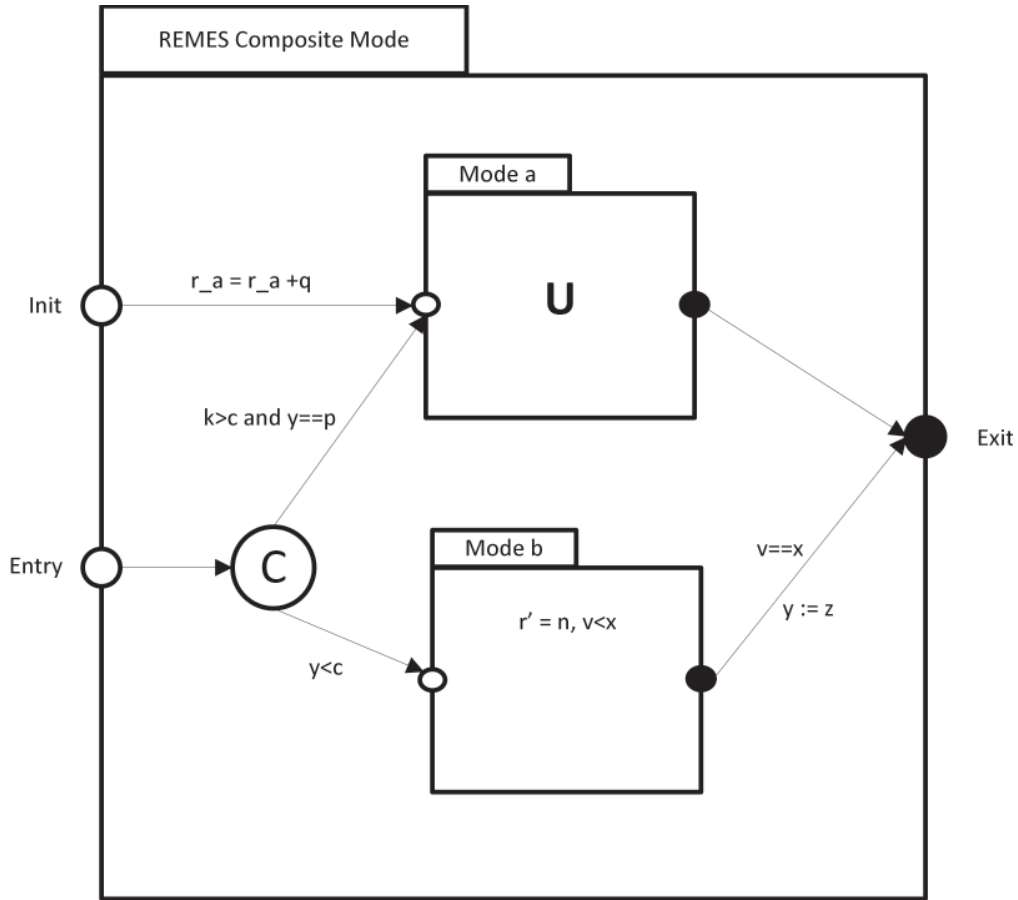


Figure 2.1: A REMES mode

over time. Consumption and utilization can be of monotonic or non-monotonic nature. Consumption can be considered as discrete (resource allocation through updates, e.g.,  $r_a = r_a + q$ ) or continuous (modelled by the first derivative of the real-valued resource variables, e.g.,  $r$  in Figure 2.1). Resources can be classified on whether they are referable or non-referable. Based on this definition, Seceleanu et al. [28] give a straightforward classification of resources. For example the energy resource is continuous and non-referable because it cannot be dynamically allocated and manipulated at run-time. For more details on this classification and on REMES language we refer the reader to [28].

### 2.1.3 REMES extension for SOS

Due to many similar concepts that SOS and CBS rely on, it might be useful to use a unified behavioural model for both paradigms [11]. So, rather than focusing the attention on how to develop a new service-oriented modelling environment, there are solid means to extend an already developed behavioural model for CBS. REMES have been identified as a suitable language for describing, modelling, and analysing SOS, for two main reasons: i) it has precise semantics, ii) it is used for describing functional and non-functional behaviour of components



[10]. By extending REMES, the aim is to provide a service behaviour description needed for proper understanding of SOS. Unlike CBS that are composed at design-time, in SOS services are published, invoked, composed, and destroyed on-the-fly. They are more loosely coupled and more independent of implementation attributes than components are. This means that the service developer can create new services out of existing ones, on demand, and that the newly created services should be able to comply to a desired QoS. The price to pay for all the mentioned benefits, is a list of challenges related to QoS prediction, lack of methods and tools to formally check the correctness of a service composition. It is obvious, that the paradigm of SOS still remains to be fully explored, developed, and utilized.

A service can be represented as an atomic or composite mode in REMES (see Figure 2.2). For a service to be published and discovered, a list of attributes are exposed at the interface of a REMES mode as depicted in Figure 2.2. These attributes are defined as follows [10]:

- **service type**: defines if a service is a web service, database service or network service;
- **service capacity**: quantifies the capability of a service to process messages per time unit, defined as a natural number ( $\in \mathbb{N}$ );
- **time to serve**: defines the worst-case time needed for a service to respond and serve a given request, defined as a natural number ( $\in \mathbb{N}$ );
- **service status**: describes the current service status, passive, idle or active;
- **service precondition**: it is a predicate<sup>3</sup> ( $Pre : \Sigma \rightarrow Bool$ ,  $Pre \equiv (PreInit \vee PreEntry)$ ) that conditions the service start and must hold in order for a service to be executed, described as a Boolean;
- **service postcondition**: is a predicate ( $Post$ ) that must hold at the end a service execution, described as a Boolean.

By using these attributes, services can be discovered and retrieved based on the requirements of the system.

It is a case that services have to synchronize their executions. To provide modelling of synchronized behaviour a special type of REMES mode, called AND/OR mode is introduced into the language [10]. These modes finish their execution at the same time from a service user perspective, but from an internal behaviour perspective, AND mode assumes that sub-services are entered at the same time, while the execution of sub-services in an OR mode is conditioned with a guard on their entry points. Also, AND/OR modes can be used with either "and" synchronization (containing services are finalizing their execution at the same time), or "max" synchronization (AND/OR mode execution come to an end when the slowest service terminate its execution). For more details about AND/OR modes we refer the reader to [10].

Hierarchical Dynamic Composition Language (HDCL) was introduced to support service composition [10]. In order to support run-time manipulation there is a need for defining a set of operations. As depicted in Table 2.1, one can create and delete a service. The symbol  $\Sigma$  represents the set of service states. Also, services can be added, removed, or replaced in a service list [10]. An example of a service list is given as follows:

$$service\_list = [service\_name1, \dots, service\_namen] \quad (2.1)$$

<sup>3</sup>in this expression  $\Sigma$  is the polymorphic type of the state that includes both local and global variables, and predicates  $PreInit$ ,  $PreEntry$  are the initial, and the entry precondition of the service, respectively.

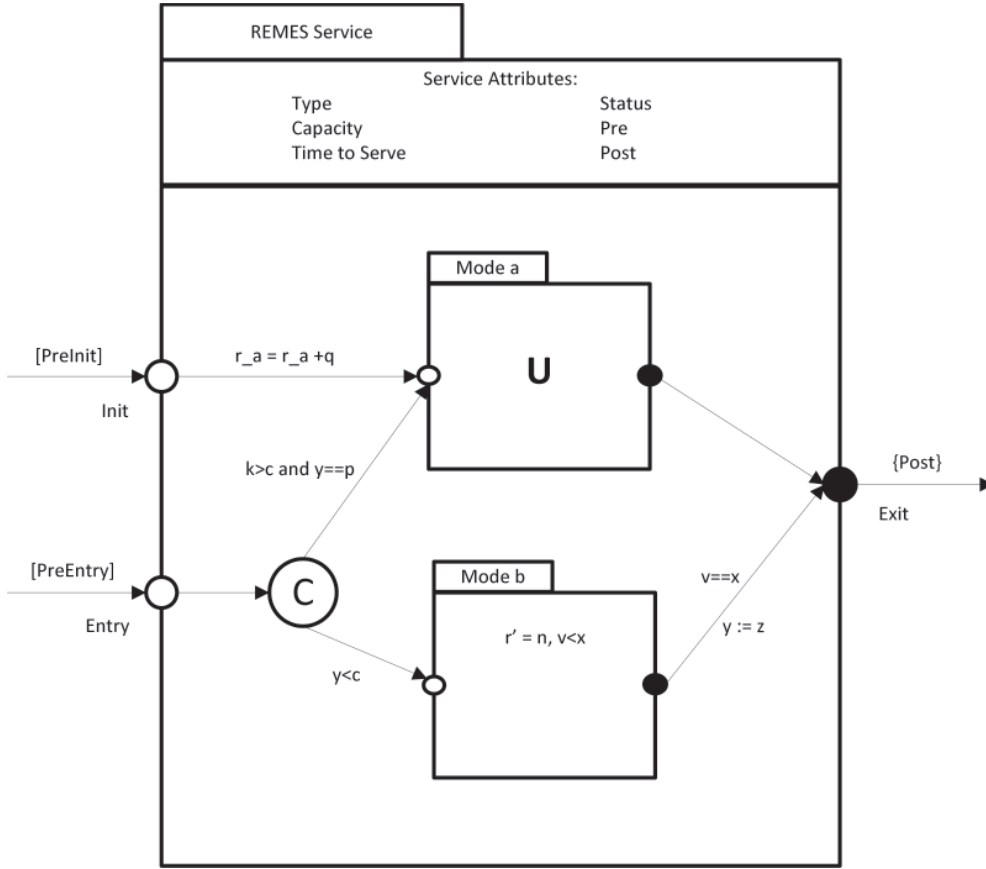


Figure 2.2: A service modelled in REMES

The service list is created empty, while services are added later. Also, the user can access elements by using an index position  $p$ , facilitating the usage of operations like replace service or insert service.

Services can be viewed as independent and distributed units that can be composed to create new services (to fulfil requirements that often evolve continuously and therefore require adaptation of an existing solution). A dynamic service composition, that facilitates the modelling of nested services, is defined in [10] as follows:

$$DCL = (service\_list, PROTOCOL, REQ); \quad (2.2)$$

$$HDCL = (((DCL^+, PROTOCOL, REQ)^+, PROTOCOL, REQ)^+, \dots); \quad (2.3)$$

The formulas 2.2 and 2.3 describe an infinite degree of nesting. The positive closure operator  $+$  is used to express that one or more DCLs are nested to form an HDCL. The PROTOCOL defines the way services are composed (the type of the binding between services), as follows:

$$PROTOCOL ::= unary\_op\ service\_name | service\_name1\ binary\_op\ service\_name2 \quad (2.4)$$

Operation's Syntax	Command
create <i>service_name</i>	create : $Type \times \mathbb{N} \times \mathbb{N} \times "passive"$ $\times (\sum \rightarrow bool) \times (\sum \rightarrow bool)$ $\rightarrow service\_name$
del <i>service_name</i>	$service\_name \rightarrow NULL$
create <i>service_list</i>	create list : $service\_list \rightarrow service\_list,$ $service\_list = List()$
delete <i>service_list</i>	del list : $s\_list \rightarrow NULL$
add <i>service_name</i> to <i>service_list</i>	add : $s\_list \rightarrow s\_list$
del <i>service_name</i> from <i>service_list</i>	del : $service\_list \rightarrow service\_list$
replace <i>service_name1</i> with <i>service_name2</i>	replace : $service\_list \rightarrow service\_list$
insert <i>service_name1</i> in <i>service_list</i>	add : $service\_list \rightarrow service\_list$

Table 2.1: REMES interface operations.

The operators defined in formula 2.4 are:

$$unary\_op = exec\_first \quad (2.5)$$

$$binary\_op = ; (sequential) \mid \parallel (parallel) \mid \parallel SYNC\_and \mid \parallel SYNC\_or. \quad (2.6)$$

The semantics of the unary and binary protocol operators are defined as follows:

- *exec\_first*: specifies witch service should be initially executed in a composition;
- *sequential composition*(;): the services are executed in the specified sequence, one after the other, uninterrupted;
- *parallel composition*( $\parallel$ ): one or more services can be executed;
- *parallel composition with AND synchronization*: services are entered at the same time;
- *parallel composition with OR synchronization*: one or all constituent services are entered;

The requirement REQ, as represented in formulas 2.2 and 2.3, is a predicate that includes both functional and extra-functional properties of the system. It identifies the required attributes constraints, capability, characteristics, or quality of a system such that it exhibits the value and utility requested by the user.

Service refinements are done by weakening or strengthening the service pre- and postcondition in order to check if the result of the correctness check does not satisfy the REQ given by a service user. The strongest postcondition technique that facilitates the forward analysis technique is used to provide the correctness check. In [10] forward analysis techniques are used to provide a correctness check of a REMES service. Assuming  $p$  and  $q$  are predicates that describe the initial and the final state of the service, and  $S$  is a service, the notation  $\{p\} S \{q\}$  means that if  $p$  holds, than  $S$  is guaranteed to finish with the condition  $q$  true. An example of an assignment statement can be given as follows:

$$\{x \geq 1\} x := 10 \{x \geq 9\} \quad (2.7)$$

$$\{x = 0\} x := x + 2 \{x = 2\} \quad (2.8)$$

The strongest post condition  $sp.S.p$  states that if  $p$  holds then the execution of  $S$  results in  $sp.S.p$  being true. This means that  $sp.S.p \Rightarrow q$ . For our example in formulas 2.7, 2.8 the strongest postcondition can be determined as follows:

$$sp.(x := 10).(x \geq 1) \equiv x = 10 \wedge (\exists x \cdot x \geq 1) \quad (2.9)$$

$$sp.(x := x + 2).(x = 0) \equiv (\exists x_0 \cdot x = x_0 + 2 \wedge x_0 = 0) \Rightarrow x = 2 \quad (2.10)$$

where  $x_0$  is the fresh variable storing the initial value of  $x$ .

The correctness proof is reduced to checking the strongest postconditions as determined in 2.10. For a more thorough description of the correctness check for a REMES service refer to [10].

## 2.2 Graphical Libraries and Frameworks

A part of this thesis work is to investigate which standard libraries to employ, and which adaptations are required, in order to create a standalone Java application for modelling REMES services. In the past 20 years, we have witnessed a huge expansion in visualization design, which is the underlying technology of every modelling tool. Some of the popular technologies are: TreeMaps [20], Graph Visualization Framework(GVF) [5], infovis toolkit [15]. Numerous visualization libraries and frameworks exist, either commercial (yFiles [32] and Tom Sawyer Visualization) or open-source( Prefuse [19], NetBeans Visual Library API, JGraph [2], Eclipse Graphical Editing Framework [9])

In the following, we provide an evaluation on several approaches that were chosen as the possible candidates. We describe how the evaluation was realized by stating the candidates and explaining the result of the assessment.

### 2.2.1 Motivation

In a modelling tool, the GUI is designed such that a user can draw diagrams on the screen almost as one would on paper. These drawing structures cannot be managed by a regular interface structures like tables, trees or lists and the capabilities go beyond regular GUI libraries like AWT or Swing [13]<sup>4</sup>. We have decided to evaluate different approaches for this standalone modelling tool before going into the actual development process.

### 2.2.2 Comparison of Graphical Libraries and Frameworks

Although visualization or graphical frameworks technologies have been proven indispensable tools for creating and representing complex structures, it is often difficult to find or to build a software framework for creating dynamic visualizations of both structured and unstructured data. In order to get a satisfactory level of performance, usability, and possibility for further development, the library that is to provide this, should be based on the standard libraries that can operate with other Java GUI technologies like Swing.

We have found four libraries that are the most customizable and easiest to use. We have further investigated and compared each by different criteria in order to determine the best

<sup>4</sup>a fully-featured user interface development kit for Java applications

Criteria	Prefuse	Visual Library	Eclipse GEF	JGraph
API Documentation	No	Yes	Yes	Yes
Model Customizability	No	Yes	Yes	Yes
Release status	Beta	Yes	Yes	Yes
Standalone	Yes	Yes	No	Yes

Table 2.2: Libraries compared

library for our needs. The assessed criteria includes customization, API documentation, samples, stability and license. In the following we provide a short description of each of these four libraries.

Prefuse is a set of software tools for creating a rich set of features for data modelling, visualization, and interaction. It supports components for layout, panning, zooming, interaction and rendering; it also provides data structures for tables, graphs, and trees. It is only available in a beta version and the documentation is not as complete as we might expect.

