Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science (120 credits) in Computer Science with Specialization in Software Engineering — 30.0 hp — DVA501

MAPPING UML DIAGRAMS TO THE REACTIVE OBJECT LANGUAGE (REBECA)

Vladimir Djukanovic
vdc16001@student.mdh.se

Examiner: Marjan Sirjani
Mälardalen University, Västerås, Sweden

Supervisor: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

June 13, 2019

Abstract

Unified Modeling Language (UML) is a de-facto standard modeling language with an extensive syntax and notations that can be used to model systems of any kind. However, being a general-purpose language, its semantics are intrinsically under-specified and broad to leave a room for different interpretations. This, in general, hinders the ability to perform formal verification of models produced with a specific domain in mind. In these cases, it is usually more suitable to map the UML models to other domains, where modeling concepts have stricter semantics. Notably, Reactive Objects Language (Rebeca) is an actor-based language with a formal foundation and formal verification support. This paper aims to identify a subset of UML modeling concepts compatible with the domain of reactive and distributed systems as modeled in Rebeca. In this respect, this work proposes a conceptual mapping between a sub-portion of UML and Rebeca, with the goal of enabling formal verification early in the design process. In particular, we investigate Rebeca syntax, and for each Rebeca concept, we provide the corresponding concept in the UML, as part of an iterative process. This process ends when all Rebeca concepts are exhausted and comprehensive mapping procedure emerges. Additionally, validation is an important part of this paper as it aims to establish confidence in the developed mapping procedure (in post-conversion validation) and avoid doing the transformation if the design is not compatible with the mapping procedure (in pre-conversion validation). As part of the pre-conversion validation, in order to establish the compatibility with the mapping procedure, we provide an extensive list of correctness attributes. As part of the post-conversion validation, the mapping procedure is validated by transformation on the provided examples. The results of this transformation show the wide range applicability of the mapping procedure and serve as an assertion of its comprehensiveness.

Table of Contents

List of Figures 4

List of Tables 5

Glossary 6

1 Introduction 7
1.1 Goal and Problem Statement 8
1.2 Thesis Outline 9

2 Background 9
2.1 Unified Modeling Language (UML) 11
2.2 UML Diagrams 12
2.2.1 Structural Diagrams 13
2.2.2 Behavioral Diagrams 15
2.3 Reactive Objects Language (Rebeca) 17
2.3.1 Rebeca - Actor-based Modeling 18
2.3.2 Rebeca Syntax and Semantics 18

3 Related Work 21

4 Research Process 23

5 A Mapping Procedure for Transformation of UML Models to Rebeca Models 26
5.1 Requirements Analysis 26
5.2 Identification of Rebeca Concepts 27
5.2.1 Extraction and Analysis of Rebeca Concepts 27
5.3 Identification of Corresponding UML Concepts - Iterative Mapping . 30
5.3.1 Structural UML Concepts 31
5.3.2 Behavioral UML Concepts 36
5.4 Detailed Mapping Procedure Description and Overview 47

6 Method Evaluation 54
6.1 Pre-Conversion Validation 54
6.2 Post-Conversion Validation 58
6.2.1 Practical Example - Validated Source Models 59
6.2.2 Manual Transformation - Acquiring Target Models 62
6.2.3 Capturing Rich Behavioral Concepts - Example 67
6.2.4 Results, Applicability and Potential Improvements of Mapping Procedure 73

7 Discussion and Limitations 78

8 Conclusion and Future Work 80

References **81**

List of Figures

2.1 Class diagram example . . . 14
2.2 Object diagram example . . . 15
2.3 Sequence diagram example . . . 16
2.4 State-machine diagram example . . . 17
2.5 Rebeca syntax [1] . . . 20
4.1 Research process cycle . . . 24
4.2 Mapping procedure - iterative creation process . . . 24
4.3 Validation process . . . 25
5.1 Combined Fragment of type ALT . . . 38
5.2 Combined Fragment of type ALT - with logical operator . . . 38
5.3 Combined Fragment of type ALT - with conditional comparison . . . 39
5.4 Initiating message call . . . 40
5.5 Asynchronous message call to another class . . . 40
5.6 Asynchronous message call to self . . . 41
5.7 Combined Fragment of type LOOP - excluding range . . . 42
5.8 Combined Fragment of type LOOP with the inclusion of range . . . 43
5.9 Combined Fragment of type LOOP with the inclusion of range and iteration variable . . . 43
5.10 Setter global method call . . . 44
5.11 Regular inline set . . . 45
5.12 Array position inline set . . . 45
5.13 Inline set by overriding default assignment operator . . . 46
5.14 Regular local variable declaration . . . 47
5.15 Regular local variable declaration with initialization . . . 47
5.16 Local array declaration . . . 47
6.1 Class diagram - structural source . . . 59
6.2 Object diagram - structural source . . . 59
6.3 Sequence diagram - Train constructor - behavioral source . . . 60
6.4 Sequence diagram - Train youMayPass method - behavioral source . . . 60
6.5 Sequence diagram - Train passed method - behavioral source . . . 60
6.6 Sequence diagram - Train reachBridge method - behavioral source . . . 61
6.7 Sequence diagram - BridgeController leave method - behavioral source 61
6.8 Sequence diagram - BridgeController arrive method - behavioral source 62
6.9 Class diagram - Node class - structure . . . 68
6.10 Object diagram - Node instances - structure . . . 68
6.11 Sequence diagram - Node constructor - behavior . . . 68
6.12 Sequence diagram - Start global transaction method - behavior . . . 69
6.13 Sequence diagram - Cooperator response method - behavior . . . 70
6.14 Sequence diagram - Create transaction method - behavior . . . 71

List of Tables

5.1 Comprehensive mapping procedure - textual representation 49
5.2 Comprehensive mapping procedure - detailed conceptual representation 51
5.3 Comprehensive mapping procedure - Rebeca-centric concepts 52
6.1 Pre-conversion validation correctness rules 57

Glossary

UML Unified Modeling Language

MDE Model-Driven Engineering

OCL Object Constraint Language

Rebeca Reactive Objects Language

rebecs Reactive Objects

RMC Rebeca Model Checker

MOF Meta-Object Facility

OMG Object Management Group

AMN Abstract Machine Notation

OOP Object-Oriented Programming

FCFS First Come First Served

1 Introduction

The exponential growth of software complexity with a notable focus on safety-critical applications [2] introduced the demand for a new approach to development. As a positive reaction to this, the UML was created, that now represents the standardized notation for modeling and documenting software systems [3]. UML initiated the creation of a new approach to design, called Model-Driven Engineering (MDE) that focuses on the use of models as the main building blocks of the system [4]. The most anticipated area of MDE is model transformation which, in its subset, enables a code to be automatically generated from models [4]. Hence, the model transformation facilitates a transition of the models towards other domains.

Despite its general use and acceptance as a de-facto standard modeling language, the UML has a serious drawback [5]. This comes from the fact its semantics are intrinsically under-specified and broad to leave a room for different interpretations. Hence, multiple and potentially contradictory interpretations of one and the same model are not excluded, and automatic interpretation must be hard-coded in some way [6]. This, in general, hinders the ability to perform formal verification of models produced with a specific domain in mind. This gap in the UML creates a demand, for formal verification to establish the correctness of the models early in the process. In these cases, it is usually more suitable to map the UML models to other domains, where modeling concepts have stricter semantics.

That is where Rebeca comes into the picture. Rebeca is an actor-based language with a formal foundation. Rebeca is an easy to use JAVA alike language and a modeling language, with formal semantics and formal verification support [1].

To address the shortcomings of UML, it would be interesting to provide a detailed conceptual mapping between a sub-portion of UML and Rebeca. In other words, we want to identify a subset of UML modeling concepts compatible with the domain of reactive and distributed systems as modeled in Rebeca. This should ultimately lead to implementation of the model transformation for such mapping.

We also want to have an appropriate validation phase with different reasons:

First, after the creation of the mapping procedure, the necessity for its validation on applicability scenarios is obvious. This could be performed differently depending on different factors including research limitations (i.e. time). In an optimistic scenario, we would like to provide a model transformation tool, in which case the automated tool would be running the transformation and afterward validation on different applicability scenarios with the purpose to establish the correctness of the target models (runnable in Rebeca and reflecting the source UML models). In a more realistic scenario, the validation could be performed in cooperation with external subjects where they would do the transformation from source UML models to the target Rebeca models by manual transformation, using the mapping procedure.

Second, by enabling a validation of the source UML models we want to avoid performing the transformation if the design is not compatible with the mapping procedure or in other words is outside the domain. In fact, in that case, the costs of fixing design defects would be far more relevant than anticipating problems in obtained Rebeca models.

1.1 Goal and Problem Statement

The past decades have witnessed significant efforts towards simplifying a process of developing a system, reducing system complexity and conducting a formal verification of the developed system. The creation of the UML and its confirmation as a de-facto standard modeling language answered on the first problem, to some extent [3]. Moreover, it shows a serious potential to reduce the complexity of the developed system and therefore provide more easily maintainable software products. However, UML has a serious shortcoming as it lacks complete formal semantics and this hinders the ability to perform formal verification of models produced with a specific domain in mind. On the other side, Rebeca language has formal semantics and provides support for formal verification that enables more accurate evaluations to be performed [5]. In this respect, it would be beneficial to establish a conceptual mapping between a sub-portion of UML and Rebeca with the goal of enabling formal verification early in the design process.
The main objective of this thesis is to investigate the viable ways of mapping UML models towards Rebeca models. The mapping should be detailed enough while focusing on the minimalist diagrammatic approach in terms of what is the minimum UML information (including both the set of diagrams and information contained in the diagrams) that is needed for target Rebeca concepts. Following the aforementioned, it is logical and expected that there exists a need for the behavioral diagrams to obtain meaningful information in Rebeca to be analyzed and verified by a formal verification tool. Moreover, there could be a need for structural diagrams, in order to model the structure before introducing behavior, and we consider them as well for the mapping. The overall objective of the thesis is to conduct an analysis to identify the minimum UML diagrams that will be used as sources for formal verification in Rebeca. Additionally, we want to provide a detailed mapping procedure to translate identified UML concepts towards fitting Rebeca concepts with consideration of important factors as available resources for conducting the thesis. Hence, we strive to provide an achievable detailed enough mapping procedure that is in accordance with the scope of this research. The mapping procedure should be detailed enough to enable the implementation of a model transformation tool. The model transformation tool shall perform the automatic translation of the UML models to Rebeca models within the established applicability scope of the mapping procedure. In the end, the thesis aims to provide a proper validation of the proposed mapping procedure in order to show the applicability on different scenarios.
The identified problems can be written in the form of research questions as follows:

1. **RQ1:** What are the minimum UML diagrams required to serve as sources for target Rebeca concepts in order to obtain something meaningful to be analyzed through Rebeca?

2. **RQ2:** What is the adequate mapping procedure between the identified UML concepts and Rebeca concepts?

3. **RQ3:** What is the applicability of the proposed mapping procedure and its proper substantiation?

1.2 Thesis Outline

In Section 2, we explain what are the main development and life-cycle issues with evolving software solutions. Directly related to this, we explain various concepts among which is UML that is attempting to address this issue. However, due to its shortcomings, it is required to go even further into the subject by introducing Rebeca language and discussing how we can reap the benefits of a mapping between Rebeca and UML, for modeling the domain of reactive and distributed systems. This leads us to the main objectives of this research by conclusively proposing the mapping procedure and the validation process. In Section 3, we discuss related work and the contribution that this research is attempting to achieve. In Section 4, we are reasoning about a research methodology that is used in this thesis, and we propose our research goals. In Section 5 we are going deeper into the subject by executing the proposed iterative process until we reach a satisfactory level of comprehensiveness of the emerged mapping procedure. Initially, this includes a specification of all the Rebeca concepts. Then, for each of them, we attempt to provide minimum information in the UML that is necessary for an adequate mapping to be accomplished between the two. Indeed, if any limitations are found we document these in a separate subsection and we attempt to reason about them and provide a method for addressing each of them. In section 6, we describe the emerged mapping procedure on two examples constructed of the UML models and corresponding Rebeca models that are produced by transformation, based on the proposed mapping procedure. Initially, before the translation to Rebeca is performed, we provide an extensive list of correctness attributes as part of the pre-conversion validation, to avoid performing the transformation and reaching faulty Rebeca models if the design is not compatible with the mapping procedure. After the transformation on pre-validated UML models is done, we proceed by presenting the results of our research and evaluating this process regarding applicability. This indeed represents the post-conversion validation phase, in particular, reasoning about the applicability of the mapping procedure with the inclusion of the limitation analysis. Finally, in Section 8, we conclude the work with a brief summary and possible future research directions.

2 Background

The common traditional techniques for software development including in its core significant programming efforts are failing to meet the newly introduced demands on the software market [3]. This situation, that caught everyone off guard, was caused by the uncontrolled expansion of software throughout all domains including safety-critical domain in which the consequences of failure could be catastrophic. The software is becoming widespread and complex [2]. This implies it is harder to build and maintain the software that causes an increase in the time and costs of these activities. This also leads to lower quality of software and inability to properly establish its correctness [2].
It is clear that there exists a need for simplifying the development process as well as reducing the time and cost for development and maintenance of software while

improving its reliability (continuity of correct service).
As an attempt to give an answer to these questions, different concepts, methodologies, and processes have been proposed during the past decade corresponding to different sides of the problem.
These solutions include methodologies for simplifying development and maintenance processes, reducing software complexity and providing new verification techniques such as model checking, theorem proving, etc. as means of formal verification with the inclusion of their automation.

On one side, we have a UML that is focused on reducing software complexity and programming efforts. UML represents the standardized notation for modeling and documenting software systems. UML initiated the creation of a new approach to the development process, called MDE [4]. We will explore a model transformation that is one of the most important concepts in MDE, that we want to use for transforming the UML models towards other domains.
On the other side, the formal verification techniques that were introduced are focused on proving that a software conforms to a formal specification for its intended behavior. Model checking is one of the most important formal verification techniques. It exhaustively and automatically checks whether the observed model meets its given specification. However, it is important to stress that not all languages are supported by a model checker in which case it is necessary to perform a transformation to another language that is supported by a model checker.
Also, it depends on what kind of systems are we verifying. Currently, we are experiencing a shift towards parallel systems that enable concurrent execution of the programs. This change is mainly due to physical limitations of processing units, that emerged, and caused a shift towards multicore processors (single integrated circuit consisting of multiple core processing units capable of concurrently executing different tasks) [7]. As it wasn't possible to keep up with making single core processors any faster, multicore processors were introduced in hope that they will result in increased processing speed.
However, these processors require software solutions to be accommodated for parallel execution. In other words, if we want to take advantage of the hardware we have available, we need a way to run our code concurrently.
One of the first approaches for parallel execution was established with the use of Threads [8]. However, Threads proved to be extremely complex and volatile that made them impossible to control, with additional danger of deadlock occurrences (a condition in which each member of a group is waiting for another member, including itself, to perform an action, i.e. sending a message or more commonly releasing a lock) [8].
A counter-approach [8] that solves the shortcomings of Threads is actor-based modeling.
The actor model is a conceptual model to deal with concurrent computation. It defines some general rules for how the system's components should behave and interact with each other [9]. We will discuss in detail about actor-based modeling and its constructing elements in Section 2.3.1.
As aforementioned, Rebeca is the actor-based language with a formal foundation,

that is created in an effort to bridge the gap between formal verification approaches and real applications [1].
Regarding the objectives of this thesis and how it relates to all this, by identifying a subset of UML compatible with the domain of reactive and distributed systems as modeled in Rebeca, we strive to enable formal verification early in the design process.

In the following sections, we will provide a detailed definition of UML and Rebeca and all concepts related to them that will be covered in this thesis.

2.1 Unified Modeling Language (UML)

UML represents a standardized notation for modeling and documenting software systems. UML is under constant evolution by the Object Management Group (OMG) group [3]. Moreover, UML has become a general-purpose language, that means any system can be modeled by using its concepts. The use of UML ultimately leads towards reduction of the complexity of both the development process and the developed system. Besides, having one standard language for modeling has many advantages to software development, such as simplified training and unified communication between development teams [5].
Furthermore, UML initiated the creation of a new approach to the development process, called MDE [4]. MDE is a software engineering branch that advocates the use of models as the main building blocks of the system. Models represent high-level abstractions of the observed real phenomena which focuses on the main segments while hiding the unnecessary information from the modeler of the system. This contributes to the reduction of the complexity of both the development process and the developed system. UML diagrams are used for a visual representation of the system based on different aspects and views, in order to provide a better understanding, maintenance, alterations, and documentation of the system.
The most anticipated area of MDE is model transformation [4] which, in its subset, includes automatic code generation from input models and aim to reduce the programming efforts and therefore human-error that is part of the programming process.
Despite its general use and acceptance as a de-facto standard modeling language, the UML has some serious drawbacks [5]. It is complex and tools can support only a part of its entire capability. Moreover, the expectation that training will become easier with UML being a standard, unified language, did not produce expected results and was characterized as over-promised. Also, UML still lacks complete formal semantics even though many achievements have been made through extensive research towards a common goal to define semi-formal semantics (with Object Constraint Language (OCL)) as well as attempts towards formalizing UML semantics [5, 10–13].
This has a negative effect on the UML and the many expectations related to it. These effects and risks associated with the use of UML are amplified if we take into consideration safety-critical domain in which a failure could have catastrophic consequences and therefore these systems cannot be left unverified under any circumstances. Furthermore, in such cases where the criticality of the systems is accentuated, it is required to model systems in formal languages to be able to use

formal verification techniques, early in the development process.
One of the advantages of UML that helps us deal with these drawbacks is the possibility for UML customization that is established by definition of UML profiles [14,15]. A profile is a subset of UML syntax with the inclusion of a set of well-defined rules. The provision of these rules add standard (known) UML elements to the subset and define some additional semantics in a natural language that ultimately leads to the creation of a hybrid UML language accommodated with the semantics of the target language [5, 14, 15]. However, UML profiles are significantly limiting the true potential of UML by creating a domain-specific UML language that is a hybrid between the source UML and other domain-specific languages used as a target. This approach restricts the use of the new hybrid UML to only that domain (language). With respect to this paper, we strive to keep the UML in its native form without introducing a UML profile in order to avoid producing a restricted hybrid UML language. Naturally, we need to set some rules that will be used as correctness attributes for validating source UML models before conversion to Rebeca takes place and establishing that the models are indeed in the Rebeca domain. These correctness attributes essentially gives us an early answer about the compatibility of source UML models with the mapping procedure.

2.2 UML Diagrams

UML is based on diagrammatic reasoning. It can be described with the proverb: a picture is worth a thousand words. Models represent high-level abstractions of the observed real phenomena which focuses on the main segments while hiding the unnecessary information from the modeler of the system. UML diagrams are used for a visual representation of the system based on different aspects and views that define the information that is modeled within them [15, 16].
By using visual representations, we are able to better understand possible flaws in the software early in the process and provide easier development, modification and documentation processes. UML diagrams can be defined as blueprints of software, where each type of diagram is used for modeling a certain aspect of the software whether it is its structure or behavior.
UML is not a stand-alone programming language like Java, C#, etc.. However, with the use of certain automation tools for model transformations, it can be used as a pseudo-programming language [16]. In such case, the whole system needs to be modeled with the use of various UML diagrams and can be directly translated to code with the use of specialized tools for automating model transformation between UML and the desired programming language.
The broadest two categories of UML diagrams which encompass all other categories are:

1. Structural Diagrams (Section 2.2.1)
2. Behavioral Diagrams (Section 2.2.2)

2.2.1 Structural Diagrams

Structural UML diagrams analyze and depict the structure of the software or process. We provide the list of available structural diagrams:

1. Class Diagram
2. Object Diagram
3. Component Diagram
4. Composite Structure Diagram
5. Deployment Diagram
6. Package Diagram
7. Profile Diagram

We will only elaborate on those diagrams that are of interest for this thesis. These diagrams are the ones that are most commonly used to represent the structure of the software and provide input information for model transformation. These diagrams include Class Diagram and Object Diagram.

Class Diagram is the most common structural diagram for designing or documenting software. Considering the current trend and common use of Object-Oriented Programming (OOP) paradigm that covers this thesis as well, using this type of UML structural diagram is completely reasonable solution [16]. It is accommodated with the main building blocks used in OOP such as *Classes* that consist of *Attributes* also known as data fields, and *functions* also known as behaviors.

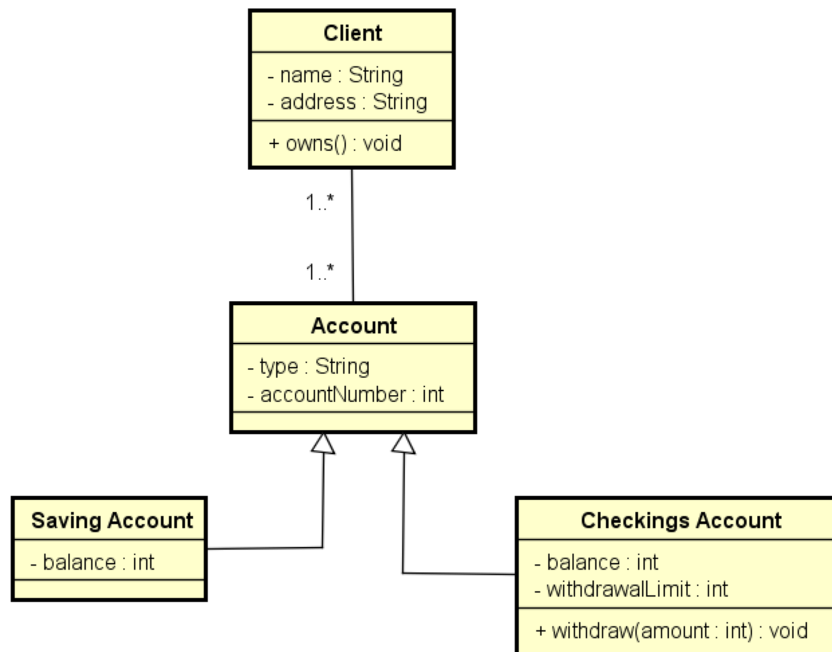

Figure 2.1: Class diagram example

As can be seen in Figure 2.1, a basic example of the class diagram is provided. We notice that both classes: *Checkings Account* and *Saving Account* are generalizations (represented by a blank headed arrow) of the class *Account*, that means they are the children, in the inheritance tree, of the class *Account*. This represents the same type of inheritance as in any OOP language. Besides, the diagram is quite self-explanatory and it clearly states the consisting classes and how they are inter-related between each other.

Object Diagram is another type of structural diagram that we are going to consider in this thesis. In order to define this type, we need to look at OOP paradigm once again. We know that classes are used as blueprints upon which objects are built by use of instantiation mechanism. A class can have objects instantiated from other classes within. To represent any kind of instantiation of a class and keep track of all objects we use object diagrams [16]. A clear example can be seen in Figure 2.2 that represents the instantiated objects from the provided class diagram in Figure 2.1.

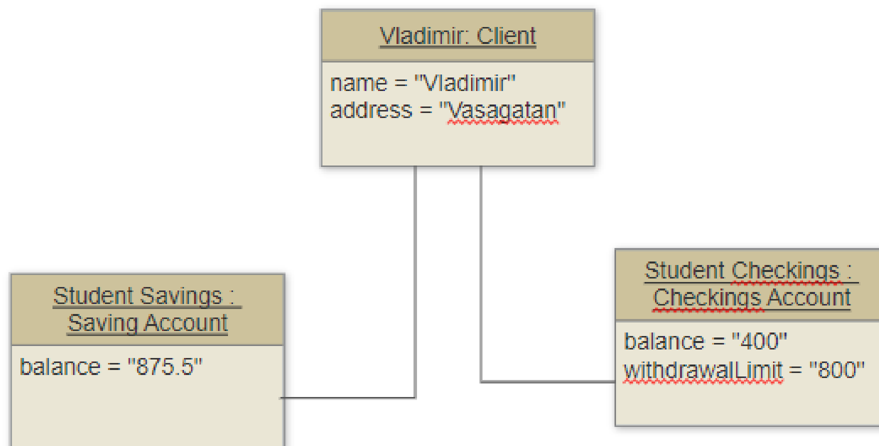

Figure 2.2: Object diagram example

2.2.2 Behavioral Diagrams

Behavioral UML diagrams analyze and depict functional behavior of the software, process and its building components [15]. We provide the available list of behavioral diagrams:

1. Activity Diagram
2. Use Case Diagram
3. Interaction Overview Diagram
4. Timing Diagram
5. State Machine Diagram
6. Communication Diagram
7. Sequence Diagram

We will only elaborate on those diagrams that are of interest for this thesis. These diagrams are the ones that are most commonly used to represent the behavior of the software and its components and provide input information necessary for model transformation. These diagrams include a Sequence diagram and State Machine Diagram.

Sequence diagram is the most commonly used diagram for defining software behavior. This includes describing the behavior of several objects within a single use case that is its primary use. Also, based on its name, we can describe it as a sequence of messages that are exchanged between the objects [16]. These interactions are represented in a chronological manner as can be seen in Figure 2.3, that establish behavioral layer for the provided class diagram in Figure 2.1.

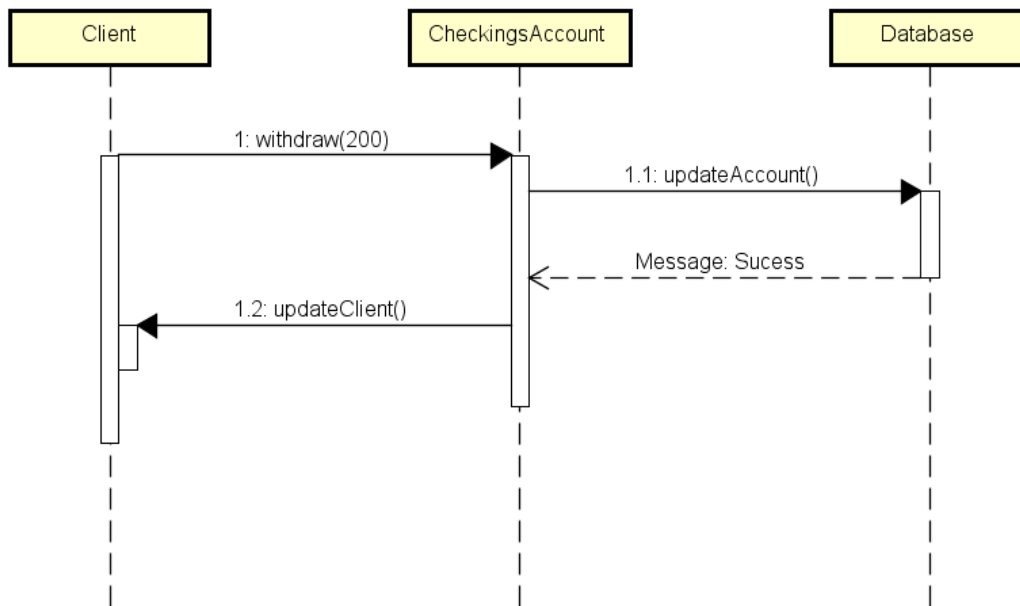

Figure 2.3: Sequence diagram example

Each object or actor has a lifeline that goes towards the bottom and each lifeline has activation bars used to represent when the object is in the active state (performing some action etc.).

State-machine diagram also known as a statechart is a behavioral type of UML diagram used to represent how a single object behaves within multiple use cases. This basically refers to all the states that an object can have within a system or how different internal and external events contribute to the state change. These diagrams are used for reverse and forward system engineering [16]. An example of the state machine diagram is given in Figure 2.4.