NetBeans Visual Library is intended for visualization purposes with support for graph-oriented modelling. It is a part of the NetBeans Platform [7], but it is possible to use it as a standalone library by adding the .jar files to the Java application class-path. The library comes with comprehensive documentation and there is a growing community behind it. It is based on the JFC Swing and has a similar programming representation. It supports zoom in/out, collapse/expand, and the possibility to encapsulate Swing components inside the visualization.

Eclipse Graphical Editing Framework(GEF) allows developers to build rich graphical editors from an existing application model. It consists of plug-ins to the Eclipse Platform [14] provided for use only within Eclipse. There is an extensive documentation on how to use and extend GEF.

JGraph is a feature-rich, standard-compliant, and open-source graph component, recommended by the open-source community [2]. It has an extensive documentation [1] on how to use the API. It is a low level library and is oriented more towards design layout. JGraph is a full-size library that provides a broad functionality set. It supports various back ends such as XML, database, edge editing, drag and drop, selection and many other functionalities.

In Table 2.2, we show an overview of each candidate based on the chosen criterion. After some initial investigation, we have eliminated Prefuse due to the lack of documentation and orientation on low level design layout. The major drawback of Eclipse Graphical Editing Framework's (GEF) is its native integration with the Eclipse Platform. In order to use it within a standalone application the designer should develop an underlying modification program that has to be updated with every new version of GEF. JGraph has an extensive manual and a thorough collection of modelling features. However, JGraph is oriented on the design layout and the programmer needs a lot of experience to get a usable application in a short time.

NetBeans Visual Library was chosen since it provides the best overall performance, it is constructed in a Swing-like composition, has superior functionality, and complete documentation. It is also a high level API which makes it less complicated for large applications. Regardless of the fact that the implementation would be possible with any of these libraries, the work in this thesis related to graphical modelling of services is designed with Visual Library API as an underlying base.

Package	Description
org.netbeans.api.visual.action	Built-in widget actions.
org.netbeans.api.visual.anchor	Defining an edge's source and target point.
org.netbeans.api.visual animator	Controlling animations on a scene.
org.netbeans.api.visual.border	Border graphics for a widget.
org.netbeans.api.visual.export	Export a Scene to an image file.
org.netbeans.api.visual.graph	Contains Built-in graph-oriented models.
org.netbeans.api.visual.graph.layout	Built-in graph-oriented layout algorithms.
org.netbeans.api.visual.laf	Style definition for colors and borders.
org.netbeans.api.visual.layout	Widget-layouts.
org.netbeans.api.visual.model	Model-objects with widgets on the scene.
org.netbeans.api.visual.router	Defines a router or a path for an edge.
org.netbeans.api.visual.widget	Widget class.
org.netbeans.api.visual.widget.general	High-level widgets.

Table 2.3: Visual Library API abstract packages

## 2.3 Visual Library API

The NetBeans Visual Library API is a high-level graphical library. It is designed to support applications that need to display editable diagrams or graphs. The library can also be used by applications that are not built upon the NetBeans Platform. The libraries that are needed for running a Visual Library API application are *org-openide-util.jar* and *org-netbeans-api-visual.jar*.

As shown in Table 2.3, the API provides a set of reusable packages. By using them the developer is creating visualization diagrams. Each package contains a set of predefined widgets, models and actions. In the next sections, we explain these reusable components.

### 2.3.1 Visual Library API Architecture

The Visual Library API is comparable in implementation to the Swing library. It provides a large set of reusable pieces and widgets. A Widget is very similar to Swing's *JComponent*. The main class of all graphic components is the widget class. A widget can also be a container for other widgets. Each widget has a location relative to its parent. The widget main class is responsible for representing the border and background.

A widget has a certain layout responsible for the positioning of its childs. They depend upon each other in order to be notified about changes and can also be linked to a series of actions that are executed when specific user events occur.

Figure 2.3 shows a class diagram representing a top level architecture of the Visual Library API. The topmost widget is always a widget called scene. It forms the bridge between Swing and the Visual Library API, by providing a *JComponent* using the *createView()* method that displays the contents of scene. In comparison with Swing components, widgets have a non-fix shape.

### 2.3.2 Widget Class

In Visual Library API the central graphical component is a widget. This class provides information relevant to different functionalities like preferred size, foreground, background,

Class	Description
ComponentWidget	This widget is used for displaying and updating the SWING components.
ConnectionWidget	This widget is used to connect two points.
ImageWidget	This widget is used to represent an image within a scene.
LabelWidget	This widget is used to display a text.
LayerWidget	Transparent Widget.
LevelOfDetailsWidget	Container for multiple widgets. Offers some visibility functionality when a zoom action is used.
Scene	Is a widget used as a super-class for the displayed widgets.
ScrollWidget	Scrollable widget.
SeparatorWidget	This widget is used to display a separator object.
IconNodeWidget	This widget is used for displaying an image and a label in the same element.

Table 2.4: Visual Library API widgets subclasses

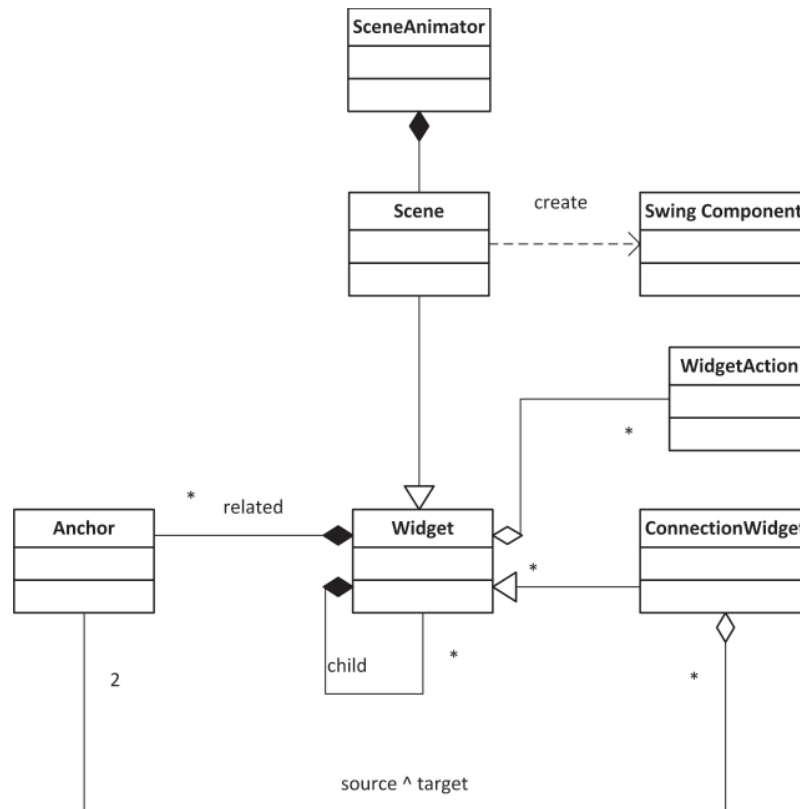


Figure 2.3: NetBeans Visual Library Architecture as a Class Diagram

tooltip, and font. A widget is a graphic super class equivalent to the *JComponent* class in Swing. Many classes are derived from the widget class (see Figure 2.4), each with different

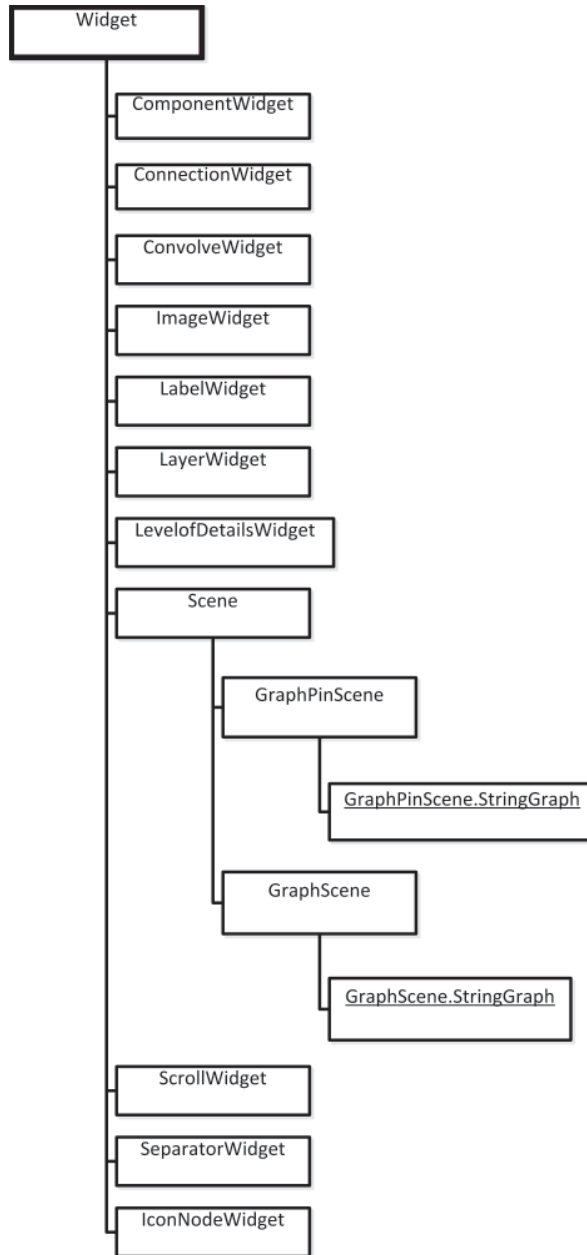


Figure 2.4: Simplified widget hierarchy

visual representation and behaviour. Table 2.4 provides an outline of various predefined widgets types. For more comprehensive descriptions of these classes, see the Visual Library API documentation found within the Javadoc of the Visual Library [26]. Each widget has different interfaces used to provide different graphical functionality. Below, we provide the description for the most important.



Action	Description
AddRemoveControlPointAction	This action can be used by ConnectionWidget widgets. By using it, you add or remove control points by double-clicking them.
MoveAction	By using the MoveAction, a widget can be moved by drag-drop method. The only constraint is that the parent widget has the default layout.
ResizeAction	By using this action, the developer can change the size of widgets on the scene. This action is useful when you want to create expandable widgets or widgets with other widgets inside.
ConnectAction	By using the ConnectAction, the developer can connect two widgets.
HoverAction	With the HoverAction, the developer can use an action when the user is hovering the mouse over a widget.
SwitchCardAction	The action is used for switching between child widgets that are contained in a CardLayout.

Table 2.5: Widgets Actions Types

Every widget can be displayed with a border. The default setting is an empty border. By using the method `setBorder()` another border can be used. The border corresponds to an interface that implements numerous sub-classes like `LineBorder`, `ImageBorder`, `ResizeBorder`, and `SwingBorder`.

Widgets can have a layout defined by the interface called `Layout`. The layout defines how the child widgets are arranged. There are four different layouts available: `AbsoluteLayout`, `FlowLayout`, `CardLayout`, and `OverlayLayout`. With `AbsoluteLayout`, the child widgets are arranged corresponding to the coordinates of the parent widget. By default a widget has this kind of layout. The `FlowLayout` is used for arranging widget in sequential order in vertical or horizontal direction (`createHorizontalFlowLayout()` and `createVerticalFlowlayout()`). It is possible to enable the alignment with a gap between individual widgets. A `CardLayout` determines the currently active widget and makes the other widgets invisible. `OverlayLayout` shows the widget and the child widgets in a minimum area and arranged on top of each other. A widget is defined by static information like content, position or size. When working on a GUI application the developer needs to use in some way the behaviour of a widget.

A widget in Visual Library API can hold a behaviour by means of adding an action that corresponds to a specific event. Many of the predefined actions are automatically performed (moving, selecting or resizing of a widget). Actions are usually created using `ActionFactory` class. They are specified by the `WidgetAction` interface. An event like clicking the left mouse button can trigger a method corresponding to this interface. Table 2.5 illustrates some of the actions used frequently in this work.

### 2.3.3 ConnectionWidget Class

To connect two widgets we can use another class, named `ConnectionWidget`. This class holds a source and a target, that corresponds to two widgets, as anchors. They are used to

represent a path between two specified points.

The visual appearance of the connection can be modified by using different anchor shapes or routers. The router specifies a line path, which is by default a straight line between source and target. An orthogonal router can be also applied instead, and multiple intermediate points can be set or deleted. For more information about routers for connections refer to Visual Library API documentation [25].

### 2.3.4 LayerWidget Class

An important type of widget is the *LayerWidget*. Similar to *JGlassPane* from Swing, it is a transparent layer that can be used to manoeuvre some temporary layers that are being changed very often in a scene.

### 2.3.5 The Scene: The Root Element

All widgets, when created are children of a scene. A scene is an extension of a widget that contains functions for rendering and repainting all objects on the screen. When a widget is updated these functions are automatically called to repaint the scene. One can think of the scene class as the root node of a tree hierarchy of widgets. The scene widget has a number of methods useful for GUI integration, one of them is *createView()* which provides a *JComponent* view that can be embedded into any Swing components.

The Scene class is extended by various classes like *ObjectScene*, *GraphScene*, and the most relevant for our work, *GraphPinScene*.

### 2.3.6 Model-View and GraphPinScene

Widgets in Visual Library API are not generically associated with a data model. That is, a widget owns only information about the graphical flow of data. In order to extend the scene, classes like *GraphScene* or *GraphPinScene* are used. Building a data model on top of a scene and making available some specific methods for assignment to a data model, are important features for widget management.

*GraphScene* and *GraphPinScene* are two abstract classes, suitable for graph-oriented modelling. The only task is the creation and management of data models and widgets. Creating the widget is achieved by using the relevant abstract methods within the subclasses. *GraphPinScene* is using a model with nodes, pins and edges. One or more pins can be attached to a node and edges can connect a source pin with a target pin. Also, there is a binding between the widget model and the graph objects by using a unique id and entity, attributes, and relations.

The *attachNodeWidget()* method is used to create nodes. Similar to creation of edges the method *attachEdgeWidget()* is used. The *ConnectionWidget* class is used as a parent widget for every connection when drawing edge paths. Also, it enables a router to determine the correct path for an edge, so that no intersections occur. To simplify the developer's creation of routers, Visual Library API comes with a built-in class named *RouterFactory*. Pins are created with the *attachPinWidget()* method. As explained in this section, a pin can be an input or output of a node to which an edge can be connected. One or more pins are assigned to a node. The developer can use the *findWidget()* method for support in determining what node is the parent widget in which the created pin is added. To specify a start and an end point for an edge, Visual Library API is using two methods: *attachEdgeSourceAnchor()* and *attachEdgeTargetAnchor()*.



## Chapter 3

# Implementation

In this chapter, we describe the implementation of the Java GUI application for service behaviour modelling based on REMES resource model. The GUI is based on NetBeans Visual Library API to display the models and Swing Library<sup>1</sup> as the user interface toolkit.

The tool is implemented as a client frontend for behavioural modelling of services. The model implemented is based on the REMES language and REMES extension for SOS, both introduced in Chapter 2.

The tool is developed to facilitate concepts of behavioural modelling of both components and services, introduced in REMES. The client frontend is responsible for graphical specification of services and their attributes, and composed them at runtime. It provides graphical environment to specify, model and compose REMES services.

### 3.1 Tool Architecture

In this section, the GUI architecture is explained. Also, a description of the traceability between the REMES model and a hierarchical language for service composition is provided. The application is split into several functional units. Here is a list of all top-level units:

- **Diagram Editor View (DEV)**: Top level unit in charge for opening a new diagram editor. It is responsible for displaying the diagram. It uses the NetBeans Visual Library API to render the scene. It contains the display Model used internally by the tool for the graphically enriched graph.
- **Console View (COV)**: Supports diagram for HDCL transformation, where services and their correctness conditions can be specified.
- **Data Model**: Is used for diagram storing purposes. It saves the data from DEV to a XML<sup>2</sup> document file.

Figure 3.1 illustrates the design flow implemented in our tool. The designer uses: (i) REMES service *editor* for building REMES models, (ii) DEV for composing services in a graphical environment, and (iii) COV for invoking services using HDCL. One can synchronize DEV and COV in order to refine the system on-the-fly and can check whether the given

---

<sup>1</sup>Is a rich set of components for building GUIs and adding interactivity to Java applications. Swing is part of the Java Foundation Classes (JFC).