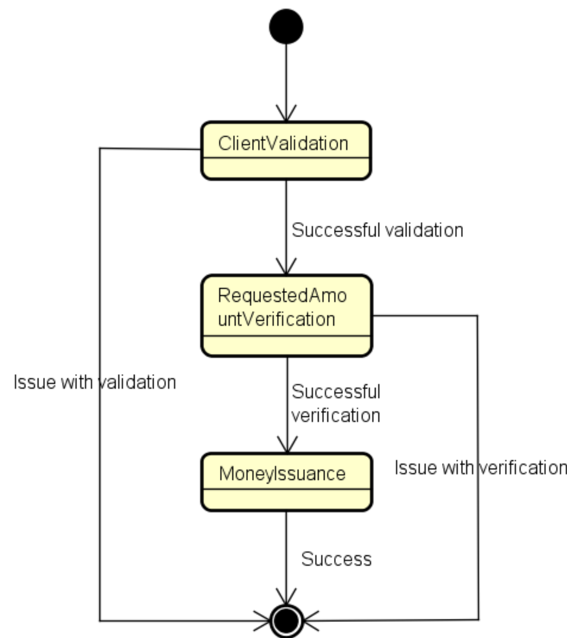

Figure 2.4: State-machine diagram example

2.3 Reactive Objects Language (Rebeca)

Rebeca is an actor-based language with a formal foundation and is created with reason to overcome an existing gap between formal verification approaches and real-world applications. Rebeca is an easy to use JAVA alike language and a modeling language, with formal semantics for models (encompassing their states, state transitions, communications and provision of accessible interfaces), and formal verification support [1].

Rebeca combines certain concepts from actor-based modeling with certain concepts from object-based modeling. This indeed implies that Rebeca is used for developing object-based distributed systems that can be formally verified by using a model checking tool. Notably, Rebeca uses abstraction techniques to reduce the state space of the model in order to make it more appropriate for model checking.

Model checking in Rebeca can be done directly with the use of Rebeca Model Checker (RMC) which is a tool for direct model checking of Rebeca models [17]. Besides, when it is of interest Rebeca can be translated to other model-checker languages among which *Promela* is the most popular for its model-checker *SPIN* that is arguably one of the world's most powerful tools for detecting software defects in distributed systems.

In Rebeca [1], computations are established by passing asynchronous messages between Reactive Objects (rebecs) and execution of the corresponding message servers of passed messages. Each message is placed in the queue of the receiving rebec (responsible for handling upcoming messages) and it specifies the method to be called when the message is finally serviced.

More about Rebeca, its semantics and its relation to the Actor-based modeling is provided in the following two sections.

2.3.1 Rebeca - Actor-based Modeling

When we elaborate on Rebeca we find it important to make a comparison with actor-based.
Rebeca model is to some extent similar to the pure actor model based on two of its main concepts:

1. Asynchronous message passing
2. Use of independent active objects

These independent active objects are reactive and self-contained [5]. They are called rebecs that stands for reactive objects. In Rebeca, computations are established by passing asynchronous messages between rebecs and execution of the corresponding message servers of passed messages. Each message is placed in the queue of the receiving rebec (responsible for handling upcoming messages) and it specifies the method to be called when the message is finally serviced [1].
The queue of each rebec is a buffer used for storing messages in an order in which they will be executed so that the first message that arrives is served first that is based on the First Come First Served (FCFS) scheduling approach. The queue length represents a maximum number of messages that can be stored in a queue and it is defined in the reactive class definition (that is indicated next to the reactive class name surrounded by parenthesis).
As aforementioned, we use FCFS scheduling of the messages inside the queue. Notably, when the message at the top of the queue of the reactive object is serviced then the defined method, in the message, is called, which triggers the removal of the message from the queue [1, 17].

2.3.2 Rebeca Syntax and Semantics

As can been seen in Figure 2.5, a list with Rebeca syntax is presented. We are going to discuss briefly the most important concepts among them.
As we already mentioned in the previous sections, each reactive object (rebec) is instantiated from the corresponding reactive class (*reactiveclass*), and it has a single thread of execution [5, 17]. In Rebeca, we have a set of reactive classes and a *main* part. In the *main*, rebecs are instantiated from the reactive classes. Furthermore, each of the reactive classes contains known objects (*knownrebecs*), message servers (*msgsrv*) and state variables (*statevars*) [1]. Known objects (*knownrebecs*) are actually the rebecs whose message servers can be called by instances of this rebec which basically means that each rebec can send messages (invoke message servers) of the rebecs specified in its parameter list (established when declaring a rebec) [1]. State variables (*statevars*) are variables which are holding the state of the rebec to which they belong. They can be accessed by the message servers in the same reactive class but not outside of it. The message servers (*msgsrv*) are methods responsible for handling the incoming messages. Beside message servers, we also have another type of methods that can be defined and those are known as *local methods*. As they are local, they can only be called by message servers and other methods within the same rebec. Both message servers and local methods can have input parameters of type

ExtType that includes regular *Type*. This rebec (or the instantiation of this reactive class) can be accessed (only within) by using a keyword *self* [1]. This is used by *local methods* to send messages to the rebec that contains the method. There is another type of variables aside from the variables that are part of the state space (*statevars*) and these are known as *local variables.* The use of local variables is self-explanatory as it is expected that we need to isolate certain variables or use them temporarily without a global effect on the class level. Similarly, in JAVA we have both global and local variables that can be comparably mapped with the state space variables and local variables in Rebeca, respectively. A constructor has to be included in Rebeca, in at least one of the reactive classes, as it serves its purpose being the initial message server (containing initial message call), that sets things going. Constructor is used in the initial state as a first message in the queue to be serviced that basically means that constructor will always be executed first. Additionally, it is used to initialize state variables, similarly as in JAVA. A simple example for defining a Rebeca model is provided in Listing 1.

```
reactiveclass Producer(2) {
  knownrebecs { Consumer knownconsumer; }
  statevars { boolean productsent; }
  Producer() {
    productsent = false;
    self.produce();
  }
  msgsrv produce() {
    knownconsumer.consume();
    productsent = true;
  }
}
reactiveclass Consumer(2) {
  knownrebecs { Producer knownproducer; }
  statevars { boolean productreceived; }
  Consumer() {
    productreceived = false;
    self.consume();
  }
  msgsrv consume() {
    knownproducer.produce();
    productreceived = true;
  }
}
main {
  Producer producer1(consumer1):();
  Consumer consumer1(producer1):();
}
```

Listing 1: Model definition in Rebeca [1]

$$
\begin{aligned}
Model &::= Class^{*}\ Main\\
Main &::= \textbf{main}\ \{\ InstanceDcl^{*}\ \}\\
InstanceDcl &::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*});\\
Class &::= \textbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars\\
&\quad MsgSrv^{*}\ LocalMethods^{*}\ \}\\
KnownRebecs &::= \textbf{knownrebecs}\ \{\ RebecDcl^{*}\ \}\\
Vars &::= \textbf{statevars}\ \{\ VarDcl^{*}\ \}\\
RebecDcl &::= className\ \langle v\rangle^{+};\\
VarDcl &::= Type\ \langle v\rangle^{+};\ |\ Type\ [\ number\ ]^{+}\ v\\
MsgSrv &::= \textbf{msgsrv}\ msgName(\langle ExtType\ v\rangle^{*})\ \{\ Stmt^{*}\ \}\\
LocalMethods &::= methodName(\langle ExtType\ v\rangle^{*})\ \{\ Stmt^{*}\ \}\\
Stmt &::= Assignment\ |\ SendMessage\ |\ MethodCall\ |\\
&\quad ConditionalStmt\ |\ LoopStmt\ |\ LocalVars\\
Assignment &::= v = Exp;\ |\ v = ?(Exp\langle, Exp\rangle^{+});\\
SendMessage &::= rebecExp.msgName(\langle Exp\rangle^{*});\\
MethodCall &::= methodName(\langle Exp\rangle^{*});\\
ConditionalStmt &::= if\ (Exp)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}]\\
LoopStmt &::= for\ (\ Exp\ ;\ Exp\ ;\ Exp\ )\ \{\ Stmt^{*}\ \}\ |\ while\ (Exp)\ \{\ Stmt^{*}\ \}\\
LocalVars &::= ExtType\ \langle v\rangle^{+};\\
Exp &::= e\ |\ rebecExpr\\
rebecExp &::= self\ |\ rebecTerm\ |\ (className)rebecTerm\\
rebecTerm &::= rebecName\ |\ sender\\
ExtType &::= Type\ |\ float\ |\ double\\
Type &::= boolean\ |\ int\ |\ short\ |\ byte\ |\ className
\end{aligned}
$$

Figure 2.5: Rebeca syntax [1]

3 Related Work

Many research papers have been done on the subject of UML formal semantics and formal verification with hope to define formal interpretations that in some cases go even beyond it. Different approaches have been proposed with similar goals and we will briefly mention the two most common approaches in this field. First is direct UML formalization, and the second is transformation of UML concepts to formal languages where they can be verified by using formal verification tools.

In the first approach mathematical theories are used for formalization of UML. In [18], authors propose a flexible and modular formalization approach based on temporal logic to deal with the lack of well-defined semantics, open to different interpretations. It also identifies some of the pitfalls, in many works done, due to the predefined, fixed semantics with limited interoperability, and proposes a modular, flexible semantics able to cope with existing interpretations and highlight the new ones [18]. In [10], authors describe the formalization of the UML meta-model in Slang (that is a formal methods language) while stating that these concepts, they researched in the paper, can be ascribed to any algebraic theory based formal language. They aim to formalize the whole checking procedure with a goal to present a process in which a UML translation can be formally verified. In [19], authors specify axiomatic semantics for UML (representing classes, associations, instances and general sub-models of UML) that are given in terms of structured theories in a simple temporal logic. These introduced semantics are appropriate for modular reasoning about UML models and they are based on the set-theoretic Z-notation model adopted by Syntropy (that is an object-oriented analysis and design method).

In the second approach, the formal semantics are provided to UML through conceptual mapping followed by a transformation of UML concepts to established formal languages where they can be verified by using formal verification tools. Our thesis can be classified as an effort oriented towards this approach where we aim to bridge the gap between UML and formal verification by providing a mapping procedure of the UML concepts towards Rebeca (that is an actor-based formal language). Related works for this approach include the mapping between UML models and Object-Z specifications (that is an object-oriented extension of Z-notation and represents a formal specification language for modeling computing systems) [20]. This paper also provides a formal semantic mapping between the two languages at the meta-level, that is responsible for making this transition more systematic. Besides, we have another related research paper regarding a formal method based around Abstract Machine Notation (AMN) or more precisely the transformation of state-machine diagrams towards the AMN of the B method [21]. Notably, B method is a software development methodology grounded on B which is a formal method (tool-supported) and is based on the AMN.

The use of UML profiles to facilitate transition to other domains is one of the most common approaches used for mapping UML to other languages. In the past decade, there have been many UML profiles proposed to OMG [22], and some of

them even became standards. In case of Rebeca and UML, the two research papers [3,5] are also proposing the use of UML profiles for enforcing transition of UML towards Rebeca. However, as we could see from their examples and many other examples that are available, UML profiles are significantly limiting the true potential of UML by creating a domain-specific UML language that is in a way a hybrid between the source UML and other domain-specific languages used as a target. This approach restricts the use of the new hybrid UML to only that domain (language).

Moreover, in the field of modeling reactive systems a lot of work have been done. Related papers cover the modeling of reactive systems by using UML and the transformation (generating code) from UML to target languages. For modeling of reactive systems, most research papers propose the use of the state-machine diagrams [14,21]. In [14], a UML profile is proposed for building concurrent reactive systems (that is grounded on state-machine diagrams). In [21], graphical design of reactive systems is introduced which is based on the transformation of state-machine diagrams towards the AMN of the B method. However, as we already mentioned there are two research papers directly related to the transformation of UML to Rebeca that are in fact not proposing the use of state-machine diagrams but a combination of structural and behavioral diagrams where sequence diagrams are used instead [3, 5]. These papers clearly justify such decision by stating that Rebeca and its underlying execution mechanisms (based on the actor-based modeling paradigm) are unique and require a different behavioral type of diagram. Notably, state-machine diagrams should only be used to portray the external and temporal events as well as the reaction of an object to them, as stated by different authors. This, however, does not apply in the case of Rebeca where dequeuing a message server and its execution cannot be considered as an external event [3, 5].

In this paper, we want to keep the UML in its native form without introducing a UML profile. In addition to that, we need to set some rules that will be used as correctness attributes for validating source UML models before conversion to Rebeca takes place and establish compatibility with the mapping procedure or essentially that we are in Rebeca domain. The combination of these rules implies a design pattern for modeling of the considered UML diagrams. Furthermore, we consider both structural and behavioral diagrams as UML models, particularly Class diagram and Object diagram as structural diagrams and Sequence diagram and State-machine diagram as behavioral diagrams. However, as we strive to get the maximum from the provided UML information and reduce the number of diagrams in the process, we primarily focus on three of the mentioned diagrams (Class diagram, Object diagram and one of the behavioral diagrams). Notably, the main differences between this paper and other two papers that also propose mapping between UML and Rebeca [3,5] is in the extensiveness and detailness of the mapping procedure as well as the lack of UML profile. In this respect, the two related papers only provide brief semantical mappings for structural concepts and include partial behavioral concepts without specifying the mappings for them, this work includes detailed conceptual mappings on the syntax level for complete structural and behavioral concepts of Rebeca.

4 Research Process

Evaluating the viable ways towards mapping of UML to Rebeca including identification of the required UML diagrams and their consisting information (for such mapping to be achievable) and providing a detailed enough mapping procedure with applicability scenarios to support the given procedure, requires a detailed research design taking into account all these activities that have to be accomplished. Considering this, we defined a clear process that corresponds to the given activities (Figure 4.1). Problem space phase of the research process cycle fully corresponds to the proposed research questions that are indeed the main sources of interest and motivation in this thesis. Solution space and assessment parts of the aforementioned process are high-level abstractions and are refined further as a separate process (Figure 4.2), in order to provide a more detailed description of the overall research process. This newly introduced process starts with a bottom-up approach where we first identify Rebeca concepts. Due to the extensiveness of Rebeca and its syntax we consider only Core Rebeca, although we briefly discuss how can Timed Rebeca and Probabilistic Timed Rebeca be modeled in the UML, as part of the potential improvements section. The next step in the process (Figure 4.2) is for each identified Rebeca concept to identify the minimum information needed to provide in the UML, for the mapping to be viable. This process is iterative, and it is repeated until all the identified Rebeca concepts have the matching pair in the UML. When this condition passes, as a result, the detailed mapping procedure emerges that will be validated in the final step of the process. The validation phase itself will be an important step in the whole story. It is divided into pre-conversion validation and post-conversion validation, where the latter entirely depends on the success of the former. In other words, the transformation shall not be done until all the correctness attributes from the pre-conversion validation are satisfied. Hence, the validation phase can be described as another important and structured bottom-up process consisting of pre-conversion validation of source models and post-conversion validation of target models (Figure 4.3).