<sup>2</sup>XML is a standard way of encoding text and data so that content can be processed and exchanged across diverse hardware, operating systems, and applications[17].

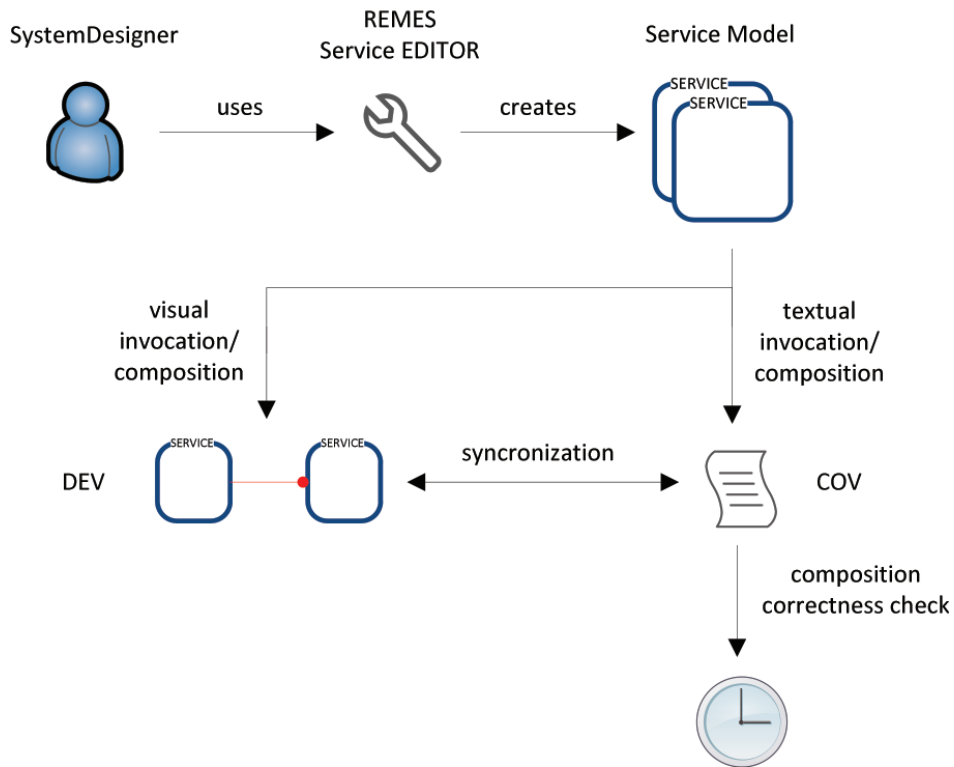


Figure 3.1: The design framework workflow

requirement is satisfied, by showing the strongest postcondition of a given composition w.r.t. a given precondition.

The data model and display model are used for representing and manipulating the diagram. Both models are based on the REMES service behaviour model. The main container for the editor is the diagram that contains services, AND modes, OR modes, parallel connectors, and service list constructors.

### 3.2 Scene hierarchy of elements

Table 3.1 describes some requirements based on the REMES model [28] and service extension specifications [10]. A container is defined as the root element of the hierarchy of displayed elements. The main container is represented by a REMES service diagram. The diagram contains six special child elements: atomic and composite service, AND/OR mode, parallel connector, and list constructor.

The services are connected via edges on control points. Let us assume that two services (be it atomic or composite) are invoked at some point in time. Their instances can be composed as follows:

- **Sequential composition** - services are connected with an oriented edge, meaning that are executed in a sequence, uninterrupted;

Container(Scene)	Child
REMES service diagram	REMES composite service REMES atomic service AND mode OR mode Parallel connector List constructor
REMES composite service	REMES atomic service Conditional connector
AND/OR mode	REMES atomic service

Table 3.1: Container (Scene) representation definition

Elements	Properties
REMES composite service	Name, control points ( init point, entry point, exit point, write point), service attributes, constants, variables, and resources
REMES atomic service	Name, control points (entry point, exit point), service attributes, constants, variables, and resources
AND/OR mode	Name, control points (Entry point, exit point)
Parallel connector	Control points (entry point, exit point)
List constructor	Name, requirement
Conditional connector	Control points (entry point, exit point)

Table 3.2: Displayed element's properties

- **Parallel composition** - services are composed in a connection using parallel connectors;
- **AND/OR synchronization** - services are added inside a AND/OR mode for synchronization;

The composite service can be seen as a container for another hierarchy structure of atomic modes and conditional connectors. Inside it, edges are defined with the following properties: action guard (contains the condition that should be fulfilled in order to execute the edge) and action body (information to be updated by the edge execution). Also, we can consider AND/OR modes as composite modes that contains as sub-modes the services that need to be synchronized.

In Table 3.2 we define the elements and requirements to design the data model. We use this information when defining the constituent components of every scene.

### 3.3 Models for REMES Service Diagram Manipulation

The tool uses two different models for representing and manipulating the diagram: a data model for the transfer of data from DEV to COV and a display model for displaying the diagram. The data model is converted into a display model whenever a diagram is opened in DEV. The advantage of using two different models comes from the practical issue of diagram

representation and storage. For example, the information related to border displaying of a node is undefined in the data model but it's predefined in the display model.

The tool saves every node in the display model to a corresponding node in the data model. Every new node in the display model represents exactly one node in the data model. Figure 3.2 shows a lifecycle diagram of interactions between the two employed models. Generating a data model from the display model is useful for storing the diagram. Therefore, a predefined structure needs to be created in order to represent the data model file. As depicted in Figure 3.2, GraphScene Manager is reading the data model and is building a memory representation of it. When DEV is opened from an existing data model it is converted automatically to a display model representation in order to be shown on screen. DEV uses NetBeans Visual Library API to show the display model. More details about these models are provided in the following sections.

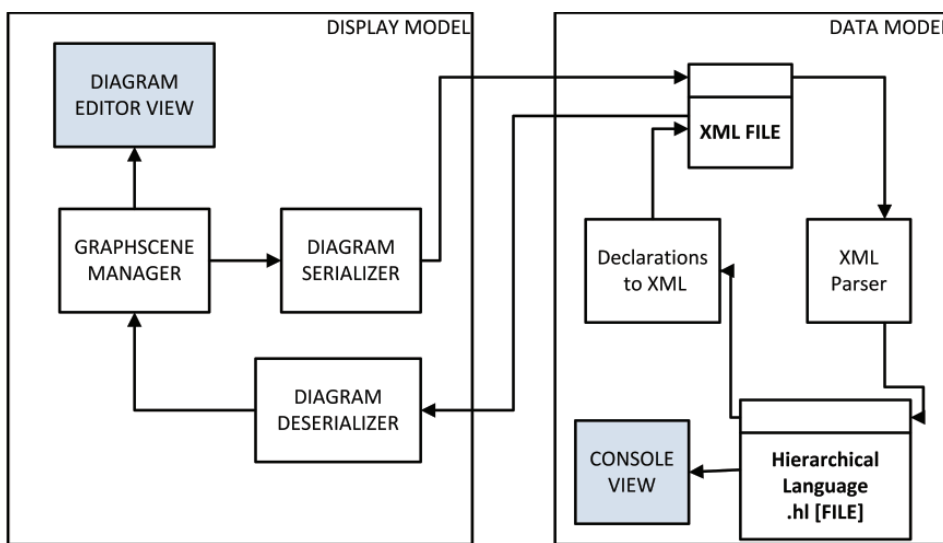


Figure 3.2: Display and Data Models Lifecycle

### 3.3.1 Data Model Structure

In order to store the diagrams we use an XML style document. Although, there are some disadvantages in using XML, such as slow access to data due to parsing and text conversion, it seems a natural choice to use it since it is portable and has a structured description. Figures 3.3, and 3.4 show the hierarchy of elements derived from the requirements defined in Section 3.2.

The top-level element, that contains a Scene child element, is called data document. We create a scene element for every diagram. As depicted in Figure 3.3, a scene element has exactly one scene control child element, which describes the number of nodes, number of edges, and number of pins stored in a diagram. This information is useful when the diagram is reloaded from the saved file.

A scene element can have an arbitrary number of node, edge, and pin child elements. There are six types of nodes included, which are shown in Figure 3.4: Atomic Service, Composite Service, Parallel Connector, AND Mode, OR Mode, and List. All of these types

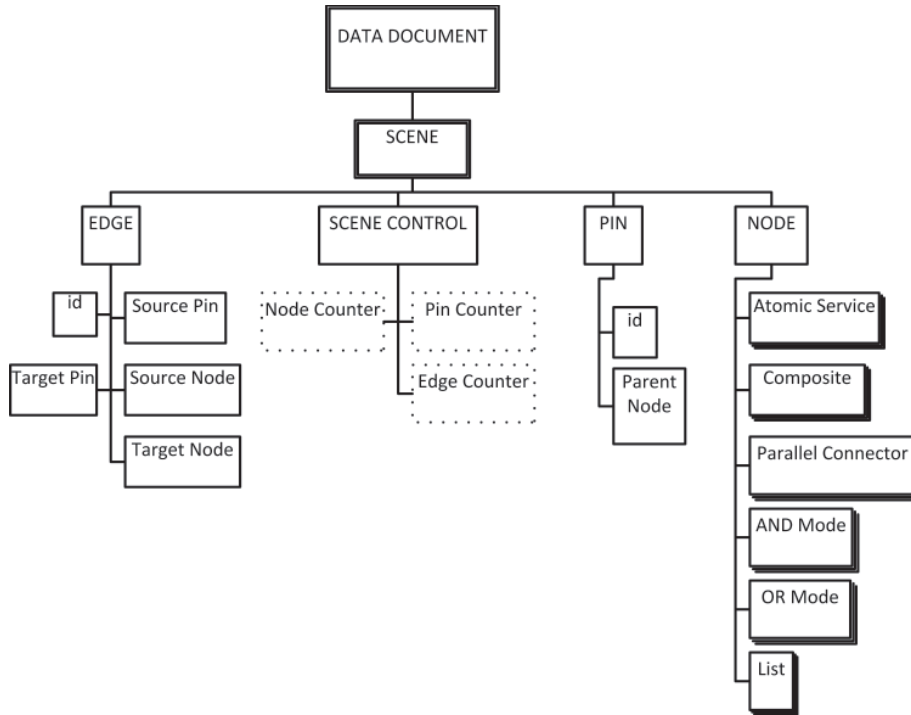


Figure 3.3: Data file Structure for Edge, Scene Control, and Pin element

are in accordance with the requirements described in Section 3.2. Every node has a unique

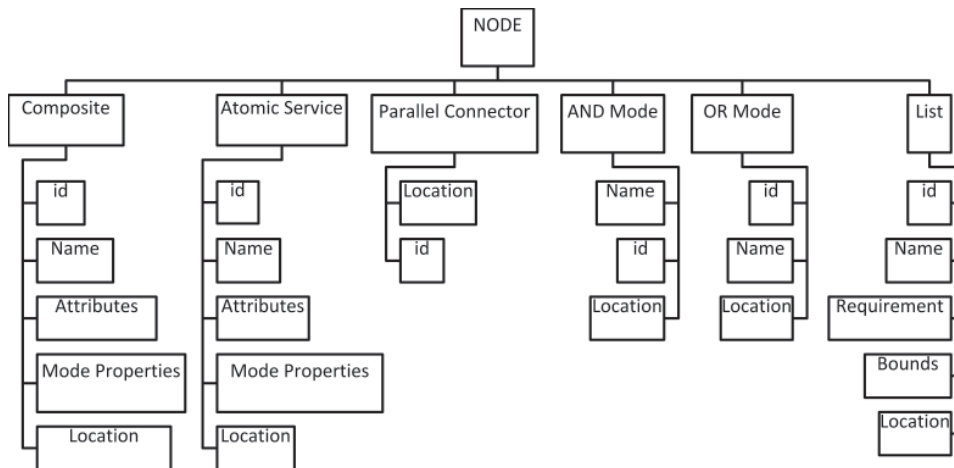


Figure 3.4: Data file Structure for Node element

identifier. Additionally, every type of node have distinctive child elements. For example the Composite Service node has a name, service attributes (pre-, postcondition, time to serve, capacity), mode properties (constants, variables, and resources), and one location element.



The location stores the coordinates of the node in the diagram. A similar mechanism is used for edges; they are referenced via their unique ids. An edge is defined by the identifiers of its source and destination node and the id of the pin at which the edge starts and ends. Pin child elements are identified by a unique id and the id of the parent node where they are created.

## 3.4 DEV and Display Model

The display model is comparable in structure to the data model with the only distinction that it contains information about nodes, pins and edges important for DEV. When a new editor window is opened, the user can create a new blank display diagram or can load an existing data model file. There is no semantic difference concerning the diagram, between the data and display model: every node, edge or pin produces exactly one output in the display model.

DEV is responsible for displaying the diagram. It uses the NetBeans Visual Library API to render the graph. The main class of DEV is named `ConnectScene` and it extends the `GraphPinScene` class. The management of elements is also included in `ConnectScene` class using widgets to hold reference information about the diagram. By having a `GraphPinScene` structure, `ConnectScene` is in charge of managing every diagram element as a node, pin, or edge.

The diagram in Figure 3.5 shows a simplified data structure of classes used in `ConnectScene`. The class `GraphPinScene` is coloured with blue to show that it is a built-in Visual Library API class. `ConnectScene` widget is the root element in our diagram hierarchy. It is responsible for rendering the main diagram area. `ConnectScene` is composed of four layers of type `LayerWidget`: `MainLayer` (all nodes are defined and used within this layer), `ConnectionLayer` (all edges are defined and used within this layer), `InteractionLayer` (it is used when the user wants to connect two nodes by an edge), and `BackgroundLayer`. All layers are transparent and are used to organize different types of widgets.

As depicted in Figure 3.6 for managing edges we provide a separate `LayerWidget` named `connectionLayer`. The creation of edges is dependent on this layer's subclasses and is reliant upon `ConnectScene`. The `attachEdgeWidget()` method is responsible for creating edges between nodes. A `ConnectionWidget` is used to enable the use of a router for selecting a path of edges between two nodes.