We can define the research to be performed as a descriptive study that is characterized by a structured approach where at each stage, we go deeper into the core of the problem from which the descriptive solution emerges. The comprehensive description of the emerged mapping procedure is the main descriptive object, during the research.

Along with it, the identification of adequate validation techniques is the second descriptive object.

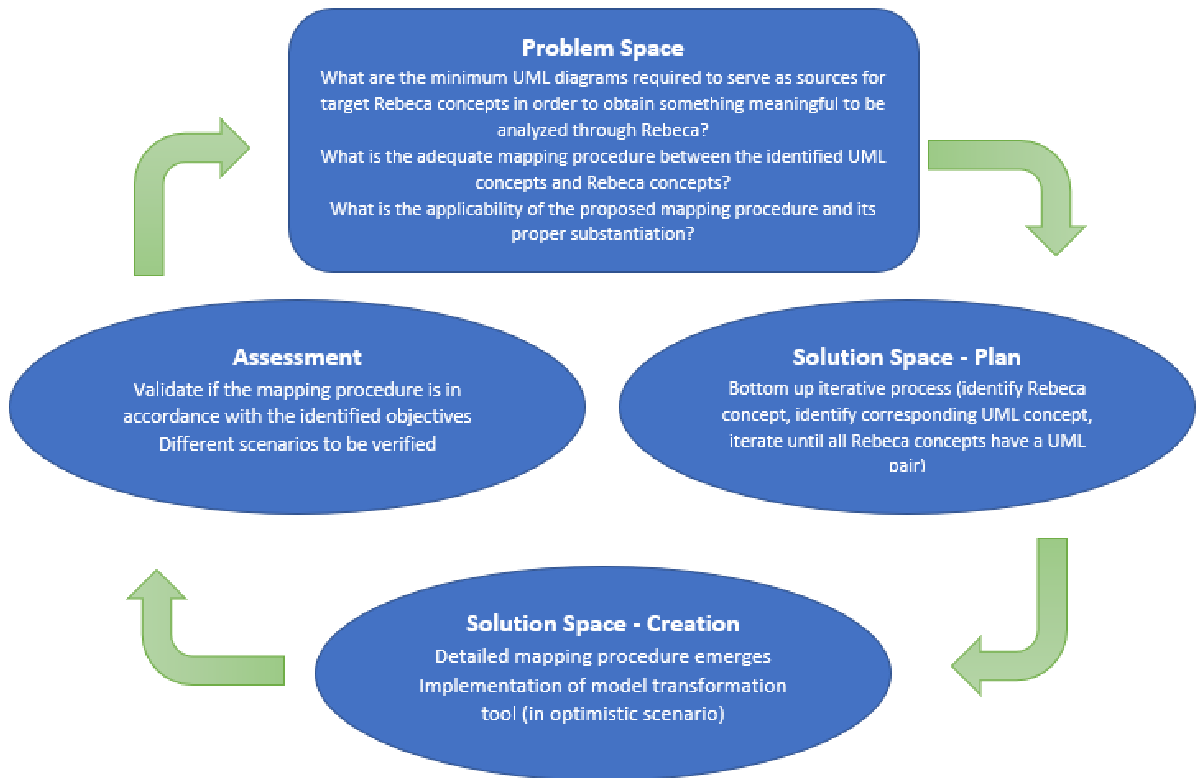

Figure 4.1: Research process cycle

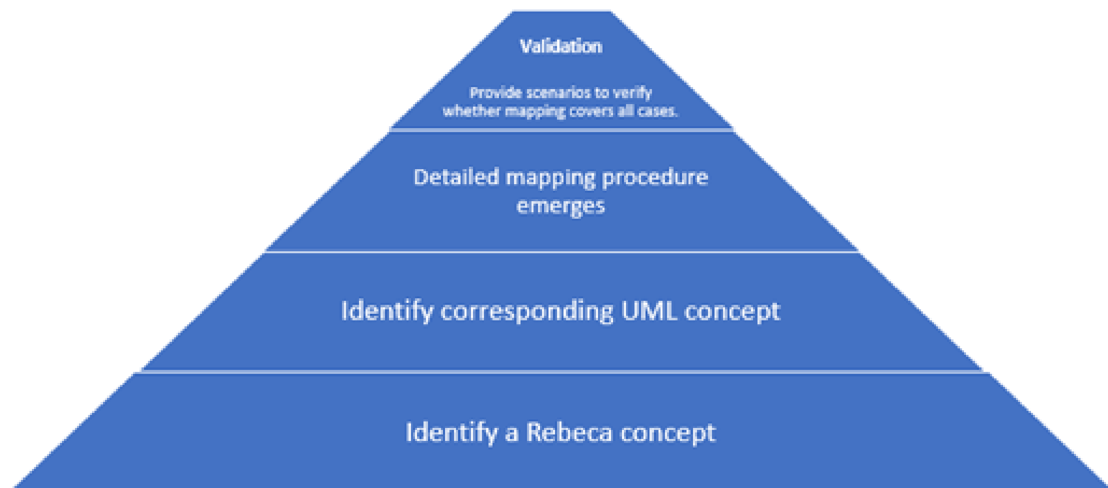

Figure 4.2: Mapping procedure - iterative creation process

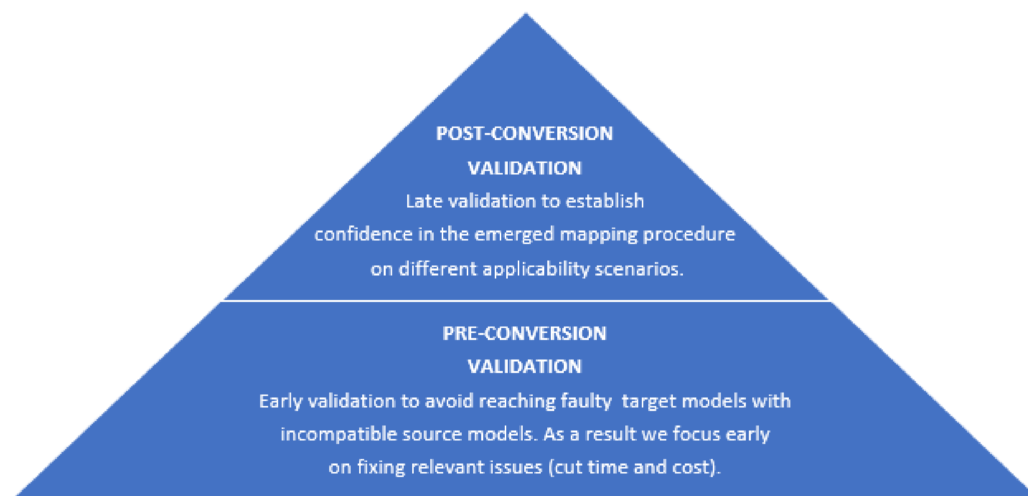

Figure 4.3: Validation process

Our main research goal is to propose a subset of UML modeling concepts that can be used for modeling the domain of reactive and distributed systems as modeled in Rebeca. Besides, this would enable transformation of UML models to Rebeca models and use of its formal verification tool. This mapping procedure is to be used after the design of a software solution is done, using a subset of UML diagrams. Among the many benefits, we mainly strive to cut the production and maintenance time as well as the cost of software solutions through formal verification early in the design process. Moreover, after establishing the correctness of Rebeca models, we can go even further and use model transformation to obtain executable JAVA code from Rebeca models. The overall objective of this thesis is to design the aforementioned mapping procedure and provide validation of the source UML models in pre-conversion validation and target Rebeca models in post-conversion validation. The research started with the formulation of research goals which are then investigated in the paper, followed by a literature review of papers and articles that focus on UML and model transformations on one side and Rebeca with the inclusion of the mapping between UML and Rebeca on the other side. The next step was a requirements analysis that lead to identification of the requirements based on which we built the conceptual mapping procedure. After the thorough investigation of Rebeca syntax [1,17], we proceeded by extraction of the complete Core Rebeca concepts. Afterward, the iterative process for the identification of corresponding subset of UML concepts and potential limitations, with respect to them, was conducted. It is worth mentioning that we used Papyrus modeling environment (in Eclipse) and UML specification for identification and modeling of UML concepts. The overview of the obtained results (comprehensive mapping procedure) is presented in the form of three tables, followed by validation phases including both pre-conversion validation and post-conversion validation. Conclusively, we discussed the limitations of this thesis and proposed directions upon which future research can be established.

5 A Mapping Procedure for Transformation of UML Models to Rebeca Models

5.1 Requirements Analysis

As aforementioned, one of the goals is to identify the Rebeca concepts that will be covered in this thesis which requires source models, to encapsulate the required information within them. These source models are represented by the identification of the corresponding subset of UML concepts. The requirement for selection of appropriate types of UML diagrams is essential while focusing on reducing their number and at the same time preserving the meaningful information within them that is necessary for mapping to be viable. Hence, we just consider a Class diagram and Object diagram as structural types of diagrams and State-machine diagram and Sequence diagram as behavioral types of diagrams.
The identification and the mapping of target Rebeca and source UML concepts results in the creation of the complete mapping procedure between the observed languages, in an optimistic scenario. However, in a more realistic scenario, when resources (as time) are considered, a line needs to be drawn and a reasonable merit of a detailed enough achievable mapping procedure has to be established. Hence, we consider only Core Rebeca by excluding Timed Rebeca, although we briefly discuss how can Timed Rebeca and Probabilistic Timed Rebeca be modeled in the UML, as part of the potential improvements section. Moreover, we are possibly going to identify certain Rebeca concepts for which there is no matching pair in the source UML or the critical merits as time, cost and complexity of introducing additional types of UML diagrams do not reflect the desired, expected results and objectives. In such situations, we document these cases as limitations of the emerged mapping procedure and we propose a possible solution for them, that is in the scope of the mapping procedure, when applicable. Some of these solutions are deemed as unreasonable for the scope of this thesis based on the available resources and objectives.
The other requirements of this thesis concern the identification of correctness attributes that will be used as a merit of the correctness/compatibility of source UML models in the pre-conversion validation. Here, we need to consider syntactic correctness regarding Rebeca concepts and what is possible or not possible to accomplish in Rebeca. Also, closely related merit of correctness is design correctness with respect to the design pattern used to convey the information within diagrams. Nevertheless, we include this category as part of the syntactic correctness, as it also deals with boundaries of Rebeca language and therefore can be attributed to its syntax. Notably, boundaries in Rebeca implies boundaries in the mapping procedure, hence necessity to equalize the source UML models and establish compatibility with the mapping procedure which inherently erases interpretations and enables to focus on the domain. As a result of the identification of such correctness attributes, we strive to provide definite correctness rules that convey the design pattern for modeling of the considered subset of UML diagrams.

5.2 Identification of Rebeca Concepts

Rebeca syntax and semantics are not explained as part of this section as they are already discussed in Section 2.3.2. This section strictly identifies the Rebeca concepts for which we will provide the mapping. These concepts are extracted from the domain [1, 17]. Hence, this section is organized as a single part that describes the identification of the sub-portion of Rebeca concepts for which we attempt to provide mapping with corresponding UML concepts.

5.2.1 Extraction and Analysis of Rebeca Concepts

After the analysis is performed, following Rebeca concepts have been identified:

1. Definition of reactive class
   (a) Generic: *reactiveclass ClassName(queue size)* {*class body*}
   (b) Example: *reactiveclass Producer(2)* {*...*}
   (c) Rule: Class name should start with a capital letter.
2. Definition of known rebecs or known reactive objects
   (a) Generic: *knownrebecs* {*specification of known rebecs*}
   (b) Example: *knownrebecs* { *Consumer knownconsumer;* }
   (c) Rule: Each known rebec is defined by a class name followed by the name of the object that is instantiated.
3. Definition of state variables
   (a) Generic: *statevars* {*specification of state variables*}
   (b) Example: *statevars* { *boolean productsent;* }
   (c) Rule: Each state variable is defined by a type (*Type* excluding *ExtType*) followed by the name of the variable.
4. Definition of constructor
   (a) Generic (without arguments): *ClassName()* {*constructor body*}
   (b) Generic (with arguments): *ClassName(ExtType argument,...)* {*constructor body*}
   (c) Example (without arguments): *Producer()* { *productsent = false; self.produce();* }
   (d) Example (with arguments): *Producer(boolean prodsent)* { *productsent = prodsent; self.produce();* }
   (e) Rule: Constructor is not preceded by any keyword other than the name of the class and it is used for initializing state variables and calling appropriate message servers. Notably, it is the first message that is executed by each rebec.

5. Definition of message servers

   (a) Generic (without arguments): *msgsrv MessageServerName()* {*message server body*}

   (b) Generic (with arguments): *msgsrv MessageServerName(ExtType argument,...)* {*message server body*}

   (c) Example (without arguments): *msgsrv produce()* {*...*}

   (d) Example (with arguments): *msgsrv produce(int numberOfUnits)* {*...*}

   (e) Rule: Message servers accept *ExtType* for its arguments and, in compare with local methods, message servers can be accessed from other reactive classes as well.

6. Definition of local methods

   (a) Generic (without arguments): *MethodName()* {*local method body*}

   (b) Generic (with arguments): *MethodName(ExtType argument,...)* {*local method body*}

   (c) Example (without arguments): *produce()* {*...*}

   (d) Example (with arguments): *produce(int numberOfUnits)* {*...*}

   (e) Rule: Local methods accept *ExtType* for its arguments and in compare with message servers, local methods can not be accessed from other reactive classes. Local methods can be *void* and *return* methods.