For managing the interaction between nodes and edges we use a separate `LayerWidget` named `InteractionLayer`. Connections are added in `ConnectScene` when the user takes a specific action. Implementation of this action class executes the creation of edges between nodes. `MainLayer` is a separate `LayerWidget` where nodes are created in `ConnectScene`. As depicted in Figure 3.4, there are six types of widgets available as nodes: atomic service widget (equivalent to a REMES atomic service that implements a REMES atomic mode), composite service widget (equivalent to a REMES composite service that implements a REMES composite mode), AND composition widget (a REMES AND mode), OR composition widget (a REMES OR mode), parallel widget, and list widget (a List Constructor). All elements have different attributes, properties, and descriptions depending on the requirements defined in Section 3.2

### 3.4.1 An Atomic Service Widget

Figure 3.7 shows a schematic diagram representing an atomic service widget. The outer black border is the border of the widget. Inside the main widget, a `VerticalFlowLayout` is

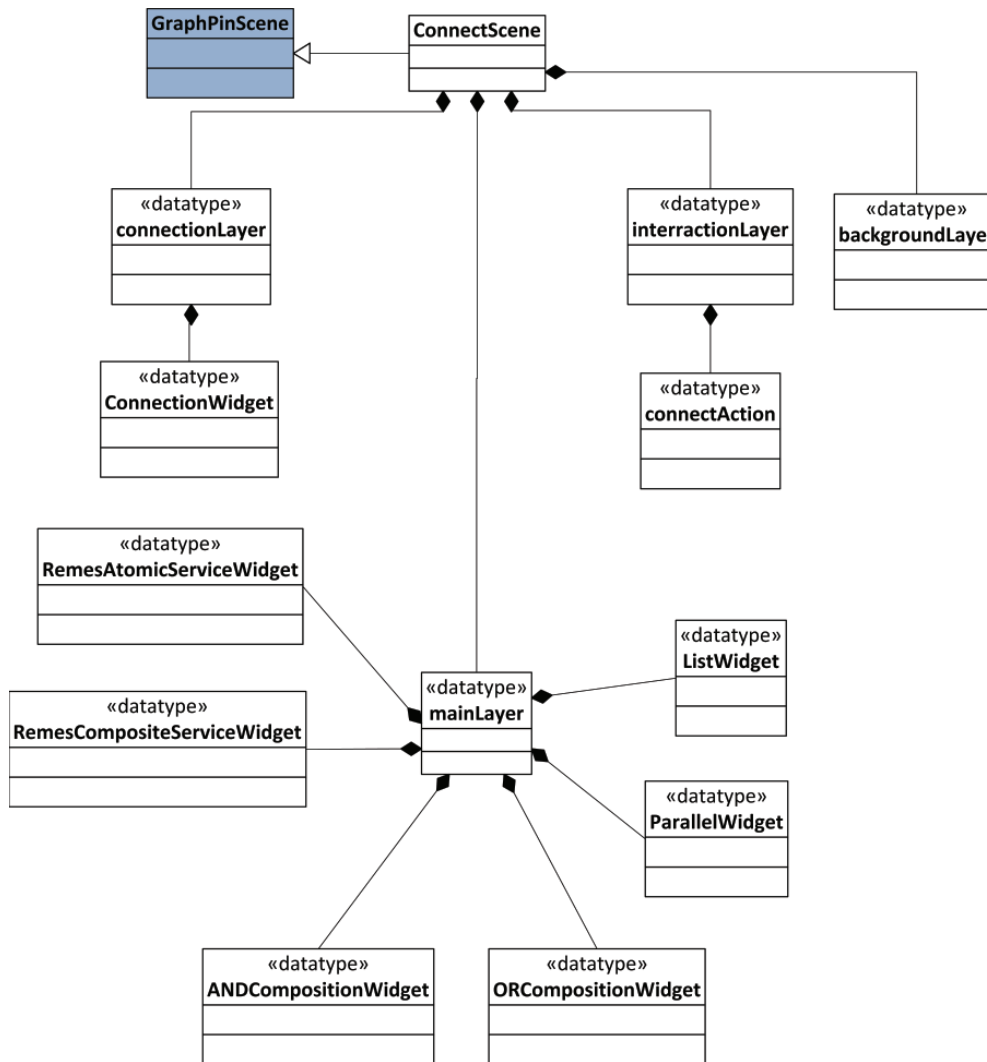


Figure 3.5: ConnectScene Structure as a Class and datatype diagram

used to arrange the inner elements in a vertical layout. In the top part, the widget image and widget name are located. We also use some horizontal divider lines that are realized by using a separator widget. In the lower part of the atomic service widget, we have divided service attributes and mode properties in two specialized widgets. Inside these widgets, all sub-components are arranged in VerticalFlowLayout in order to stack them vertically.

The attributes widget, depicted in Figure 3.8, is the visual representation of service attributes, as described in Section 2.1.3. It controls how attributes are drawn in ConnectScene. In the upper part of the widget, we have implemented a LabelWidget named attributes label widget. When the user clicks on this label, a SwitchCardAction() method is called. This action is required for switching between widgets that are located in the attributes container. When the user collapses attributes container, only the attributes label is shown in the par-

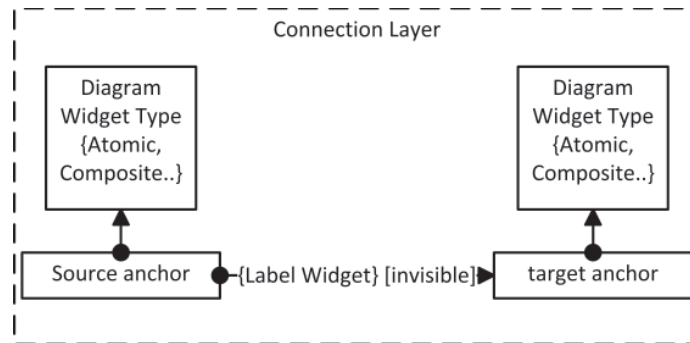


Figure 3.6: Composition of the Connection Layer in the ConnectScene

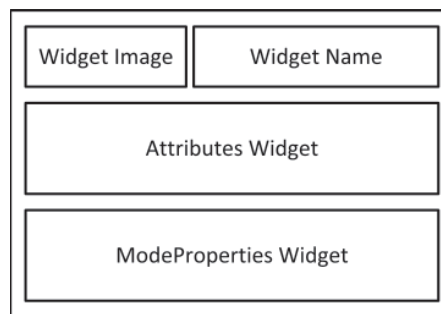


Figure 3.7: Construction of an atomic service widget

ent widget. When expanding the active widget, all components contained in the attributes container are shown. Inside the attributes container, a vertical flow layout is used to arrange the inner elements. These inner elements are of type `ComponentWidget` and therefore used to display and update a Swing component.

Mode properties widget, depicted in Figure 3.9, is the visual representation of constants, variables, and resources (also called mode properties) that are attached to a REMES mode. It extends the `Widget` class and controls how mode properties are drawn on the scene. In the upper part of the widget we have implemented a label named `ModePropertiesLabel` widget. When the user clicks on the label a `SwitchCardAction()` method is selected. This action is required for switching between widgets located in the properties container. When the properties container is collapsed, only the attributes label widget is shown. When switching the active widget, the widgets contained in the properties container are shown. Inside this container, a vertical flow layout is used to arrange the inner elements.

The atomic service widget has two pins as control points: entry pin and exit pin. When created in the diagram, these two pins are automatically attached to an atomic service widget by calling `attachPinWidget()` method as shown in Figure 3.10.

### 3.4.2 Parallel Connector Widget

Figure 3.11 shows a schematic diagram that represents a parallel widget. This type of widget is used to model a parallel connector. We consider that a parallel connector widget is a widget annotated with entry and exit points. In order to give it a circle-like form,

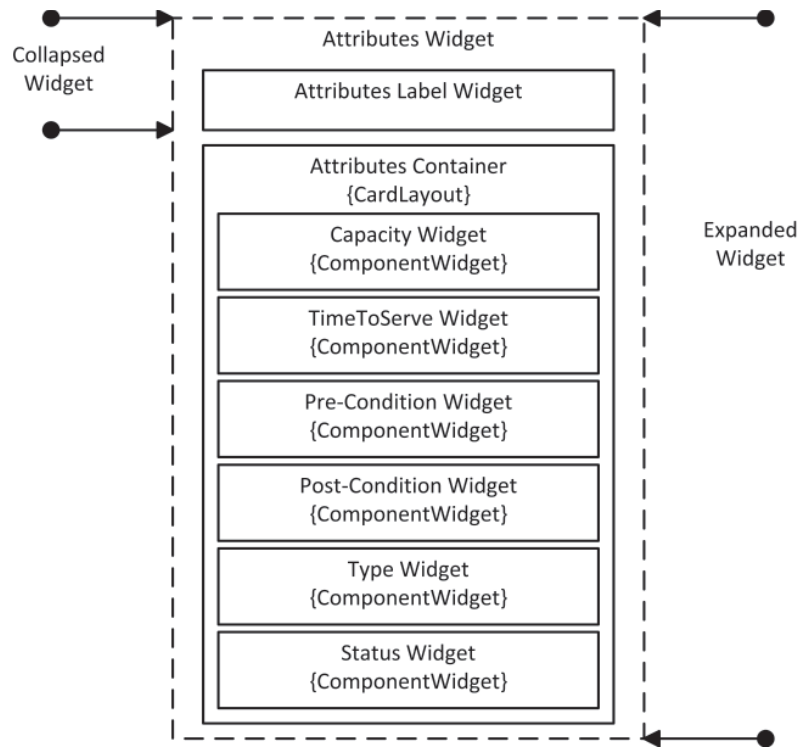


Figure 3.8: Attributes widget composition inside an atomic service widget

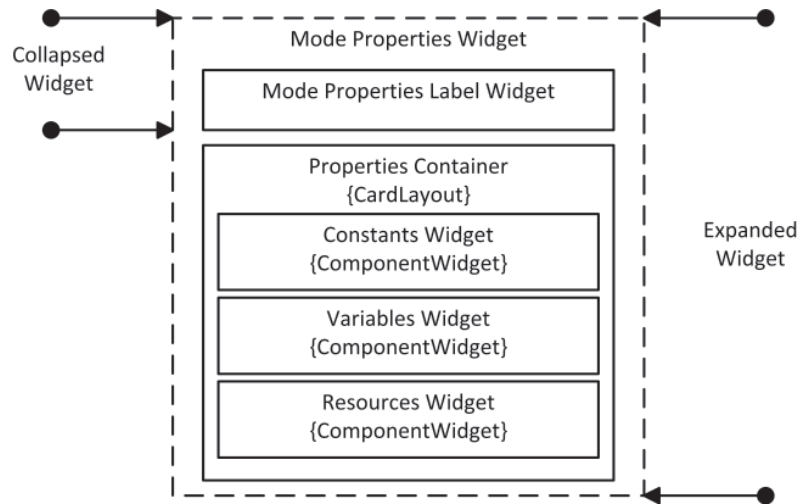


Figure 3.9: Mode properties widget composition inside an atomic service widget.

we use a function name *paintWidget()* which uses the Graphics2D instance acquired from Scene.getGraphics method within the Visual Library API.

In order to represent services in a parallel composition, two widgets are created, one at

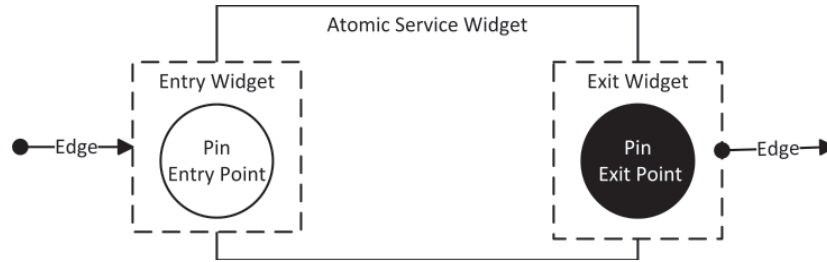


Figure 3.10: Entry and exit widgets composition inside an atomic service widget

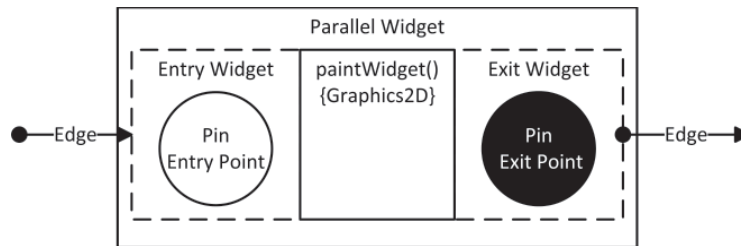


Figure 3.11: Parallel widget composition

the start of the composition and the other at the end. Between two parallel widgets, services can be connected as a part of the parallel composition.

### 3.4.3 Composite Service Widget

Figure 3.12 shows a schematic drawing of a composite service widget. The outer black line is the border of the widget. Inside the widget, a vertical flow layout is used to arrange the inner elements in a vertical layout. In the top part, the image and name widgets are located. In the lower part of the composite service widget, we have divided the service attributes and mode properties in two specialized widgets. Inside these widgets, all sub-components are arranged in vertical flow layout in order to stack them vertically.

From the requirements defined in Section 3.2, we know that a composite mode can contain any number of atomic modes. Therefore, to implement a nested display model we use a `ComponentWidget` as a container. By using this type of widget another scene can be declared inside it. This sub-scene represents the connection between the composite service control points and the control points of its sub-modes. We define here the sub-scene as the inner container which shows the sub-modes.

### 3.4.4 AND/OR Mode

Figure 3.13 depicts an AND/OR mode widget, which is a display model of a AND/OR mode. A vertical flow layout is used to arrange the inner elements in a vertical layout. In the top part, image widget and name widget components are located.

AND/OR mode is a special kind of mode which contains as sub-modes services. Therefore, in the lower part of the AND/OR mode widget, we implement a nested display model that uses a `ComponentWidget` for representing the connection between the AND/OR mode control points and control points of its sub-modes.

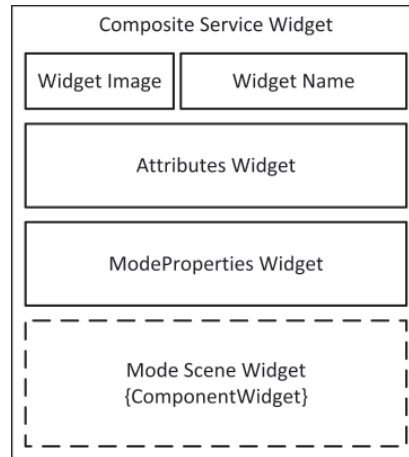


Figure 3.12: Construction of a composite service as a widget

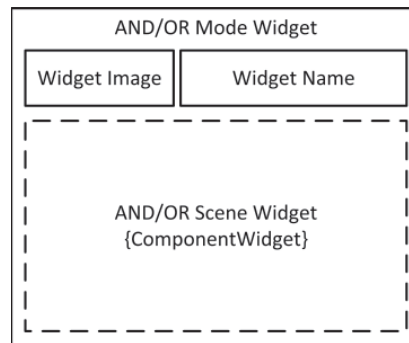


Figure 3.13: Construction of an AND/OR mode as a widget

### 3.4.5 List Widget

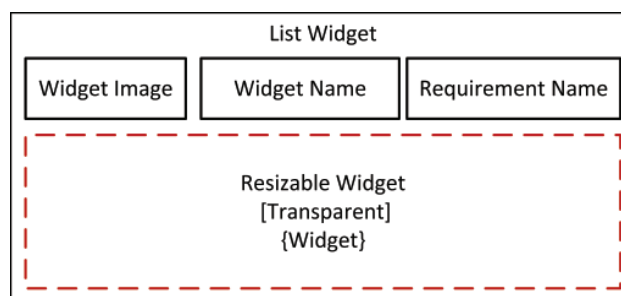


Figure 3.14: Construction of a list as a widget

In order to implement a visual representation of a list we have composed a special kind of resizable and transparent widget. The widget, as shown in Figure 3.14, is created with a

resizeable border by using the class `ResizeBorder` in order to expand the widget over other widgets. In the top part of the list widget we include an image widget, name widget, and a requirement predicate widget. This requirement widget represents a requirement (REQ), as defined in Section 2.1.3.

### 3.5 Composite Mode Scene

The main container for a composite service widget is modelled as a `ComponentWidget`. Within this class we use a `JComponent`, which serves as a placeholder and is responsible for displaying and updating the contained sub-scene. This sub-scene, as shown in Figure 3.15, is called `SubModesScene` and it extends the `GraphPinScene` abstract class. The function of `SubModesScene` is to manage widget mapping to an associated data model, which is in our case a structure with two types of nodes (atomic modes and conditional connectors), entry and exit pins, and edges annotated with guards and actions.

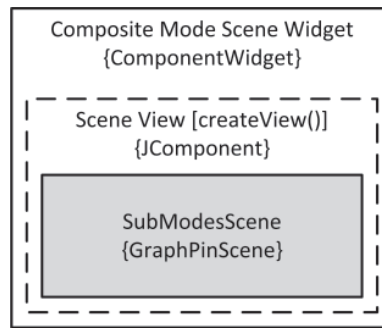


Figure 3.15: Construction of a `SubModesScene` for composite service widget

Four pins are attached to a composite service widget: entry, init, write, and exit. When a composite service widget is created, four pins are graphically attached with the `attachPinWidget()` method, as shown in Figure 3.16. These pins are defined in `ConnectScene` class, which is the external scene. Inside the composite service widget, we create four additional pins defined in `SubModesScene`, which is the internal scene. These inner pins correspond to the outer pins. The idea behind this graphical container is to describe a control flow between `ConnectScene` and `SubModesScene` using Visual Library API.

The diagram in Figure 3.17 shows a simplified data structure of classes used in `SubModesScene`. The abstract class `GraphPinScene` is coloured with blue to show that it is contained in Visual Library API. `SubModesScene` is the root element of a composite service widget. It is responsible for control and representation of the rendered area inside a composite service widget. `SubModesScene` extends the `GraphPinScene`. Like `ConnectScene`, is represented by four transparent layers of type `LayerWidget`: `MainLayer` (atomic modes and conditional connectors are defined and used within this layer), `ConnectionLayer` (edges are defined and used within this layer), `InteractionLayer` (used when the user wants to connect two nodes by an edge), and `BackgroundLayer`.