7. Definition of main

   (a) Generic: *main* {*main body*}

   (b) Example: *main* { *Producer producer1(consumer1):(); Consumer consumer1(producer1):();* }

   (c) Rule: After the reactive classes are defined, we use *main* for instantiating reactive classes and passing required arguments and known rebecs which enables the execution.

Moreover, with respect to the building blocks of the message servers, local methods and constructors, or more specifically their behavior and usage, we also identified certain semantics, as follows:

1. Conditional statements (if/else)

   (a) Generic: *if(condition)* {*logic if condition is TRUE*} *else* {*logic if condition is false*}

   (b) Example: *if(signal2 == false)* {*...*} *else* {*...*}

   (c) Rule: Conditional statement *if* can be defined alone without definition of *else* depending on the outcome that we are trying to achieve. In other words, *else* is optional. We can nest conditional statements and have a condition inside a condition.

2. Conditional logical operators

   (a) Generic: logical AND - *condition* && *condition*, logical OR - *condition* || *condition*, negation - *!condition*

   (b) Example: logical AND - *signal1 == true* && *signal2 == false*, logical OR - *signal1 == true* || *signal2 == false*, negation - *!signal1*

   (c) Rule: Conditional logical operators are used for connecting multiple conditions inside a single clause with the use of logical AND (&&) and logical OR (||) and to negate the value of a certain condition with the use of negation (*!*).

3. Conditional comparisons

   (a) Generic: equality comparison - *variable == value*, inequality comparison - *variable != value.* Other types: *variable < value, variable > value, similarly: <= (less than or equal to) and >= (greater than or equal to)*

   (b) Example: signal1 == true, signal2 != true etc.

   (c) Rule: Comparative operators are self-explanatory and do not require additional description.

4. Assignment

   (a) Generic: *variable = value;*

   (b) Example: *signal1 = true;*

   (c) Rule: Beside this, we can also use following assignment operations *+=, -=, *=, /=, %=.*

5. Call of message servers (and local methods)

   (a) Generic (message server defined in same rebec):
   self.messageServerName();

   (b) Generic (message server from known rebec):
   knownRebec.messageServerName();

   (c) Generic (message server with arguments in same rebec):
   self.messageServerName(argument,...)

   (d) Generic (message server with arguments from known rebec):
   knownRebec.messageServerName(argument,...)

   (e) Example (message server defined in same rebec): *self.Passed();*

   (f) Example (message server from known rebec):
   knownconsumer.consume();

   (g) Rule: Local methods are not presented. However, the same procedure applies for local methods except that local methods can only be called within the same *rebec*. This means that only the first case of message server invocation (within same *rebec*) applies to local methods.

Besides all the semantics that we identified from the provided examples, we will consider some additional Rebeca semantics that are important to provide a mapping for. As follows:

1. Loops (for and while)
   (a) Generic (for loop): *for(i = 0; i < N; i = i + 1) {loop body} OR for(i = N; i > M; i = i - 1) {loop body}, where N and M are natural numbers and i is iteration variable name*
   (b) Generic (while loop): *while(condition) {loop body}*
   (c) Example (for loop): *for(i = 0; i < 5; i = i + 1) {...}*
   (d) Example (while loop): *while(i < 5) {...}*
   (e) Rule: Loop condition can be defined slightly different but the core structure remains the same.
2. Arrays (definition and usage)
   (a) Generic (definition): *Type [size] variable;*
   (b) Generic (usage): *variable[N]* to get the value at the specified array position OR *variable[N] = value;* to assign the value to the array position. Notably, N is natural number within the scope of array size (*N >= 0, N < variable.length*).
   (c) Example (definition): *int [4] numberArray;*
   (d) Example (usage): *numberArray[0]* to get the value OR *numberArray[0] = 5;* to set the value.
   (e) Rule: The arrays can only be of type *Type* excluding *ExtType*.
3. Non-deterministic expressions
   (a) Generic: *variable = ?(value1, value2,.. valueN);*
   (b) Example: *signal1 = ?(true, false);*
   (c) Rule: Value that is passed in the non-deterministic clause can be an expression and it is deemed as valid if it's value is determined at the compile time.

5.3 Identification of Corresponding UML Concepts - Iterative Mapping

When all the considered Rebeca concepts have been identified and documented we can proceed by providing a corresponding UML concepts that can be used as mapping pairs in the process of creation of the mapping procedure. This process is iterative as we iterate both through UML and identified Rebeca concepts and perform detection of mapping pairs. After the deep analysis of the documented Rebeca concepts is completed, we considered the subset of modeling concepts of the Class

Diagram, Object Diagram and Sequence Diagram, by excluding a State-machine diagram. This was done with reasons to avoid adding an unnecessary additional layer of complexity that would not significantly benefit the already provided mapping procedure and semantic richness of the other three included diagrams. Moreover, we opt to provide a behavioral richness that is significant and sufficient by the use of only sequence diagram that will be enriched with minor additional concepts, to compensate for the elimination of the State-machine diagram.

We divide this section based on the type of UML diagrams that are used as sources for mapping with identified Rebeca concepts. Hence, we have a separation of structural and behavioral UML concepts and for each of them, a specification of corresponding Rebeca concepts is identified and documented.

5.3.1 Structural UML Concepts

UML structural diagrams are very important for a provision of a skeleton of the modeled system and establishing a connection with Rebeca's structural blocks. Only two structural diagrams are considered in this thesis and these are class diagram and object diagram. This was done as an attempt towards a minimalist approach for the provision of a detailed mapping procedure as these two types of diagrams provide the exact structural information that is necessary for mapping to be viable and complete. A class diagram is a natural choice of a structural diagram when object-oriented languages are considered which in this case is true as Rebeca uses JAVA like syntax and therefore certain concepts from Object Modeling. Moreover, as Rebeca has a *main* block in which it instantiates the objects and passes the required *known rebecs* for all instantiated objects as well as constructor arguments when applicable, we need another type of UML diagram that can serve as a source of information for this cause. Object diagram suits perfectly for this purpose as it lets us to define relations between instantiated objects as well as define the values of constructor arguments when applicable. On the other side, the class diagram could not be used for this purpose as it focuses strictly on modeling the class level layer.

Class Diagram Class diagram and its semantics include the following elements that can be used for mapping with Rebeca concepts in the following manner:

1. Class
   (a) Mapped with: Rebeca's definition of reactive class
   (b) Definition of mapping: *ClassName* $\xrightarrow{\text{mapped}}$ *reactiveclass ClassName (??queue size??) {class body}*
   (c) Comment/Limitation: The queue size is not an explicitly available option when setting a class name in the class diagram and we propose two approaches that can be applied here. One is to leave it as it is and define it afterward manually which is not a solution but limitation. The other concerns the definition of the altered class name to accommodate for the queue size, Rebeca property. An example of this is given in the next list item.

(d) Possible solution: Definition of altered class name in a Class diagram: *ClassName[queuesize]*, or as an example: *Producer[2]*.

2. Attribute

   (a) Mapped with: Rebeca's definition of state variable

   (b) Definition of mapping (regular attributes): *attributeName:Type[1]* $\xrightarrow{\text{mapped}}$ *Type variableName;* ENCLOSED by the state space container: *statevars*{}. WHERE, *attributeName* and *variableName* are the same.

   (c) Definition of mapping (regular attributes with default value): *attributeName:Type[1] = value* $\xrightarrow{\text{mapped}}$ *Type variableName;*, PLUS assignment in the constructor: variableName = value;

   (d) Definition of mapping (array attributes): *attributeName:Type[N]* $\xrightarrow{\text{mapped}}$ *Type [N] variableName;*, ENCLOSED by the state space container: *statevars*{}. WHERE N>1

   (e) Comment/Limitation: Notably, the definition of state variables in Rebeca requires a state space that is defined by *statevars*{} and inside which the state variables are specified. This means that along with the first state variable, we must also provide state space used as a container for all the upcoming state variables inside the observed class. Regarding the constraints ([1] and [N]), that separates singular variables from arrays, there is a minor limitation here in the UML. This comes from the fact that in order to define an array, the *N* constraint has to be greater than one (N>1) which therefore excludes the creation of single element arrays. In favor of this approach, we state that there is no logical explanation for the creation of singular arrays, as the main reason for their existence, in the first place, is having a variable capable of containing more than one element.
   Additionally, default value for the regular attributes has to be assigned in the constructor and provided for all attributes whose values are statically assigned in the constructor (without dynamic argument passing). The assignment has to be done after the constructor has been generated as part of the operation to constructor mapping, considering that we do not know if the constructor has arguments or not at this point.

   (f) Possible solution: Not applicable.

3. Operation/method

   (a) Mapped with: Rebeca's definition of message server or local method or constructor

   (b) Definition of mapping (message server): *methodName(in parameterName: Type,...)* $\xrightarrow{\text{mapped}}$ *msgsrv MessageServerName(ExtType argument,...)* {*message server body*}

(c) Definition of mapping (local method): *methodName(in parameterName: Type,...):Type* $\xrightarrow{\text{mapped}}$ *ExtType LocalMethodName(ExtType argument,...)* {*local method body with a return statement of the adequate type*}

(d) Definition of mapping (constructor): *methodName(in parameterName: Type,...)* $\xrightarrow{\text{mapped}}$ *ClassName (ExtType argument,...)* {*constructor body*}

(e) Comment/Limitation: Notably, arguments are optional and the mapping remains the same without them, in which case we just exclude them from the mapping. A constructor has to be included in Rebeca, in at least one of the reactive classes, as it serves its purpose being the initial message server (containing initial message call), that sets things going. Additionally, it is used to initialize state variables. The rules for the constructor arguments are defined in the next mapping. Additionally, there is a limitation regarding a definition of operations in the class diagram with respect to the available Rebeca semantics or more precise definition of message servers and local methods and constructors. This is due to the possibility to have message servers, local methods and constructors in Rebeca that are defined in a different way. However, on the UML side, we have just regular operations defined in the class diagram. Moreover, as we want to avoid introducing Rebeca semantics into the UML we need to find a better way to distinguish between message servers, local methods and constructors, on the UML side, to be able to provide an adequate mapping between these semantics in UML and Rebeca. We identified a possible way to deal with this issue and we document it under the following list item.

(f) Possible solution: We make local methods *private* in the class diagram (as they are only accessible from the same *rebec* and can be perceived as the class level methods). On the other side, we leave message servers *public*. Additionally, when we consider constructors we can notice that they are defined in the same manner as message servers in the class diagram (as public operations), which is problematic. To deal with this issue, we specify that any public operation, whose name is equal (in addition to case sensitive) to the name of the consisting class, is taken as the constructor. Otherwise, it is taken as the message server.
We take this approach for the mapping as it appears to be the most meaningful and simple solution. Public and private markers represent a modeling semantic in the class diagram, that is not expressed in a textual format but graphical (as green plus or a red minus signs at the bottom right corner, respectively).

4. Constructor operation/method argument

(a) Mapped with: Rebeca's definition of constructor argument and assignment to corresponding state variable

(b) Definition of mapping: *in parameterName: Type* $\xrightarrow{\text{mapped}}$ *ExtType argumentName* PLUS assignment in the constructor body to the state variable: *stateVariableName = argumentName;*

(c) Comment/Limitation: Notably, assigning the argument to the adequate state variable is not straightforward as we need to match the appropriate argument with its corresponding state variable. We identified two possible ways of handling this issue and we present them in the following list item.

(d) Possible solution

i. We add a separate sequence diagram to model the assignment of the arguments to their corresponding state variables. This makes sense as this is how we handle the assignment in local methods and message servers.

ii. We decide on a naming convention for constructor arguments. For example, arguments are named as follows: *argumentName = stateVariableName + "Arg" keyword => argumentName = stateVariableNameArg*

We choose the second approach for this issue with reasons to avoid introducing additional complexity that arises with the provision of too many sequence diagrams. Additionally, constructors are not equivalent to message servers and local methods as constructors are used for initializing objects and are invoked implicitly whereas message servers and local methods are used to exhibit functionality of an object and are invoked explicitly. Also, a sequence diagram is a behavioral diagram while assignment conducted in the constructors is predictable from the aspect of structural diagrams in the UML. On the other side, constructors may exhibit some behavioral aspects as well, but more with respect to invocation of message servers that serve as a behavioral initialization of the class level logic in Rebeca. For this part of constructors, we truly need a sequence diagram as it is impossible to represent it in any other way, in a reasonable fashion. These are some of the arguments that go in favor of the selection of the second approach.

5. Association or relation with other classes

(a) Mapped with: Rebeca's definition of known rebecs

(b) Definition of mapping: *associationCardinalityAndName* $\xrightarrow{\text{mapped}}$ *dictates the number of known rebecs (of the specified type or class from which association is drawn) and the name of instantiated objects*

(c) Comment/Limitation: Notably, we could argue if this information should be used for mapping with instantiated objects within known rebecs with reasons that it may not be provided in its adequate form in UML class diagram, that is necessary for mapping to be viable. Most often, only the cardinality is defined in a proper form while the name may not be defined appropriately as the list of comma-separated object names or a single object name (if the cardinality is one). However, we could make an assumption that the consisting information within the class diagram

is modeled accordingly. In any case, we could use this information for establishing correctness or identifying inconsistencies with the information provided elsewhere in the structural or behavioral UML semantics, that could then be used for correctness evaluation of source UML models as part of the pre-conversion validation with respect to these concepts.

(d) Possible solution: Not applicable.