For managing edges inside a `SubModesScene` we provide a separate `LayerWidget` named `ConnectionLayer`. The creation of edges depends on the layer's subclasses and it relies upon the `SubModesScene`. As shown in Figure 3.18, all edges in `SubModesScene` are marked with guards and actions. For managing the interaction between nodes and edges in `SubModesS-`

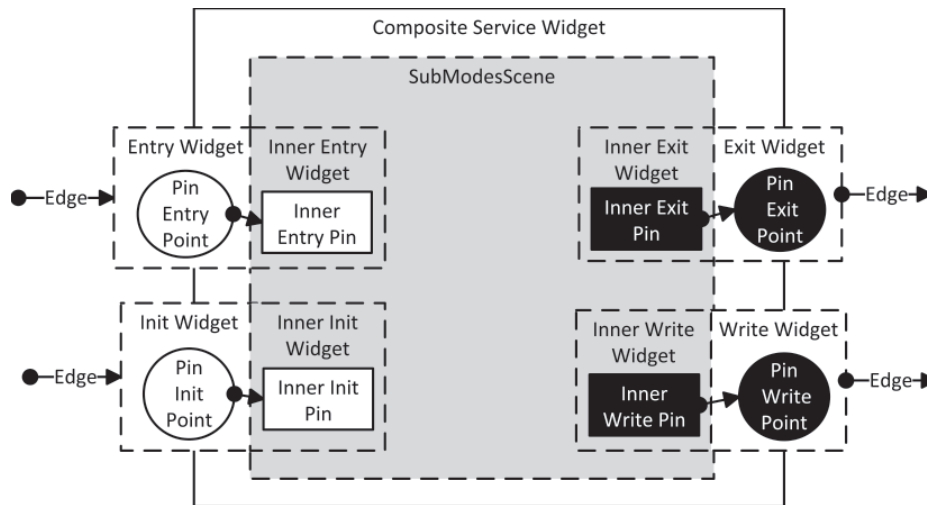


Figure 3.16: Composition of entry, init, write, and exit widgets inside the composite service widget

cene we use a `LayerWidget`. `MainLayer` is a separate `LayerWidget` used in the same way as described for `ConnectScene`.

### 3.6 AND/OR mode scene

An AND/OR mode contains as sub-modes services needed to be synchronized. In order to model the inner container, we have introduced a new `ComponentWidget` class, as depicted in Figure 3.19. Within this class we set up a `JComponent`, which serves as a container for the current hierarchy of displayed service widgets. Similar to composite service widget, we define a sub-scene inside the `JComponent`. The sub-scene function is to manage widget mapping to an associated data model, which is in our case a structure with one type of nodes (atomic services), entry and exit pins, and edges annotated with guards and actions. As shown in Figure 3.20, two pins are attached to the AND/OR mode widget: entry and exit pins. These pins are mapped to `ConnectScene` class. Internally we create two additional inner pins which are mapped in `AND(OR)ModeScene`. These inner pins are corresponding to the outer pins. The role of this container is to compose a control flow relationship between `ConnectScene` and `AND(OR)ModeScene`.

Like `ConnectScene`, `AND(OR)ModeScene` is represented by four transparent layers of type `LayerWidget`: `MainLayer` (atomic services are defined within this layer), `ConnectionLayer` (edges are defined and used within this layer), `InteractionLayer` (it is used when the user wants to connect two nodes by an edge), and `BackgroundLayer`.

### 3.7 Console View

Console View is a functional unit responsible for transforming a diagram model (data model) to a textual description of a service and their attributes. It also supports specification of service's correctness conditions using strongest post condition predicate transformer [12].



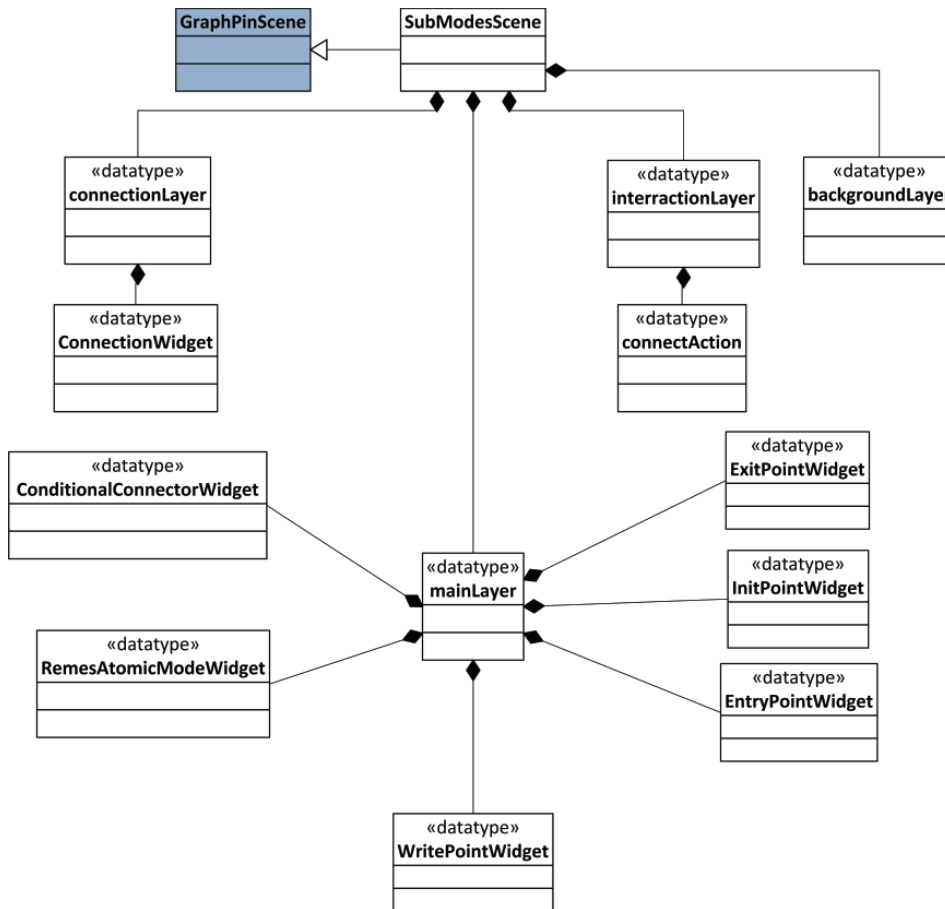


Figure 3.17: SubModesScene structure as a class diagram

Because our framework is based on Java, we have used parsing and manipulation of XML documents with one of the most well known standards called DOM (Document Object Model)<sup>3</sup>.

### 3.7.1 Parsing with DOM

DOM defines the API of a parse tree for XML documents. The `org.xml.dom.Node` interface specifies the basic features of a node in this parse tree. `Document`, `Element`, `Entity`, and `Comment` interfaces are defining the features of specific types of nodes.

The DOM API enables the XML parser to specify a structure. The content of the XML document is parsed in the form of a tree. The DOM tree consists of several nodes that represent elements, text, etc. defined in the XML document<sup>4</sup>. These nodes are arranged according to the structure of the XML document. The XML parser stores the DOM tree

<sup>3</sup>The DOM API is a standard defined by the World Wide Web Consortium (W3C); consists of the `org.w3c.dom` package and its subpackages.

<sup>4</sup>Refer to the javadoc documentation of the interface `org.w3c.dom.Node` for a complete list of node types

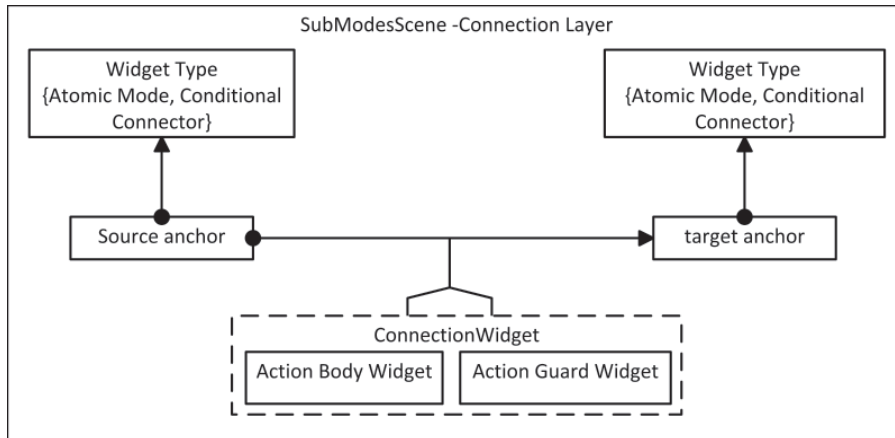


Figure 3.18: Composition of ConnectionLayer in a SubModesScene

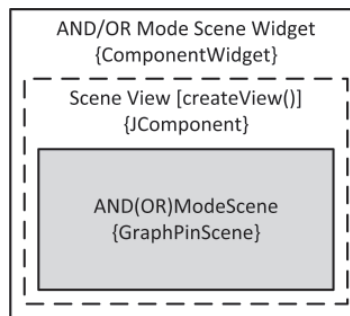


Figure 3.19: Construction of the AND(OR)ModeScene for AND/OR mode.

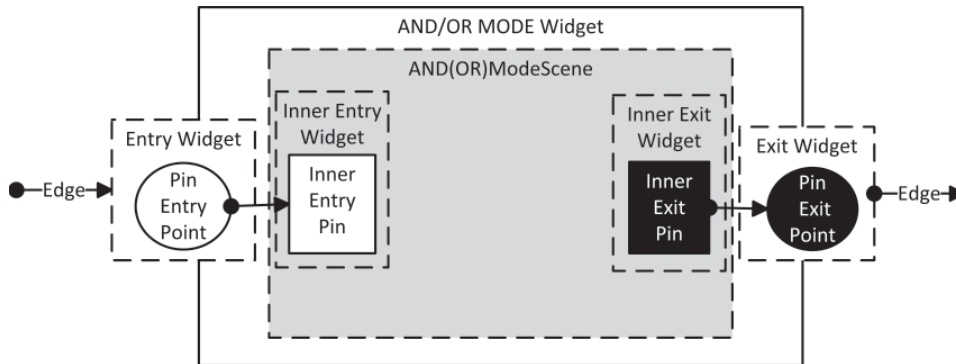


Figure 3.20: Composition of entry, init, write, and exit widgets inside the AND/OR mode widget

in memory. The DOM tree enables an application program to access and modify the XML document it represents.

There are a couple of drawbacks related to using the DOM processing model [18]:

- DOM is memory intensive because the parsed document is instantiated into memory before it can be used;
- DOM is slow when used for processing large XML documents.

### 3.7.2 Generating the hierarchical language from the data model

An XML parser is a software module that reads an XML document and verifies that the content and structure of the XML document conforms to the rules specified in its corresponding grammar(i.e. DOM). The data model is used to store the diagram in an XML document. Based on this stored model we can generate the declarative part of the on-the-fly service composition defined in [10].

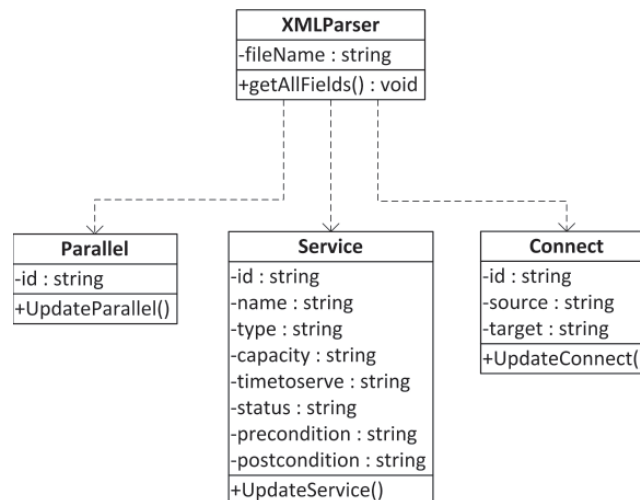


Figure 3.21: XML Data file as a class diagram

Figure 3.21 shows the XML Parser class. This class uses four types of classes in order to parse the XML file. The classes are defined as follows:

- **Service class**: stores the information for both atomic and composite services, and it has as parameters the service id, the service name, all 6 attributes (type, capacity, time-to-serve, status, pre-, and postcondition) and the X and Y coordinates for the location;
- **Connect class**: stores the information for the sequential connectors and has as parameters the connector id, the source and the target of the connection;
- **Parallel class**: stores the information about the parallel connectors and has as parameters the connector id and X and Y coordinates for the location in diagram;
- **ServiceList class**: stores the information about the list used in the diagram and has as parameters the list id, list name, the X and Y coordinates, height and width, and the DCL/HDCL name.

We use the `getElementsByTagName()` method to return a node list of all elements defined in the data model. The node list is an abstraction of an ordered collection of nodes, without

defining or constraining how this collection is implemented. From this node list we retrieved all the necessary child elements using the *getAttribute()* method.

The retrieved information is used to form a list of elements of different types like: Service, Connect, Parallel, and ServiceList. After all the information from the XML file has been organized we can start to manipulate the information. The needed services are introduced through the declarative part as follows:

$$\text{declare } service\_name ::= \langle name, type, capacity, time\_to\_serve, status, pre, post \rangle \quad (3.1)$$

The instances of the selected services and service lists are created:

$$\text{create } service\_name; \text{ create } list\_name; \quad (3.2)$$

After the lists are created we add to the list all the services that are modelled in the diagram:

$$\text{add } service\_name \text{ to } list\_name; \quad (3.3)$$

At this point we can establish which service belongs in which service list, by checking which service has its coordinates inside the service list area, that is calculated from X, Y coordinates, height and width elements retrieved from the data model. Finally, nested services are composed by DCL [10]:

$$DCL\_count ::= (list\_name, PROTOCOL, REQ) \quad (3.4)$$

To provide means to compose the existing DCLs with other services or DCLs, we introduce a HDCL nested composition as follows:

$$HDCL\_count ::= ((DCL\_count^+, PROTOCOL, REQ) \quad (3.5)$$

The "+" operator is used to express that one or more DCLs and/or services are nested through HDCL. The protocol defines the way services are composed. Checking the requirements is shown in the form of the strongest postcondition predicate transformer [12] as follows:

$$\text{check}(sp.(system\ composition).(system\ precondition) \Rightarrow (system\ requirements)) \quad (3.6)$$

At this moment the algorithmic computation of strongest post-conditions for HDCL is not yet implemented, and therefore we only show that we can automatically derive the check operator in the correct form.

### 3.7.3 Generating the data model out of hierarchical language

There are cases when we want to generate data into the diagram from the HDCL declarations, for example when the user modifies some service attributes. We parse the information contained in HDCL declarations and we convert them into the data model. HDCL declarations are saved in a file. We use two java.util<sup>5</sup> classes: Scanner and regex. By using the Scanner class we break the file into tokens using a delimiter pattern. The resulting tokens are converted into values by using various *next* methods.

<sup>5</sup>Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

We use the *nextLine()* method defined in *Scanner* class to parse throughout every line in the HDCL file. Using the *hasNext(String pattern)* method we determine which lines are starting with "declare". Using the method:

```
findInLine("declare(.+) ::= < (.+), (.*), (.*), (.+), (.+), (.+) >;") ,
```

together with the *MatchResult result = scanner.match()* method from the *regex*<sup>6</sup> package we create a resulting array that contains the name and the attributes of every service declared in HDCL.

By providing this automated way of ensuring traceability between the diagram and HDCL declarations, we have intended to facilitate efficient design through run-time of service compositions.

---

<sup>6</sup>Classes for matching character sequences against patterns specified by regular expressions.

## Chapter 4

# User Interface

The user guide introduces the most important functionalities of the GUI for service behaviour modelling. The tool is implemented completely in Java, it is platform independent, and thus requires Java Runtime Environment (JRE)<sup>1</sup>. The user interface is implemented with NetBeans IDE. In the first stage of the project some testing has been performed on building a GUI with the drag and drop form editor<sup>2</sup>. However, we have experienced that the code generated automatically by NetBeans GUI design tool has not been efficient and it was difficult to extend its functionality any further. Therefore the user interface was coded from "scratch" (using Swing JFC and Visual Library API as base libraries).

The tool consists of a Java application which is used to design, display and analyse services and their attributes. It has to be mentioned that no continuous scenario is considered in this chapter. Also, it is important to mention that in the user guide the explanations will be restricted to a user interface perspective.

### 4.1 GUI Overview

When the tool is launched a main view is opened. As depicted in Figure 4.1, this view is dominated by a large modelling area, also called Diagram Editor View (DEV). The main concern during implementation was testing and manipulating the Visual Library API in DEV, in order to make sure that screen actions attached to elements functioned correctly. Apart from this, there is a menu bar at the top of the view, a Palette at the right side, and a Console View in the bottom side. In the next sections we will give a short description of the containing components.