Object Diagram Notably, since UML 2.5, there is no explicit specification of Object diagram. But, considering its similarity and close relation with the class diagram (without which it makes no sense for the object diagram to exist), the UML 2.5+ specifies a notation for instances of classifiers. This is essentially the class diagram with the use of specialized modeling elements for representing the instances of the class. Basically, it requires a class diagram from which it will reference the classes and properties, that are instantiated. Moreover, this new definition of the object diagram representation is employed by Papyrus, where instance specifications (both nodes and edges) are used for definition of objects, in the class diagram. As aforementioned, the object diagram is used for mapping with the corresponding Rebeca concepts that are defined as a part of the *main*. It suits perfectly for this purpose as it enables us to define relations between instantiated objects. Object diagram representation and its semantics include the following elements that can be used for mapping with Rebeca concepts in the following manner:

1. Object name
   (a) Mapped with: Rebeca's rebec name when instantiating a class within *main*
   (b) Definition of mapping: *objectName: ClassName* $\xrightarrow{\text{mapped}}$ *ClassName objectName():();* , ENCLOSED by the *main* block
   (c) Comment/Limitation: The definition of the *main* or enclosing *main* block is not exactly the part of the mapping but will be created with the first mapped object as the container that all subsequent objects from the object diagram will use.
   (d) Possible solution: Not applicable.
2. Object argument
   (a) Mapped with: Rebeca's definition of constructor argument when instantiating a class within *main*
   (b) Definition of mapping: *attributeName:Type[N] = argumentValue* $\xrightarrow{\text{mapped}}$ *ClassName objectName():(argumentValue);* , ENCLOSED by the *main* block
   (c) Comment/Limitation: The definition of the *main* or enclosing main block is not exactly the part of the mapping but will be created with the first mapped object as the container that all subsequent objects from the object diagram will use. Nonetheless, the definition of mapping shows how

a single argument is mapped. Notably, multiple arguments are handled in a similar fashion except they appear as a comma-separated list of arguments in the Rebeca. Here, we can argue on how to provide the mapping between object arguments and constructor arguments to ensure that the right argument is passed in the constructor. However, considering that, in Papyrus, we cannot explicitly specify the name of the object argument, but need to provide a reference to the attribute from the corresponding class diagram, we can simply use that referenced name for the mapping with the constructor argument. This is only true because we specified that the name of the constructor argument is equal to the attribute name with an addition of "Arg" keyword at the end.

(d) Possible solution: Not applicable.

3. Object relation

   (a) Mapped with: Rebeca's definition of known rebec argument when instantiating a class within *main*

   (b) Definition of mapping: *object1 in relation with object2* $\xrightarrow{\text{mapped}}$ *ClassName objectName1(objectName2):(); ClassName objectName2(objectName1):();* , ENCLOSED by the *main* block

   (c) Comment/Limitation: The definition of the *main* or enclosing main block is not exactly the part of the mapping but will be created with the first mapped object as the container that all subsequent objects from the object diagram will use. Nonetheless, these known rebecs that are mapped and passed as arguments along with the instantiation of classes within *main* has to coincide with the known rebecs defined within reactive classes and we assume their correctness. Hence, it will be used as part of the correctness check in the pre-conversion validation.

   (d) Possible solution: Not applicable.

Conclusively, we noticed how certain semantics of the object diagram can also be used for establishing correctness of the source UML models, with respect to the pre-conversion validation. These semantics include the *known rebecs* that are defined within each reactive class and correctness is accomplished by comparing relations between objects modeled in the object diagram with the closely related semantics defined in the sequence diagram as well as associations between classes defined in the class diagram. Nonetheless, these semantics also include the arguments defined in the constructors and correctness is accomplished by comparing the object arguments defined in the object diagram with the constructor arguments defined in the class diagram.

5.3.2 Behavioral UML Concepts

UML behavioral diagrams are necessary for injection of logic into the skeleton of application established by structural diagrams. The requirement for the behavior

is well expected in order to obtain something meaningful to be analyzed in Rebeca. Only the sequence diagram is considered in this thesis as a behavioral type of diagram. This was done with reasons to avoid introducing an additional layer of complexity that arises with the usage of State-machine diagrams (considering the complexity of its semantics) that would not significantly benefit the already provided mapping procedure. The reason why we favor a sequence diagram over the state-machine diagram is due to its clear and rich semantics that are adequate for the behavioral semantics of the Rebeca. As aforementioned, a sequence diagram describes the behavior of several objects within a single use case which is suitable for modeling the behavior of the Rebeca's message servers and local methods that are indeed the major containers of behavioral logic in Rebeca. Moreover, we opt to provide the behavioral richness, that is sufficient, by the use of only sequence diagrams, that will be enriched with minor additional semantics, for certain Rebeca concepts, to compensate for the elimination of State-machine diagrams.

Sequence Diagram As aforementioned, a sequence diagram is used for mapping with the corresponding behavioral Rebeca concepts that are defined as a part of the message servers and local methods as well as the constructor logic. In other words, each sequence diagram represents a single method (whether it is a message server, local method or constructor). It suits perfectly for this purpose as it lets us to define sequences of messages that are exchanged between the objects via message servers as well as internal circular messages that are exchanged via message servers and local methods. Moreover, as can be seen in the Figure 5.5, the *lifeline* that is the source of the message call is presented on the left side with only the class specified in its name and the *lifeline* that is the target of the message call is on the right side with both the object name and the corresponding class name specified. The reason for this becomes clear when we consider that we are modeling a single method within each sequence diagram (which is attributed to class level logic) and each of these methods can invoke methods from the objects of other classes, that are defined within them (which is common when it comes to Rebeca and definition of known rebecs). This is the expected modeling notation with respect to this thesis and these minor additional semantics, that we introduced, will be part of the pre-conversion validation.
For most of the semantics of the sequence diagram we provide a figure along with the mapping to clarify used textual representations of UML modeling elements. Sequence diagram and its semantics include the following elements that can be used for mapping with Rebeca's concepts in the following manner:

1. Combined Fragment of type ALT
   (a) Mapped with: Rebeca's conditional statements (if/else)
   (b) Definition of mapping: [*condition*] and [*else*] (see Figure 5.1) $\xrightarrow{\text{mapped}}$ *if(condition)* {*logic if condition is TRUE*} *else* {*logic if condition is false*}
   (c) Comment/Limitation: Before the condition statement inside the first *Interaction Operand* (Figure 5.1), we do not specify any keyword as it is common knowledge when in the *ALT Combined Fragment* the condition

is stated, it refers to the *if* statement. Hence, specifying only condition is self-sufficient. On the other side, we do not specify any condition for *else* (except the keyword itself) as we logically assume it is the negation of the *if* condition. Moreover, *else* is optional and we can have one Interaction operand representing specification of *if* condition, if we do not need an alternative branch.

(d) Possible solution: Not applicable.

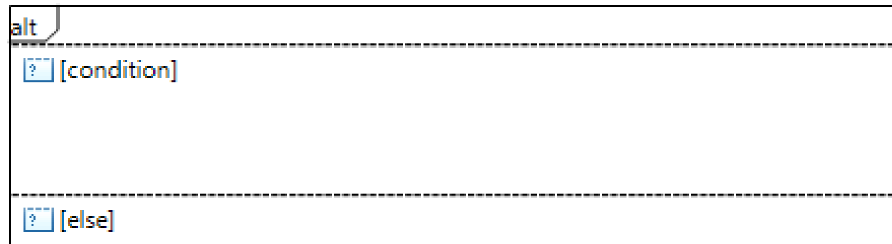

Figure 5.1: Combined Fragment of type ALT

2. Combined Fragment of type ALT with logical operator

(a) Mapped with: Rebeca's conditional logical operators

(b) Definition of mapping: logical AND - [*condition1* && *condition2*] and [*else*] (see Figure 5.2) $\xrightarrow{\text{mapped}}$ *if(condition1 && condition2)* {*logic if condition is TRUE*} *else* {*logic if any of two conditions is FALSE*}

(c) Comment/Limitation: Similarly, we can use logical OR (||) as well as negation (!). However, this is not exactly the common UML practice. Usually, conditions are plain and singular and in the case when multiple conditions are necessary, they can be nested. Anyway, the presented approach can also be used mostly with reasons to avoid nesting too many conditions that might produce unreadable and unclear sequence diagrams.

(d) Possible solution: Not applicable.

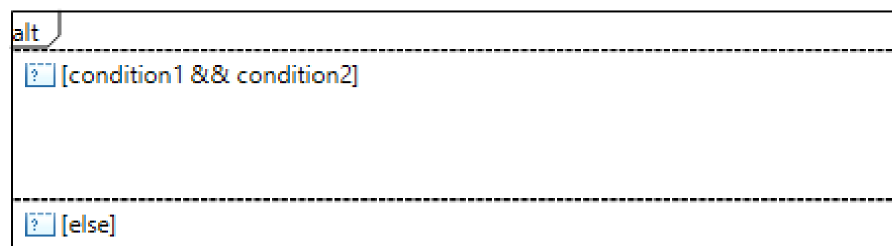

Figure 5.2: Combined Fragment of type ALT - with logical operator

3. Combined Fragment of type ALT with conditional comparisons

(a) Mapped with: Rebeca's conditional comparisons

(b) Definition of mapping: equality comparison - [*variable == value*] (see Figure 5.3) $\xrightarrow{\text{mapped}}$ *if(variable == value)* {*logic if condition is TRUE*}

(c) Comment/Limitation: Similarly, we use other conditional comparisons as inequality comparison (*!=*) as well as the others (*<, >, <=, >=*). Conditional comparisons are handled in the same manner in both the UML and Rebeca. Although, UML provides more freedom when specifying conditional comparisons, this is the most common approach and the expected one for this mapping. Moreover, these semantics are taken as correctness rules and will be used to validate source UML models in pre-conversion validation.

(d) Possible solution: Not applicable.

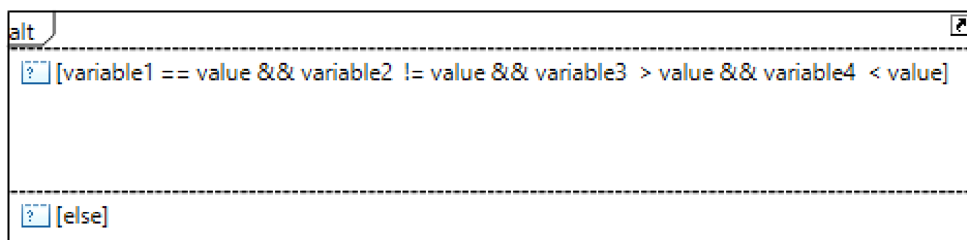

Figure 5.3: Combined Fragment of type ALT - with conditional comparison

4. Initiating message call from generic sender

(a) Mapped with: Rebeca's identification of message server or local method that represents a container of all the proceeding semantics from the corresponding sequence diagram

(b) Definition of mapping: *Action Execution Specifications* between generic sender and containing class (of the modeled message server or local method): *messageName()* (see Figure 5.4) $\xrightarrow{\text{mapped}}$ *No target of the mapping.*

(c) Comment/Limitation: Notably, initiating message call is used for locating the message server or local method, that is generated as part of the class diagram, which is indeed the main modeling object or container of the corresponding sequence diagram. In other words, we do not get any target Rebeca semantics as part of this mapping, but simply get the crucial information about the name of the message server or local method and its containing class where all the proceeding semantics of the particular sequence diagram will be contained. Sender represents any possible call of the certain message (from inside or outside the class) and it serves its purpose as the identifier or pointer to the location where the proceeding translated semantics, from the sequence diagram, will be extracted. The name of the class that contains the aforementioned message server or local method is retrieved from the sequence diagram *Lifeline* name of the target node, in the scope of the invoked method. Notably, actor-based modeling is by definition based on the asynchronous message passing which is why

it is reasonable to only expect the async calls in the sequence diagram or to treat all calls as async no matter of their modeled definition.

(d) Possible solution: Not applicable.

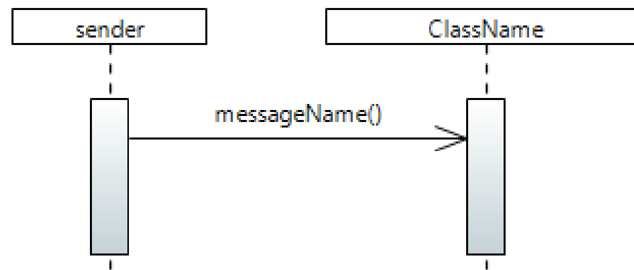

Figure 5.4: Initiating message call

5. Message call representing invocation of method of a different class from the initiating (excluding initiating message call)

   (a) Mapped with: Rebeca's call of message server of a different class from the initiating

   (b) Definition of mapping: async message call between two *Action Execution Specifications* of different classes: *messageName()* (see Figure 5.5) $\xrightarrow{\text{mapped}}$ *knownRebec.messageServerName();*

   (c) Comment/Limitation: Notably, in the sequence diagram, another class is represented as an object of that class that is contained by the initiating class, in the list of its known rebecs. The name of the known rebec, inside the mapping, is retrieved from the *Lifeline* name of the target object in the scope of the invoked method. Also, actor-based modeling is by definition based on the asynchronous message passing which is why it is reasonable to only expect the async calls in the sequence diagram or to treat all calls as async no matter of their modeled definition.

   (d) Possible solution: Not applicable.

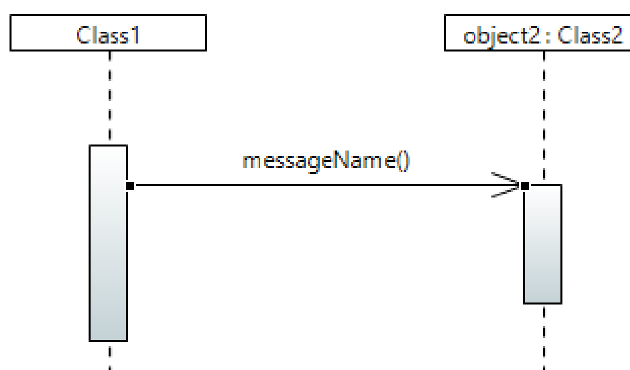

Figure 5.5: Asynchronous message call to another class

6. Message call representing invocation of method of the same class as the initiating

   (a) Mapped with: Rebeca's call of message server or local methods of the same class as initiating
   (b) Definition of mapping: async message call within one *Action Execution Specification* of the same class: *messageName()* (see Figure 5.6) $\xrightarrow{\text{mapped}}$ *self.messageServerName();* OR *self.localMethodName();*
   (c) Comment/Limitation: Message servers and local methods are mapped here with reasons that both can be invoked inside the same class, in the same manner. In other words, we do not need to make a separation between them as in the sequence diagram they appear identically as well as in Rebeca where they are preceded by keyword *self* followed by the method name (which in one case is message server and in the other local method). Notably, actor-based modeling is by definition based on the asynchronous message passing which is why it is reasonable to only expect the async calls in the sequence diagram or to treat all calls as async no matter of their modeled definition.
   (d) Possible solution: Not applicable.