#### 4.1.1 Diagram Editor View

The Diagram Editor View contains a Main Menu at the top, a Palette at the right and a Diagram Scene at the center. Figure 4.1 shows DEV and the corresponding containers.

##### Main Menu

The menu consists of two top menus, the file menu and the help menu (to open the User Manual). The file menu has the following menu items:

---

<sup>1</sup>The tool requires that a Java Runtime Environment (JRE) be installed on your machine to run. Java SE 5 or greater is recommended.

<sup>2</sup>GUI design tool in NetBeans, formerly known as project Matisse.

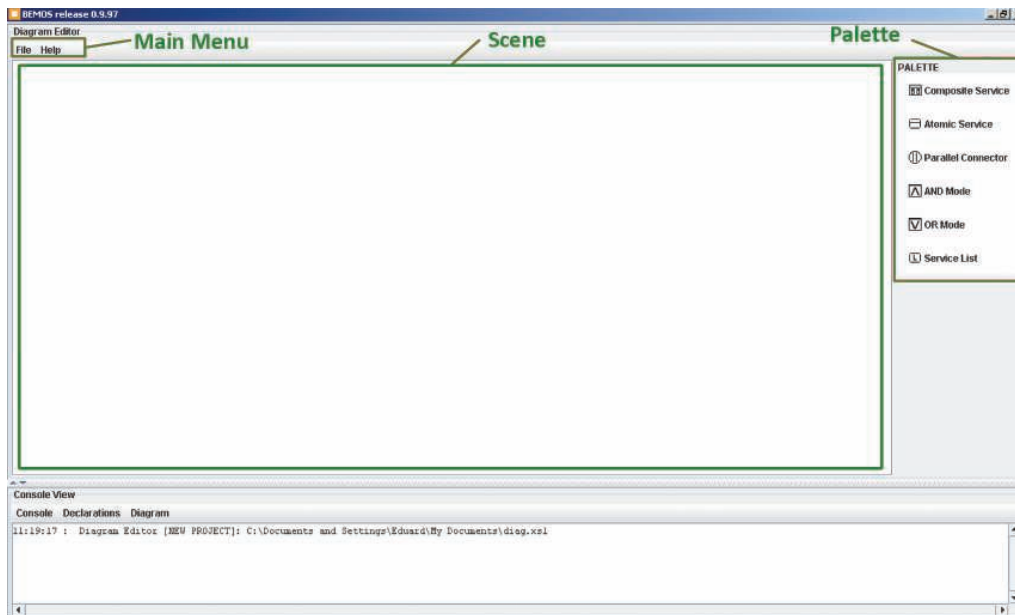


Figure 4.1: Diagram Editor View

- New Project: create a new project. It requires the user to state the name of the project;
- Load Project: open an existing project;
- Close Project: close the current project;

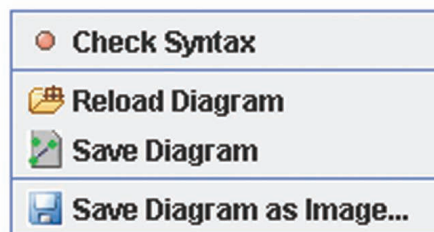


Figure 4.2: Additional Menu

Depending on the user actions the Palette can be visible or hidden. Only after the user creates or opens a project file, Palette and Diagram Scene are ready to use. Creating a new project in this tool is an easy task. With the use of New Project menu a File Chooser Window is opened, and the user can assign the project name.

Also, Diagram Scene has an extra functionality, by right clicking on a scene a pop-up menu with the following menu items appears:

- Check Syntax: partially verifies the correctness of the specified Attributes and Mode Properties, and if necessary can displays an error message in the Console View;
- Reload Diagram: reloads the diagram;
- Save Diagram: saves the diagram;
- Save Diagram as Image...: exports the diagram into a .png image file.

Currently, a diagram can be saved as an XML document file. This format saves all nodes, edges and pins separately. XML is proven to be an easy to use format for keeping the diagrams flexible and portable to other applications. Once a diagram is saved it can be reloaded in case of any changes in the Console View. The Reload Diagram menu item is implemented using a part of the XML handler used to load a project.

## Palette

As depicted in Figure 4.1 the Palette is a vertical group of tools located on the right-hand side of the DEV. This toolbar provides a quick access to the most import work tools used in this application. The Pallette allows the user to create different graphical elements like:

- Atomic Service: to create a REMES atomic service;
- Composite Service: to create a REMES composite service;
- Parallel Connector:to create a parallel connector;
- AND Mode: to create a REMES AND composite mode;
- OR Mode: to create a REMES OR composite mode;
- Service List: to create a list constructor.

When a new project is created, the user can pick up elements from the Palette. Every element in the Palette has an icon and a name corresponding to REMES elements.

## Current Scene

The Current Scene is the initially white zone visible in Figure 4.1. Our tool enables the user to develop REMES Service diagrams in a quick and simple way. When elements are chosen from the Palette, they are shown in the Current Scene view. Every element can be renamed, moved, and linked to other elements.

### 4.1.2 Console View

As depicted in Figure 4.3, Console View is responsible for showing the transformation of a diagram to a textual description according to HDCL as stated in Section 3.7. It contains two main parts: a Console Window and a Console Menu where the user can generate the declarations, save them in a .hl file for storing purposes and generate the diagram from the .hl file.



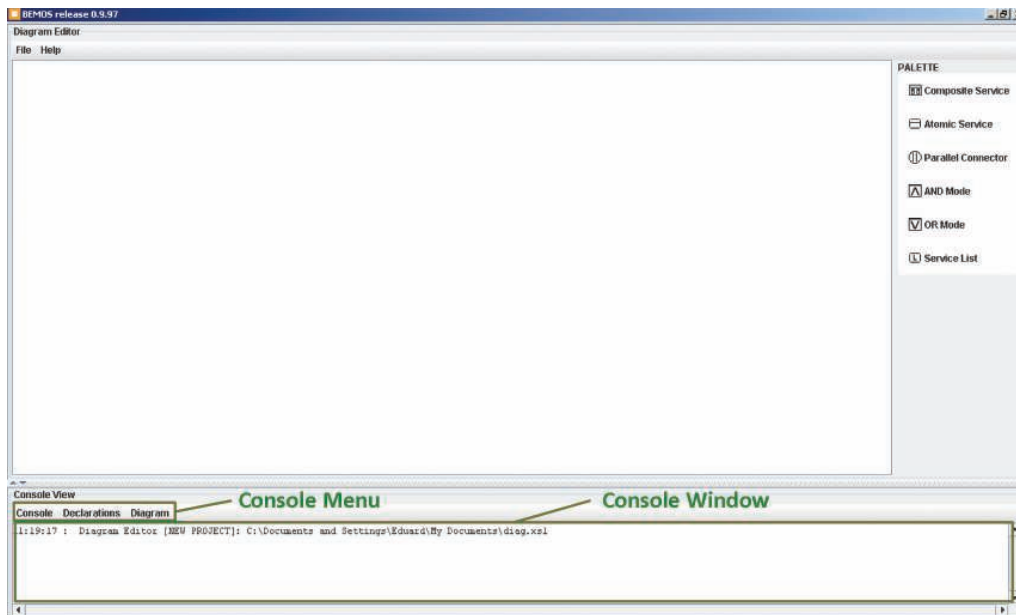


Figure 4.3: Console View

## 4.2 Diagram

When added to the diagram, all elements are placed in the upper right part of the diagram and can be moved to the desired position by left clicking on the element and holding the mouse button pressed and dragging the element to the desired position. The elements can also be deleted. The user can do this by right clicking on the element → Delete.

### 4.2.1 Atomic service

After a project has been opened, the user can start to create a REMES SOS diagram. To add a new atomic service one has to follow next steps:: Palette → Atomic Service. To edit the service name select "service name" and type over the desired name for the service(see Figure 4.4).

To fill in the service attributes the user has to click on the "Attributes" bar for the list of attributes to become visible. Capacity field specifies the maximum service frequency and it accepts only natural numbers. The user also can specify the worst-case time needed for a service to respond to a given request (also natural number) in the Time to Serve field. Pre-, and postcondition fields can also be specified by providing a predicate condition. The user can also choose the service status and service type by choosing one of the available options.

Every atomic service has an entry point (white circle on the left-side of the service) and an exit point (black circle on the right side).

### 4.2.2 Composite service

To add a new composite service the user should click on: Palette → Composite Service. Mode properties and attributes are specified in "Attributes" and "Mode Properties" fields.

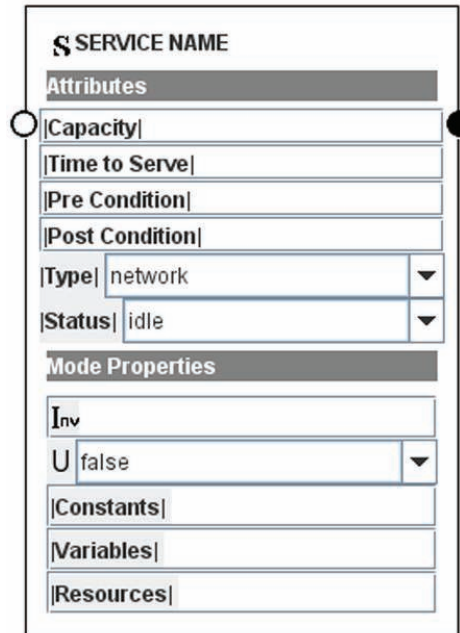


Figure 4.4: Atomic Service

To add sub modes, the user should right-click on the composite service scene and a pop-up menu will appear. From this pop-up menu the user can select "Add New Atomic Mode" or "Add New Conditional Connector", the elements available for composition. To resize the composite service scene one has to left click on the lower right corner of the service and hold the mouse button pressed changing the size of the service. In addition to the entry/exit points, the composite service has two control points more: init point (lower left side) and write Point (lower right side).

### 4.2.3 Sequential Connection

The sequential composition defined in [10] is represented here by the sequential connection. The need for this composition comes from cases when two or more services are executed in a sequence, uninterrupted. To connect two elements from the REMES SOS diagram the user has to identify the Exit Point of the first element and the Entry Point of the second element. To connect this two points, the user should press Ctrl key and the left mouse button on the "Exit Point" and start dragging the arrow that appears to the "Entry Point" and then release the buttons. The direction of the arrow is important, because the starting element is executed before the target element.

### 4.2.4 Parallel Connection

The Parallel Composition defined in [10] is represented by the parallel connectors. To create a parallel connection we need to add two parallel connectors, one that represents the start

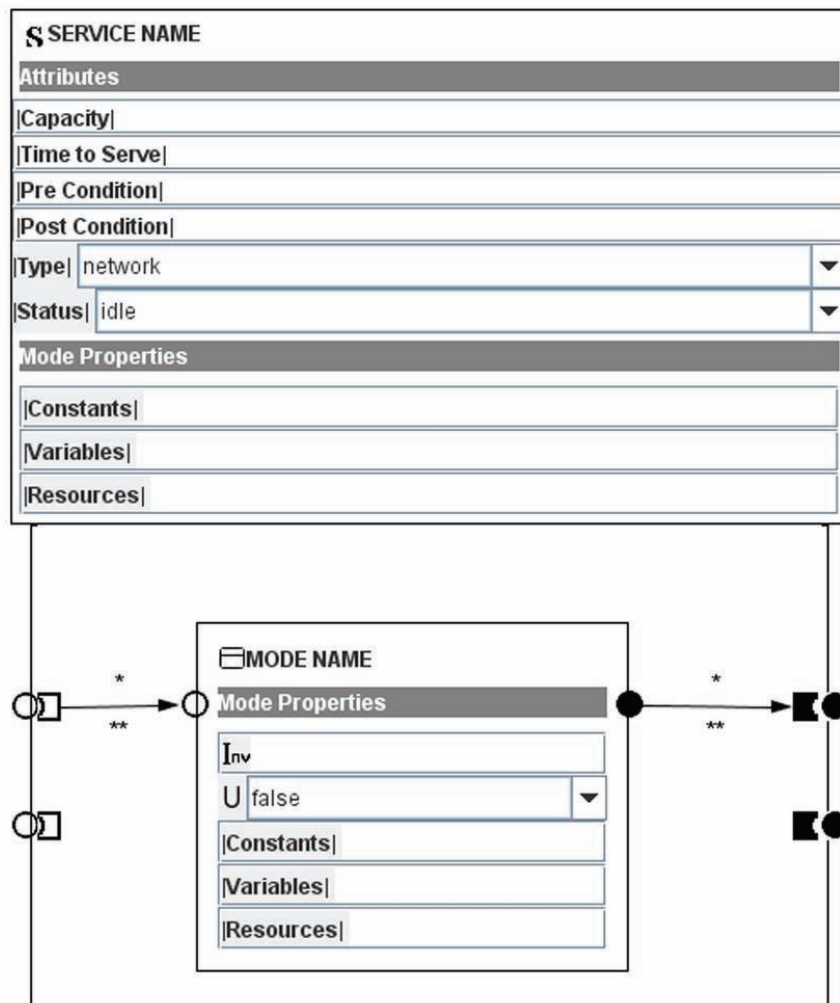


Figure 4.5: Composite Service

of the parallel area and another one that represents the end of the parallel connection. To add a new parallel connector: Palette → Parallel Connector.

#### 4.2.5 AND/OR Mode

The parallel composition with synchronization defined in [10] is represented by an AND/OR mode. AND and OR modes are used to model synchronized behaviour. Depending on whether the services need to be entered simultaneously or not, we can add to the diagram either a AND mode, or a OR mode.

To add an AND mode: Palette → AND mode. To change the name of an AND mode, the user has to select "AND Mode Name". To resize the mode the user has to click on the

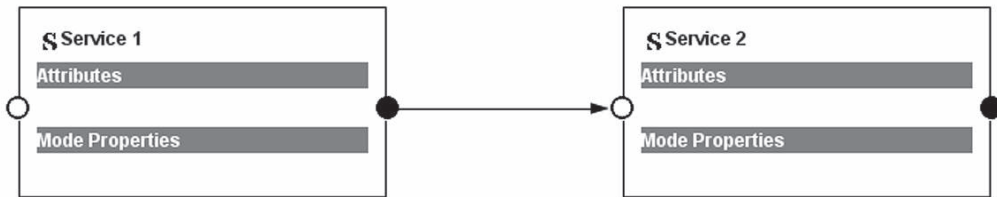


Figure 4.6: Sequential connection

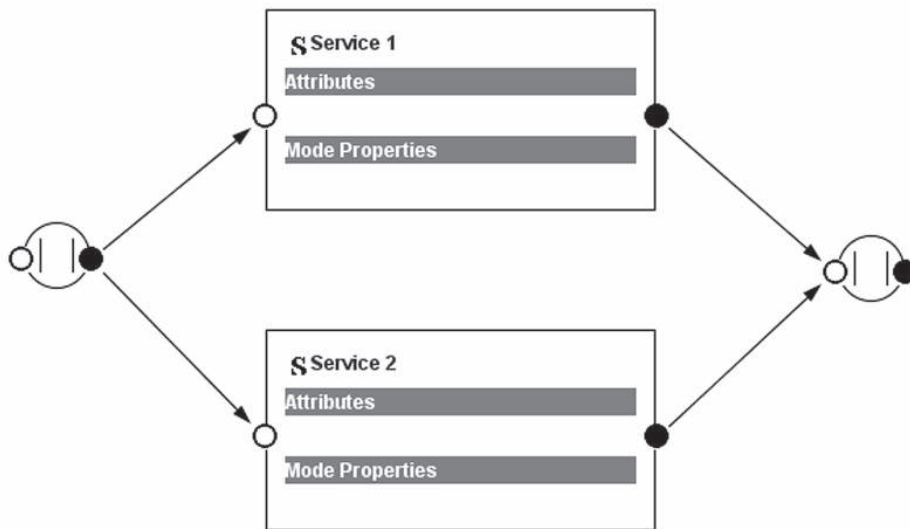


Figure 4.7: Parallel Connection

lower right corner and drag it till the AND mode has the desired dimension. The user can add atomic services to AND mode by clicking the mouse right button inside the mode scene and selecting from the pop-up menu "Atomic Service", as depicted in Figure 4.8.

The connection between a service and an entry/exit point of the mode is done by first identifying the entry point of the AND Mode and the Entry Point of an atomic mode. To connect two control points the user should press the Ctrl key and the left mouse button when clicking on the source point and start dragging the arrow to the target point and then release the buttons. Depending on the required synchronization type, AND modes can be employed with either "and" or "max" synchronization by right-clicking inside the mode.

To add an OR mode: Palette → OR Mode. In the same way as for an AND Mode the user can change the name of the mode. The OR mode entry point is used for executing one or more of the constituent services. Manipulation with an OR mode is similar to the AND mode. The difference are the entry edges marked with \*, that might be annotated with guards. Depending on the required synchronization type, OR modes can be employed also with either "and" or "max" synchronization by right-clicking the inside of the mode.

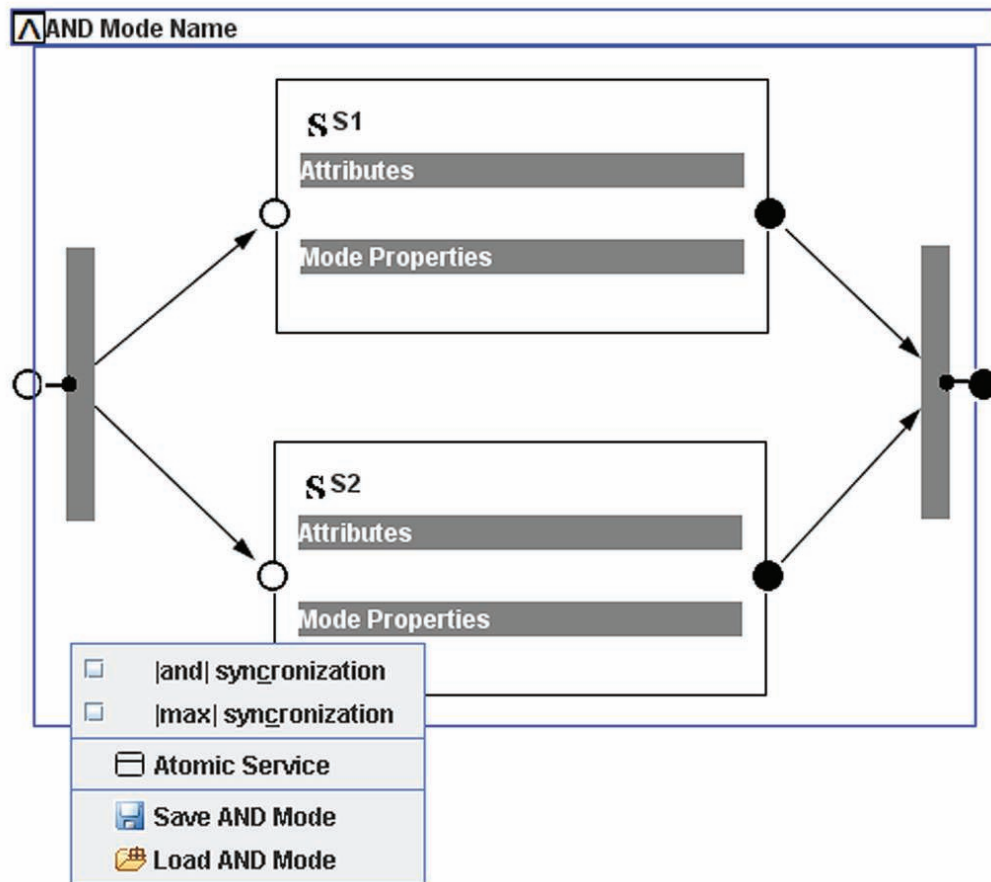


Figure 4.8: AND Mode

#### 4.2.6 Service List

The List Constructor defined in [10] is given here as service list. To add a new service list from the palette, the user has to select service list (Palette → Service List). To change the name of the list the user can select "List Name" and type over the desired name. The user has to select the service list requirement. The list can be moved and resized in order to encapsulate all the desired services, and parallel connectors if needed. As depicted in Figure 4.10, three services are connected in sequential composition. Only *SERVICE\_1* and *SERVICE\_2* are added to the service list because they are included in the red line bounds.

### 4.3 Data Model and HDCL

As depicted in Figure 4.11, in Console View the user can select the menu item "Declarations", then "Generate Declaration" and the resulting textual service composition will be automatically be generated. We illustrate the use of the hierarchical language for modeling

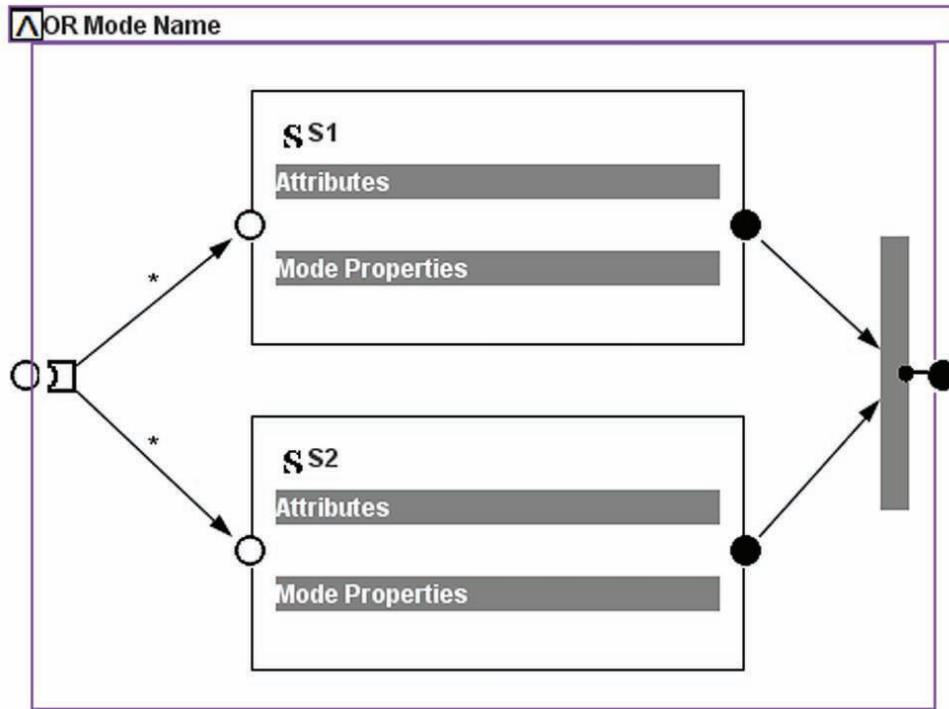


Figure 4.9: OR Mode

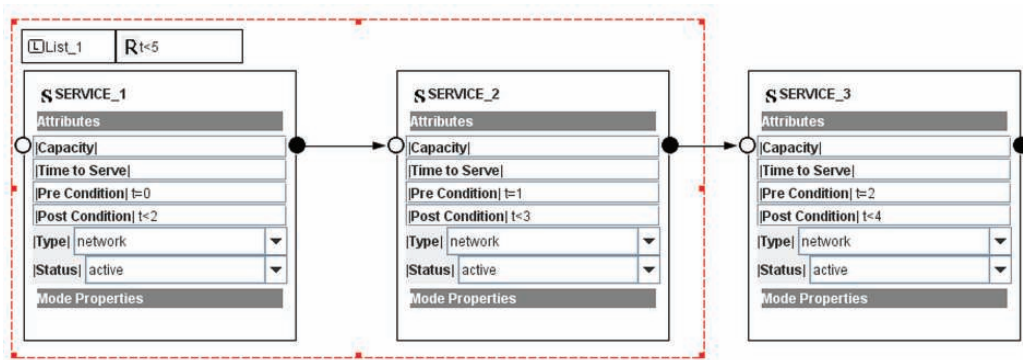
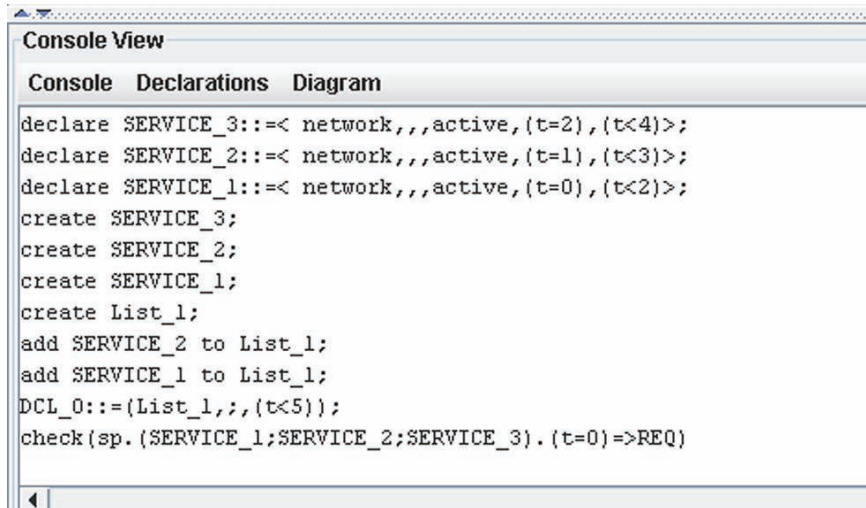


Figure 4.10: Service List

service composition, as depicted in 4.11, on the example diagram shown in Figure 4.10. The needed services are introduced through the declarative part (first three lines in Console Window that start with *declare* operation). A service declaration contains the service name, type, status, TimeToServe, pre-, and postcondition. This information is used when choosing a service. After the selection, the instances of the selected services are created (with *create* interface operation), and added to the service list using the *add* command. Afterwards, the chosen services are composed by DCL.0. The list of services, sequential composition

(service binding type ”;”), and DCL\_0 requirements are given as parameters.

The advantage of HDCL language is that, after each diagram composition, one can check whether the given requirement is satisfied, by forward analysis, e.g., by calculating the strongest postcondition of a given composition. As depicted in Figure 4.11 in Console View we show the *check* command for the given composition. The result of the command *check* should return a main proof obligation that is not yet implemented in this design framework.



```

Console View
Console  Declarations  Diagram
declare SERVICE_3::=< network,,,active,(t=2),(t<4)>;
declare SERVICE_2::=< network,,,active,(t=1),(t<3)>;
declare SERVICE_1::=< network,,,active,(t=0),(t<2)>;
create SERVICE_3;
create SERVICE_2;
create SERVICE_1;
create List_1;
add SERVICE_2 to List_1;
add SERVICE_1 to List_1;
DCL_0:=(List_1,,, (t<5));
check(sp.(SERVICE_1;SERVICE_2;SERVICE_3).(t=0)=>REQ)

```

Figure 4.11: HDCL manipulation inside Console View

To generate the diagram after describing the system in Console View, the user should enter the menu (Diagram → Generate Diagram). The last step is to reload the diagram in the Diagram Editor View by right clicking on → Reload Diagram. In this way new services can be instantiated in Console View and added to the diagram for further compositions.

## Chapter 5

# An Illustrative Example

To demonstrate the capabilities of our design framework, we consider an example scenario taken from the real-time systems domain. Let us consider an Automated-Teller Machine (ATM) which is a typical design case in safety critical real-time systems and software systems modelling [21, 23, 31, 22].

### 5.1 ATM Description

As a control system, the ATM is characterized by a high degree of complexity and elaborate interactions with different hardware resources. These factors are making the ATM system an ideal SOS example. When designing the controller for an ATM system, one may have a scenario in which:

- a customer inserts his/her card;
- the ATM asks for the password;
- the customer gives the password;
- ATM verifies the password;
- the customer verifies the account balance;
- ATM shows the balance;
- and so on.

This scenario describes some possible interactions between the ATM system and the customer, but clearly not the only one. One can have another scenario, for example one in which the customer enters a wrong password. Our design framework deals with this scenarios by using services as basic units. Figure 5.1 depicts a simplified conceptual model of an ATM system. A service repository is available to the users, which consists of several services defined using REMES extended interface and the behavioural language. This registered services can be invoked and composed in different ways , based on the preferences of the user.



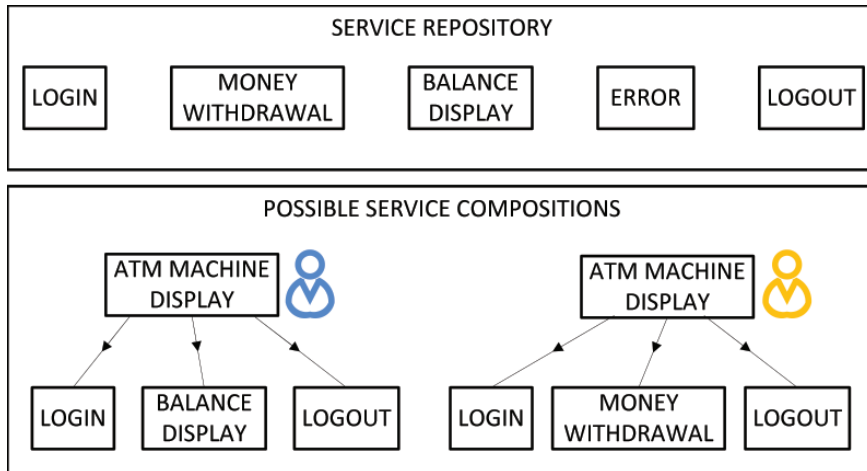


Figure 5.1: Service oriented ATM system

## 5.2 Service Repository

We define the service repository as a collection unit, which allows the users to obtain and modify the interfaces of the registered services to which they have access. Using the repository, services can update their published interfaces. By supporting these interface services, REMES provides both an invocation capability as well as the flexibility afforded by the behavioural description. Here, we exemplify the internal behavioural modelling, as well as the available interface of a simple ATM system. The ATM machine has a display through which the customer communicates with the bank. For simplicity we will not model the communication of the ATM with the bank. Each customer is required first to login and depending on the success/failure can proceed with different transactions (e.g., balance display, money withdrawal) depending on the chosen service composition.

The *LOGIN* service, depicted in Figure 5.2, is modelled as a REMES composite service. It is responsible for validating the customer login scenario. The mode consists of two atomic modes: *INSERT CARD* and *TYPE PIN*. The mode waits for the customer to insert his/her card (which will modify the global variable *card* of type Boolean) and then is validating the password (a correct password will change the global variable *pin* from 0 to 1). At the end the service changes the global variable *log* from 0 to 1 if the login is successful.

The *BALANCE DISPLAY* service, depicted in Figure 5.3, is modelled as a REMES atomic mode. It is responsible for displaying the current account balance. This service can be executed only if the login was successful ( $log == 1$ ). The displayed account balance is stored in a local integer variable named *acc.balance*.

The *MONEY WITHDRAW* service, illustrated as a REMES composite mode in Figure 5.4, can be accessed only after a correct login (i.e.,  $log == 1$ ). The mode consists of two atomic modes: *TYPE SUM* (it waits for the customer to type the amount of money he/she wants to withdraw and stores it to a local variable named *sum*) and *GIVE SUM* (responsible for actual money withdrawal). The latter service can be executed if the customer has enough money in his/her account. As depicted in Figure 5.4, we use a conditional connector in order to model the availability of funds in the customer account.