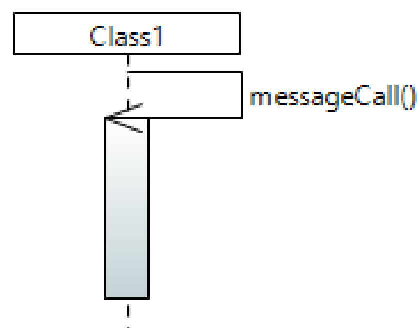

Figure 5.6: Asynchronous message call to self

Considering that we can have two types of loops (for and while) we need to ration about appropriate representations in the sequence diagram to distinguish between them. Based on the definition of a while loop we can conclude that any loop that has only the specified boolean condition without the range is considered to be a while loop. On the other side, any loop that besides a boolean condition also includes the range is considered to be a for loop. Notably, *while* loop can include a wider range of conditions (generally all possible boolean conditions). On the other side, *for* loop is focusing only on range conditions as it is commonly used to iterate over the range of values (i.e. arrays etc.). The mapping is as follows:

1. Combined Fragment of Type Loop with the inclusion of only boolean condition (excluding range)

   (a) Mapped with: Rebeca's while loop

(b) Definition of mapping: [*condition*] (see Figure 5.7) $\xrightarrow{\text{mapped}}$ *while(condition)* {...}

(c) Comment/Limitation: In the while loop, any boolean condition can be passed down in the condition clause. However, keep in mind that, that use of conditions that are always true (refers commonly to static condition clauses but also certain dynamic clauses), result in infinite loops.

(d) Possible solution: Not applicable.

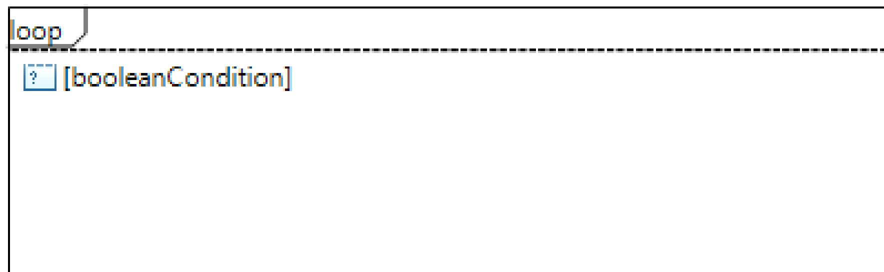

Figure 5.7: Combined Fragment of type LOOP - excluding range

2. Combined Fragment of Type Loop with the specification of range (excluding explicit iteration variable)

   (a) Mapped with: Rebeca's for loop

   (b) Definition of mapping: [*startRangeValue, endRangeValue*] (see Figure 5.8) $\xrightarrow{\text{mapped}}$ *for(int i = startRangeValue; i < endRangeValue; i++)* {...}

   (c) Comment/Limitation: Notably, when the iteration variable is not specified, then the default name is used. In this case, it is crucial not to declare the variable with the same name inside the loop (to avoid overriding its value), which means that the chosen default name of iteration variable is, up to some degree, a reserved word. Moreover, it is on the modeler to be self-aware of these rules in order to avoid unexpected results. All the rules are specified under pre-conversion validation. It is also important to mention a scenario when multiple loops are nested in which case the name of the iteration variable is incrementally created (when not explicitly specified), so that inside each nested loop except the first, iteration variable gets an incremented number to its name, starting by two (i.e. i, i2, i3, i4, etc.).

   (d) Possible solution: Not applicable.

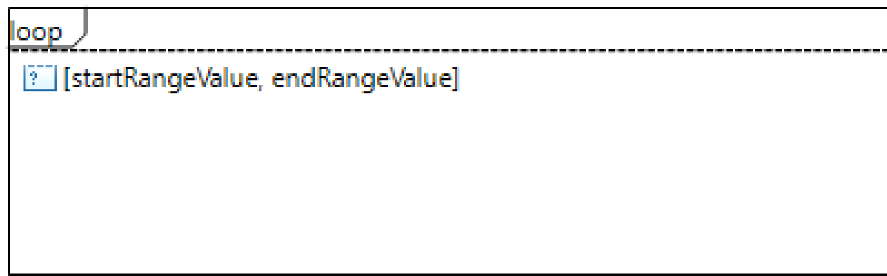

Figure 5.8: Combined Fragment of type LOOP with the inclusion of range

3. Combined Fragment of Type Loop with the specification of range and iteration variable
   (a) Mapped with: Rebeca's for loop
   (b) Definition of mapping: [*startRangeValue, endRangeValue, iterationVariableName*] (see Figure 5.9) $\xrightarrow{\text{mapped}}$ *for(int iterationVariableName = startRangeValue; iterationVariableName < endRangeValue; iterationVariableName++)* {...}
   (c) Comment/Limitation: Notably, in the case of nested loops with the inclusion of the iteration variable specification, it is obvious that their names should differ to avoid overriding the value of the iteration variable defined in the parent loop.
   (d) Possible solution: Not applicable.

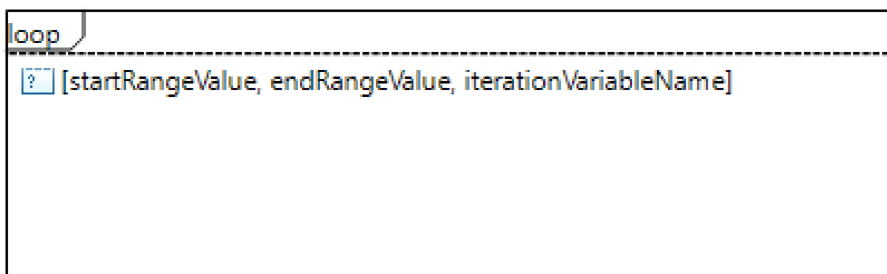

Figure 5.9: Combined Fragment of type LOOP with the inclusion of range and iteration variable

Considering that sequence diagram represents the sequence of messages or method invocations, performing common inline assignment expression for assigning the value to a variable is therefore not possible in the way it is done in Rebeca. Hence, we need a way to express the assignment as the message call that will then be mapped to corresponding assignment expression in Rebeca. Nonetheless, we distinguish between two types of assignment expressions: inline assignment that is commonly used for both local and global variables with special accent on local variables, and use of setter methods that can only be used for setting the value of global variables. Hence, we provide mapping for both of them. As follows:

1. Assignment of values to attributes (global)
   (a) Mapped with: Rebeca's setter method
   (b) Definition of mapping: *setStateVariableName(value)* (see Figure 5.10) $\xrightarrow{\text{mapped}}$ *self.setStateVariableName(value);*, WHILE auto-generated form of setter is: *void setStateVariableName(ExtType value)* { *stateVariableName = value;* }
   (c) Comment/Limitation: There is no explicit information to be provided on the UML side for the creation of setter methods as they are automatically created when mapping of state variables is performed. These methods follow the naming convention of the corresponding setter methods commonly defined in the JAVA so that for all types, setter method name starts with the word *set* followed by the name of the variable (using camel-case practice). Naturally, in the setter argument clause, there is a new value of the same type as the state variable, to be set. Notably, state variables defined as array could potentially employ two types of setters. First, regular setter of a certain type of array. Second, setter of the value on the particular position in the array, so that besides the new value that is passed as an argument we also have a position in the array on which the new value is written. In this case, the value argument is not of array type but basic singular type. This being said, the call of these setter methods is done as for any other local method (Figure 5.10).
   (d) Possible solution: Not applicable.

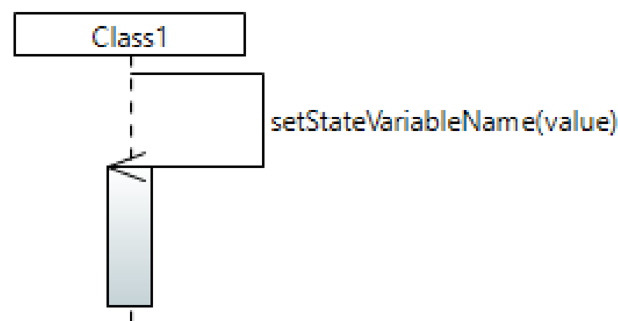

Figure 5.10: Setter global method call

2. Inline assignment of values to attributes (local and global)
   (a) Mapped with: Rebeca's inline assignment of value
   (b) Definition of mapping (regular assignment): *set(variable, value)* (see Figure 5.11) $\xrightarrow{\text{mapped}}$ *variable = value;*
   (c) Definition of mapping (array assignment at specified position): *set(array[position], value)* (see Figure 5.12) $\xrightarrow{\text{mapped}}$ *array[position] = value;*,

(d) Definition of mapping (assignment with specified argument operator): *set(variable, value, assignmentOperator)* (see Figure 5.12) $\xrightarrow{\text{mapped}}$ *variable assignmentOperator value;*

(e) Comment/Limitation: Notably, instead of the value, we can also pass a casted value as well as non-deterministic expression, in the same way as it is accomplished in Rebeca. Additionally, a regular inline setter can be used to set an array but only if we are assigning an array to some array variable. While, on the other side, if we want to assign the value to a certain position in the array, then we can use the presented approach (Figure 5.11). Nonetheless, it is important to provide support for other assignment operators. Default assignment operator is =, and the available arithmetic assignment operators are *+=, -=, *=, /=, %=*, that can be used as it is presented (Figure 5.13).

(f) Possible solution: The limitation of the sequence diagram with respect to the inline assignment is identified and the solution is provided as part of the mapping above.

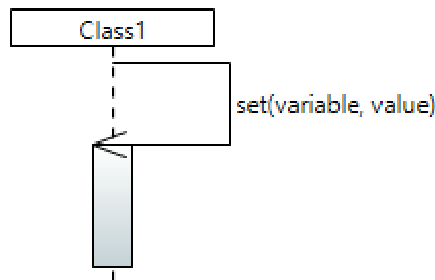

Figure 5.11: Regular inline set

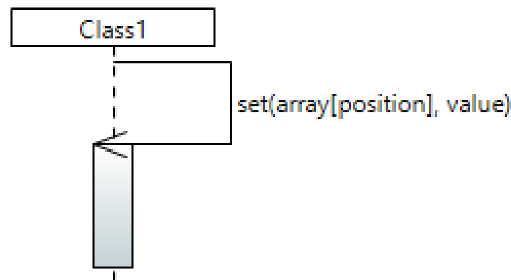

Figure 5.12: Array position inline set

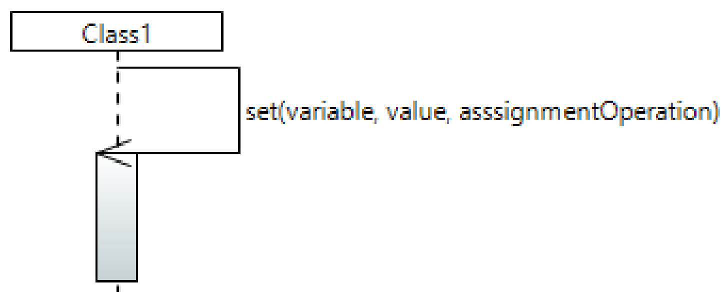

Figure 5.13: Inline set by overriding default assignment operator

Similarly, as for the inline assignment, declaration of local variables in the sequence diagrams faces the same issues. Hence, we need to express the declaration as a message call in the sequence diagram. As follows:

1. Declaration of local attributes (with and without initialization)
   (a) Mapped with: Rebeca's declaration of local variables
   (b) Definition of mapping (declaration of regular local variable without initialization): *declare(variableName, Type)* (see Figure 5.14) $\xrightarrow{\text{mapped}}$ *ExtType variableName;*
   (c) Definition of mapping (declaration of regular local variable with initialization): *declare(variableName, Type, value)* (see Figure 5.15) $\xrightarrow{\text{mapped}}$ *ExtType variableName = value;*
   (d) Definition of mapping (declaration of local array without initialization): *declare(variableName, Type[N])* (see Figure 5.16) $\xrightarrow{\text{mapped}}$ *Type [N] variableName;* WHERE N>1
   (e) Comment/Limitation: Notably, declared variables can be initialized with casted values as well as non-deterministic expressions (provided as value argument), in the same form as they are defined in Rebeca. It is worth mentioning, that regular local variables are of type *ExtType* in Rebeca. However, the arrays are limited to the type *Type*.
   As for the declaration of local arrays with initialization, it is handled in an identical manner as for the regular variables, except that for arrays, the value has to be an appropriate array. Additionally, instead of the value, we can also pass a casted value and in that manner provide support for casting. Same goes for non-deterministic expressions.
   (f) Possible solution: The limitation of the sequence diagram with respect to the declaration of local variables is identified and the solution is provided as part of the mapping above.