The *ERROR* service is described as a REMES atomic service in Figure 5.5. It was

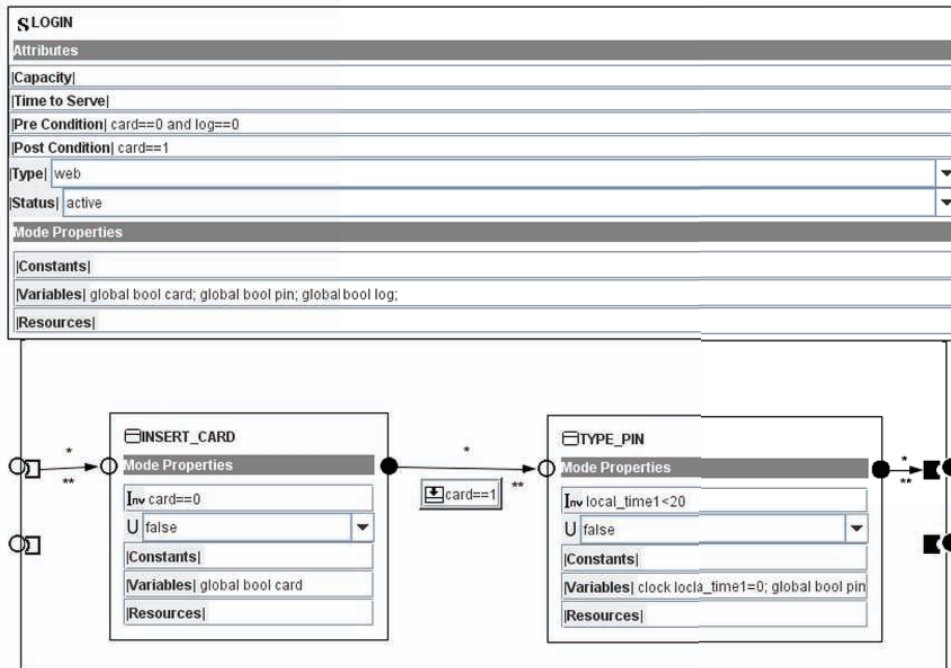


Figure 5.2: Login service modelled as a REMES composite service

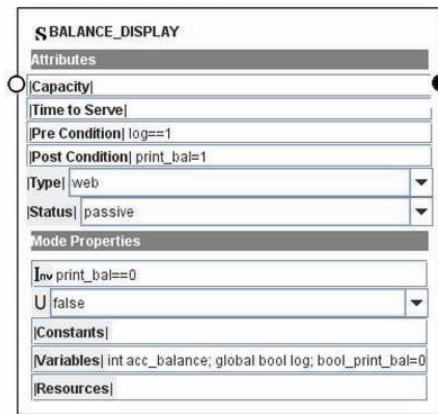


Figure 5.3: Balance service modelled as a REMES atomic service

designed to deal with an incorrect behaviour related to the customer login. It is an *urgent* mode for processing a login mistake instantaneously after its activation.

The *LOGOUT* service, depicted in Figure 5.6, is modelled as a REMES atomic mode that is responsible for ending the connection with the customer by ejecting the card.

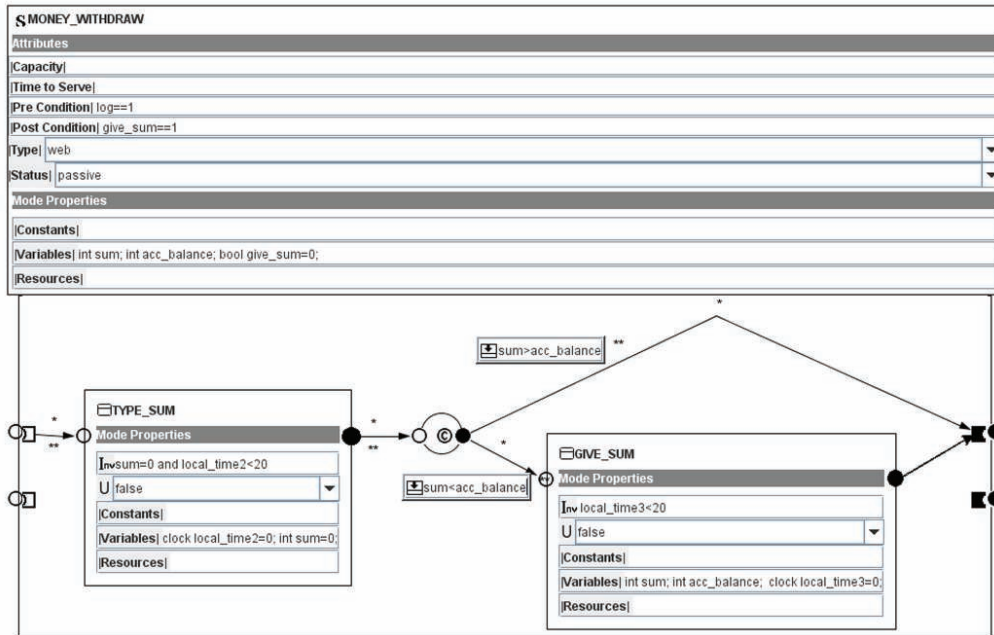


Figure 5.4: Money withdraw service modelled as a REMES composite service

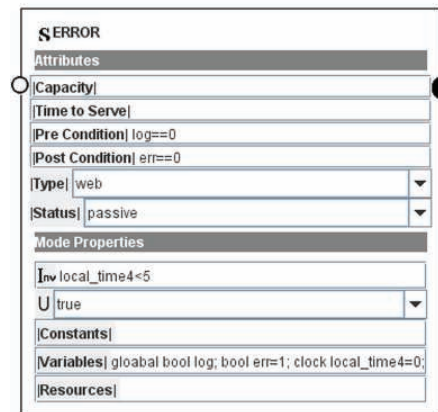


Figure 5.5: Error service modelled as a REMES atomic service

### 5.3 A service composition

The service user can invoke and compose the available services in different ways depending on the his/her preferences. For example, as illustrated in Figure 5.8, one can compose a system using our design framework in the following way: *LOGIN* service is executing first, in a sequence with a parallel composition and the *LOGOUT* service. The parallel composition is used to describe the execution of *BALANCE DISPLAY* || *MONEY WITHDRAW* || *ERROR*.

LOGOUT	
Attributes	
Capacity	
Time to Serve	
Pre Condition	card==1
Post Condition	card==0 and pin==0 and log==0
Type	web
Status	passive
Mode Properties	
In	local_time5<20
U	true
Constants	
Variables	global bool card; global bool pin; global bool log; clock local_time5=0
Resources	

Figure 5.6: Logout service modelled as a REMES atomic service

```

Console View
Console  Declarations  Diagram
declare LOGOUT::=< web,,passive,(card==1),(card==0 and pin==0 and log==0)>;
declare ERROR::=< web,,passive,(log==0),(err==0)>;
declare BALANCE_DISPLAY::=< web,,passive,(log==1),(print_bal=1)>;
declare LOGIN::=< web,,active,(card==0 and log==0),(card==1)>;
declare MONEY_WITHDRAW::=< web,,passive,(log==1),(give_sum==1)>;
create LOGOUT;
create ERROR;
create BALANCE_DISPLAY;
create LOGIN;
create MONEY_WITHDRAW;
check(sp.(LOGIN;(MONEY_WITHDRAW|BALANCE_DISPLAY|ERROR);LOGOUT).(card==0 and log==0)=>REQ)

```

Figure 5.7: HDCL and the generated check command for the ATM system

We illustrate in Figure 5.7 the use of the textual modelling service composition for the visual service composition depicted in Figure 5.8. The services are introduced through the declarative part and the selected services are created. Moreover, we provide an automatic *check* command for showing if the given requirement is satisfied or not.

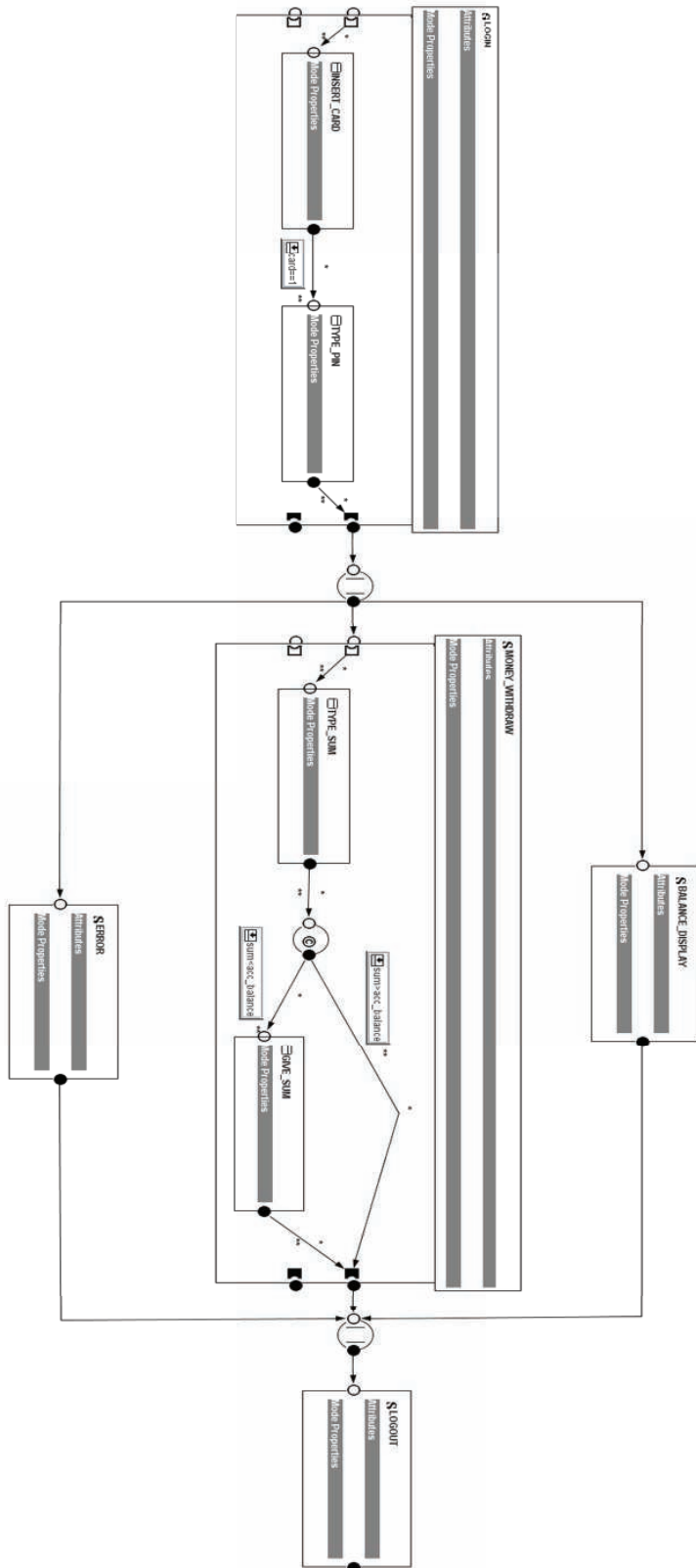


Figure 5.8: A service composition for the ATM system modelled in the design framework

## Chapter 6

# Further Ideas

During the implementation of the tool, we have discovered some additional possibilities for a future improvement and extension. The primary reason for not implementing them in this work is that their implementation has been recognized as a lower priority at this point, and too time consuming.

The aim of this thesis is to have a complete tool-based support of the REMES extension for SOS modelling process, rather than missing some functionalities. The tool, in this stage, provides powerful support for the creation of REMES models in service-oriented context. Consequently, future additions will complement the automatic functionalities of the tool and reduce manual tasks. Upgrades could enlarge either the functionality of the GUI, or provide some extensions of the display and data model.

In order to improve the model, one of the possibilities is to organize the diagram scene. Also, one could bring a huge benefit by grouping the widgets. The content of the scene could be even more structured by minimizing different widgets to a single icon.

One could consider to implement a dialog window to process the project creation. With the help of a wizard extra information necessary for the system model would be collected. It has to be noted that this functionality would work if changes on the underlying id's of all model elements is done when model copies are temporary stored.

Also, the appearance of the models could be improved by having an auto-layout feature. In this way the entities on the scene would be arranged in the optimal way. The connections between services and modes could also be improved by giving the user a choice for different edge routing algorithms, e.g. orthogonal router.

The modelling process could be improved, with implementation of copy and paste functionality. This feature will reduce the time needed to create a diagram as redundant work would be avoided.

Described features could serve as an inspiration for further development of the tool. As the tool is a fast-growing software whose code is changing very often, this recommendation refers to the current version.



## Chapter 7

# Conclusions

Many approaches that treat SOS, are only concerned with service interface description. Some of them consider the specification of services, their inner behaviour, and hierarchical composition. Inner behaviour-related information, i.e., actions taken in order for a service to be executed, resource annotations, etc., is provided to enable the developer to understand the inner functionality of the system, to interconnect services in a correct manner, and provide a better analysis. While specifying, modelling and reasoning about SOS, it could be beneficial to include both service interface description, but also information provided within the inner service behaviour.

In this thesis we have presented a design framework that contains a graphical user interface for behaviour modelling of services based on the resource-aware timed behavioural language REMES [28]. The development of a graphical representation of services and their composition is driven by the need of modelling and simulation in an environment where services can interact on demand. In this work, Visual Library API is used to display editable service diagrams with support for graph-oriented modelling. By using the commonly used DOM XML to manipulate the diagram structure, it has been possible to separate the data structure from the modelling tool. In this way, service composition scenarios can be saved and reused by both designers and analysts.

We have implemented a textual dynamic service composition interface, together with means to automatically verify service composition correctness to enable automatic service verification. We also provide an automated traceability between the two design interfaces resulting in an efficient tool that enhances the potential of SOS design by focusing on intuitive service manipulation.

The design framework presented in this thesis, serves the purpose of providing a solid, modularized base for further development. Nevertheless, it can already be used for educational purposes in SOS, as a modelling and documentation tool.





# References

- [1] Gaudenz Alde. The jgraph swing component. 2001.
- [2] G. Alder. Design and implementation of the jgraph swing component. *Technical Report*, 1(6).
- [3] J. Amsden. Modeling soa, parts i–v. *IBM developerWorks (October 2007)*.
- [4] A. Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 2004.
- [5] D. Auber. Tulip—a huge graph visualization framework, 2003.
- [6] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced time automata. *Hybrid Systems: Computation and Control*, pages 147–161, 2001.
- [7] Heiko Boeck and J. Tulach. *The Definitive Guide to NetBeans Platform*. Springer, 2009.
- [8] M. Broy. Service-oriented systems engineering: Modeling services and layered architectures. *Formal Techniques for Networked and Distributed Systems-FORTE 2003*, pages 48–61, 2003.
- [9] F. Budinsky. *Eclipse modeling framework: a developer’s guide*. Prentice Hall Ptr, 2004.
- [10] A. Čaušević, C. Seceleanu, and P. Pettersson. Modeling and reasoning about service behaviors and their compositions. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 82–96, 2010.
- [11] A. Caušević and A. Vulgarakis. Towards a unified behavioral model for component-based and service-oriented systems. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pages 497–503. IEEE, 2009.
- [12] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [13] R. Eckstein, M. Loy, and D. Wood. *Java swing*. O’Reilly & Associates, Inc., 1998.
- [14] GEF Eclipse. Eclipse graphical editing framework, 2008.
- [15] J.D. Fekete. The infovis toolkit. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 167–174. IEEE, 2004.
- [16] N. Gold, A. Mohan, C. Knight, and M. Munro. Understanding service-oriented software. *Software, IEEE*, 21(2):71–77, 2004.

- [17] E.R. Harold and W.S. Means. *XML in a Nutshell*. O'Reilly Media, Inc., 2004.
- [18] M. Harren, M. Raghavachari, O. Shmueli, M.G. Burke, V. Sarkar, and R. Bordawekar. Xj: Integration of xml processing into java. In *Proceedings of the 13th International World Wide Web conference on Alternate track papers & posters*, pages 340–341. ACM, 2004.
- [19] J. Heer, S.K. Card, and J.A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM, 2005.
- [20] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd conference on Visualization'91*, pages 284–291. IEEE Computer Society Press, 1991.
- [21] K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto. Automated support for modeling oo software. *Software, IEEE*, 15(1):87–94, 1998.
- [22] A. Moreira and R. Clark. Combining object-oriented analysis and formal description techniques. *Object-Oriented Programming*, pages 344–364, 1994.
- [23] E. Nasr, J. McDermid, and G. Bernat. Eliciting and specifying requirements with use cases for embedded systems. In *Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on*, pages 350–357. IEEE, 2002.
- [24] M.P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.
- [25] NetBeans Platform. Visual library 2.0 - documentation, June 2010.
- [26] NetBeans Platform. Visual library api javadoc, 2011.
- [27] A. Rajeev and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [28] C. Seceleanu, A. Vulgarakis, and P. Pettersson. Remes: A resource model for embedded systems. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 84–94. IEEE, 2009.
- [29] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. *Component-Based Software Engineering*, pages 310–317, 2008.
- [30] A. Vulgarakis, S. Sentilles, J. Carlson, and C. Seceleanu. Integrating behavioral descriptions into a component model for embedded systems. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 113–118. IEEE, 2010.
- [31] Y. Wang and Y. Zhang. The formal design model of an automatic teller machine (atm). *International Journal of Software Science and Computational Intelligence (IJSSCI)*, 2(1):102–131, 2010.
- [32] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles: Visualization and automatic layout of graphs. In *Graph drawing: 9th international symposium, GD 2001, Vienna, Austria, September 23-26, 2001: revised papers*, page 453. Springer Verlag, 2002.