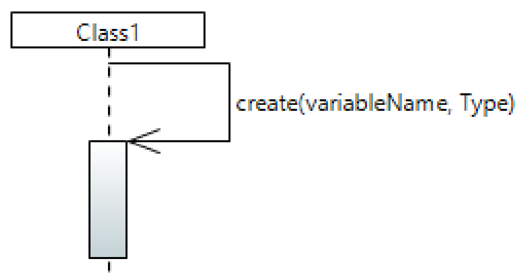

Figure 5.14: Regular local variable declaration

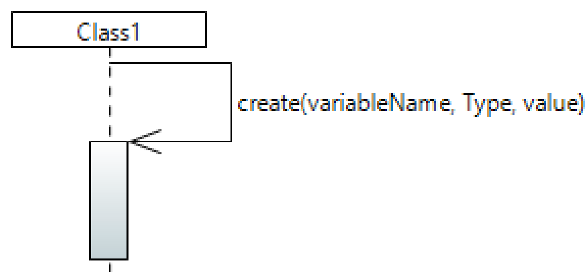

Figure 5.15: Regular local variable declaration with initialization

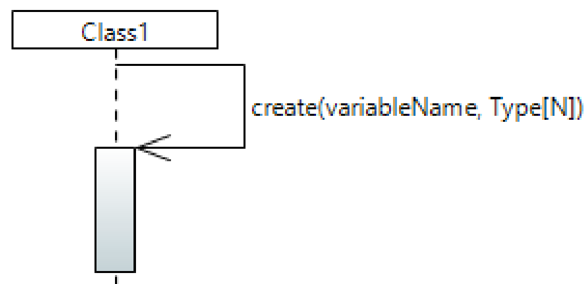

Figure 5.16: Local array declaration

Regarding the definition and usage of arrays, we already covered the definition as part of the class diagram *attributes* mapping and usage as part of the sequence diagram *assignment* mapping. Apart from that, arrays are used similarly as regular variables with a slight difference in specifying a position when writing to or reading from arrays, which is not the case with regular variables.

5.4 Detailed Mapping Procedure Description and Overview

After the detailed mapping procedure emerges, we find it important to present a summary, in the form of a table, for readability and understandability reasons. Such a representation of the mapping procedure should point to the most important segments while attempting to encompass the complete information behind the mapping. As part of this section, we created three tables. The first table (Table 5.1) is a simple textual representation of the comprehensive mapping procedure. This table has three main columns (Rebeca concept, UML diagram and UML concept)

and one identification column (ID). The identification column is used to identify a certain mapping row and it is used in the second and third table as well (to provide a reference to the particular mapping row in the first table and establish traceability). In the second table (Table 5.2), we identified four main columns (UML concept to Rebeca concept, Source UML, Target Rebeca and Additional semantics), and reference identification section (that connects the second table with the first). *Additional semantics* column provides additional information (including any additional semantics that are generated along with the mapping but not explicitly part of it, as for example containers). Considering that a big chunk of Rebeca semantics are reused from the JAVA programming language, especially the behavioral ones, we feel responsible to provide an additional table (Table 5.3), that is a subset of the second table (Table 5.2) and includes only the mappings with respect to the Rebeca-specific semantics.

This section combines a complete list of all semantics and their appropriate mapping pairs, that is described in Section 5.3 and provides an overview of the emerged comprehensive mapping procedure. Notably, in all provided tables, the mapping with respect to the Setter method (with table ID = 15), is marked with a star as it is a entirely theoretical proposition that is based on a JAVA programming paradigm and not grounded in Rebeca, just yet. That's why it is treated, in this thesis, as an optional semantic and is not included in any examples, that follow.

ID	Rebeca Concept	UML Diagram	UML Concept
1	Reactive class	Class	Class
2	State variable	Class	Attribute
3	Message server / local method / Constructor	Class	Operation
4	Known rebec	Class	Association / relation
5	Main / Rebec name	Object	Object name
6	Main / Rebec constructor argument	Object	Object argument
7	Main / Rebec known rebec argument	Object	Object relation
8	Conditional statement (if/else)	Sequence	Combined Fragment of type ALT
9	Conditional logical operators	Sequence	Combined Fragment of type ALT with logical operator
10	Conditional comparisons	Sequence	Combined Fragment of type ALT with conditional comparison
11	Call of message server of different class from initiating	Sequence	Message call representing the invocation of a method of a different class from the initiating
12	Call of message server or local method of the same class as initiating	Sequence	Message call representing the invocation of a method of the same class as the initiating
13	While loop	Sequence	Combined Fragment of Type Loop with the inclusion of only boolean condition (excluding range)
14	For loop	Sequence	Combined Fragment of Type Loop with the specification of the range (with and without iteration variable)
15*	Setter method	Sequence	Assignment of values to attributes (global)
16	Inline assignment	Sequence	Inline assignment of values to attributes (global and local)
17	Declaration of local variables	Sequence	Declaration of local attributes
18	Identification of message server or local method (container of the semantics from the sequence diagram)	Sequence	Initiating message call from generic sender

Table 5.1: Comprehensive mapping procedure - textual representation

Ref. ID	UML concept -> Rebeca concept	Source UML	Target Rebeca	Additional semantics
1	Class -> Reactive class	ClassName	reactiveclass ClassName (??queue Size??) {class body}	UML: ClassName[queuesize]
2	Regular attribute ->Regular state variable	attributeName:Type[1]	Type variableName;	Rebeca (container): statevars{}
2	Regular attribute with default value -> Regular state variable with assignment in the constructor	attributeName:Type[1] = value	Type variableName; PLUS assignment in the constructor: variableName = value;	Rebeca (container): statevars{}
2	Array attribute ->Array state variable	attributeName:Type[N]	Type [N] variableName;	Rebeca (container): statevars{}
3	Operation -> Message server	methodName (in parameterName: Type,..)	msgsrv MessageServerName(ExtType argument,...) {message server body}	UML: Operation has to be marked as public
3	Operation -> Local method (return)	methodName (in parameterName: Type,..) :Type	ExtType LocalMethodName (ExtType argument,..) {local method body with return statement}	UML: Operation has to be marked as private
3	Operation -> Local method (void)	methodName (in parameterName: Type,..)	void ExtType LocalMethodName (ExtType argument,..) {local method body}	UML: Operation has to be marked as private
3	Operation -> Constructor	methodName (in parameterName: Type,...)	ClassName (ExtType argument,...) {constructor body}	UML: Operation has to be marked as public and the name has to coincide with the Class name
3	Constructor operation argument -> Constructor argument and assignment to corresponding state variable	in parameterName: Type	ExtType argumentName PLUS assignment in the constructor: stateVariableName = argumentName;	UML: Constructor argument has to be named as follows: argumentName = stateVariableName + "Arg"
4	Association -> Known rebec	[associationCardinalityAnd Name]	reactiveClassName rebecName;	Rebeca (container): knownrebecs{}
5	Object name -> Main - rebec name	objectName: ClassName	ClassName objectName():();	Rebeca (container): main {}
6	Object argument -> Main - constructor argument	attributeName:Type[N] = argumentValue	ClassName objectName():(argumentValue);	Rebeca (container): main {}
7	Object relation -> Main - known rebec argument	[object1 in relation with object2]	ClassName objectName1(objectName2):(); ClassName objectName2(objectName1):();	Rebeca (container): main {}
8	Combined Fragment of type ALT -> Conditional statements (if/else)	[condition] and [else] (see Figure 5.1)	if (condition) {logic if condition is TRUE} else {logic if condition is false}	UML: Else is optional Can be defined as one. Interaction operand for if statement.
9	Combined Fragment of type ALT (logical operator) -> Conditional logical operator	[condition1] && [condition2] and [else] (see Figure 5.2)	if (condition1 && condition2) {logic if condition is TRUE} else {logic if any of two conditions is FALSE}	Other: logical OR (||), negation (!)
10	Combined Fragment of type ALT (conditional comparisons) -> Conditional comparisons	[variable == value] (see Figure 5.3	if (variable == value) {logic if condition is TRUE}	Other: inequality comparison (!=) and others (<, >, <=, >=)
11	Message call of method of a different class ->Call of message server of a different class	[Aync message call between two Action Execution Specifications of different classes]: messageName() (see Figure 5.5)	knownRebec.messageServerName();	UML: Target lifeline definition has to contain the object name along with the class name in the format: objectName: ClassName
12	Message call of method of the same class -> Call of message server of the same class	[Async message call within one Action Execution Specification of the same class]: messageName() (see Figure 5.6)	self.messageServerName();	N/A

12	Message call of method of the same class -> Call of local method of the same class	[Async message call within one Action Execution Specification of the same class]: messageName() (see Figure 5.6)	self.localMethodName();	N/A
13	Combined Fragment of Type Loop with only boolean condition (excluding range) -> While loop	[condition] (see Figure 5.7)	while (condition) {logic if condition is TRUE}	N/A
14	Combined Fragment of Type Loop with the specification of range (excluding explicit iteration variable) -> For loop	[startRangeValue, endRangeValue] (see Figure 5.8)	for (int i = startRangeValue; i <endRangeValue; i++) {logic if condition is TRUE}	N/A
14	Combined Fragment of Type Loop with the specification of range and iteration variable -> For loop	[startRangeValue, endRangeValue, iterationVariableName] (see Figure 5.9)	for (int iterationVariableName = startRangeValue; iterationVariableName < endRangeValue; iterationVariableName++) {logic if condition is TRUE}	N/A
15*	Assignment of values to attributes (global) -> Setter method	setStateVariableName (value) (see Figure 5.10)	self.setStateVariableName(value);	Rebeca: Auto-generated setter along with the state variable in the form: void setStateVariableName (ExtType value) { stateVariableName = value; }
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (regular assignment)	set(variable, value) (see Figure 5.11)	variable = value;	Instead of a value we can also pass a casted value as well as non-deterministic expression. Same applies for all inline assignment variations
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (Array assignment at position)	set(array[position], value) (see Figure 5.12)	array[position] = value;	Regular assignment is possible with arrays but than one array is assigned to the other
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (assignment with specified assignment operator)	set(variable, value, assignmentOperator) (see Figure 5.13)	variable assignmentOperator value;	Operators: = (default), +=, -=, *=, /=, %=
17	Declaration of local attributes ->Declaration of local variables (regular local variable without initialization)	declare(variableName, Type) (see Figure 5.14)	ExtType variableName;	N/A
17	Declaration of local attributes ->Declaration of local variables (regular local variable with initialization)	declare(variableName, Type, value) (see Figure 5.15)	ExtType variableName = value;	Instead of a value we can also pass a casted value as well as non-deterministic expression.
17	Declaration of local attributes ->Declaration of local variables (local arrays)	declare(variableName, Type[N]) (see Figure 5.16)	Type [N] variableName;	Initialization of arrays, along with declaration is possible, identically as for regular local variables with initialization
18	Initiating message call from generic sender ->Identification of message server or local method	[Async initiating message call between two Action Execution Specification]: messageName() (see Figure 5.4)	No target of the mapping.	UML: Name sender has to be static. Rebeca: It is used for locating the message server or local method (container of the semantics from the sequence diagram)

Table 5.2: Comprehensive mapping procedure - detailed conceptual representation

Ref. ID	UML concept -> Rebeca concept	Source UML	Target Rebeca	Additional semantics
1	Class -> Reactive class	ClassName	reactiveclass ClassName (??queue Size??) {class body}	UML: ClassName[queuesize]
2	Regular attribute ->Regular state variable	attributeName:Type[1]	Type variableName;	Rebeca (container): statevars{}
2	Regular attribute with default value -> Regular state variable with assignment in the constructor	attributeName:Type[1] = value	Type variableName; PLUS assignment in the constructor: variableName = value;	Rebeca (container): statevars{}
2	Array attribute ->Array state variable	attributeName:Type[N]	Type [N] variableName;	Rebeca (container): statevars{}
3	Operation -> Message server	methodName (in parameterName: Type,..)	msgsrv MessageServerName(ExtType argument,...) {message server body}	UML: Operation has to be marked as public
4	Association -> Known rebec	[associationCardinalityAnd Name]	reactiveClassName rebecName;	Rebeca (container): knownrebecs{}
5	Object name -> Main - rebec name	objectName: ClassName	ClassName objectName():();	Rebeca (container): main {}
6	Object argument -> Main - constructor argument	attributeName:Type[N] = argumentValue	ClassName objectName():(argumentValue);	Rebeca (container): main {}
7	Object relation -> Main - known rebec argument	[object1 in relation with object2]	ClassName objectName1(objectName2):(); ClassName objectName2(objectName1):();	Rebeca (container): main {}
11	Message call of method of a different class ->Call of message server of a different class	[Aync message call between two Action Execution Specifications of different classes]: messageName() (see Figure 5.5)	knownRebec.messageServerName();	UML: Target lifeline definition has to contain the object name along with the class name in the format: objectName: ClassName
12	Message call of method of the same class -> Call of message server of the same class	[Async message call within one Action Execution Specification of the same class]: messageName() (see Figure 5.6)	self.messageServerName();	N/A
12	Message call of method of the same class -> Call of local method of the same class	[Async message call within one Action Execution Specification of the same class]: messageName() (see Figure 5.6)	self.localMethodName();	N/A

Table 5.3: Comprehensive mapping procedure - Rebeca-centric concepts

As can be seen in Table 5.2 and 5.3, a detailed mapping procedure (semantic level of detail), focusing on the minimal types of UML diagrams for accomplishing comprehensive results, is displayed. Moreover, each row in these tables can be traced back, based on the reference identification number, to the Table 5.1, that represents a brief textual representation of the mapping. Considering the similarities between Rebeca and JAVA, we provided an additional table (Table 5.3), that only includes the semantics specific to Rebeca language. The provided mapping procedure gives an answer to the two of the three proposed research questions and those include:

1. **RQ1:** What are the minimum UML diagrams required to serve as sources for target Rebeca concepts in order to obtain something meaningful to be analyzed through Rebeca?
   (a) **Answer:** The minimum UML diagrams necessary for establishing a detailed enough mapping procedure that ensures a meaningful Rebeca infor-

mation, include Class diagram and Object diagram as structural diagrams and Sequence diagram as behavioral (see Table 5.1).

2. **RQ2:** What is the adequate mapping procedure between the identified UML concepts and Rebeca concepts?

    (a) **Answer:** This cannot be simply answered in a few words, as the answer is essentially conveyed in a complete emerged mapping procedure (Section 5.3 and Tables 5.2 5.3).

6 Method Evaluation

The validation phase is an important step in the whole story, as it strives to establish confidence in the developed mapping procedure (through applicability scenarios in post-conversion validation) and avoid performing the transformation and reaching faulty Rebeca models if the design is not compatible with the mapping procedure (in pre-conversion validation).
Initially, as part of the pre-conversion validation, we need to set some rules or identify the correctness attributes that will be used as a merit of correctness/compatibility in the process of evaluating the source UML models, before conversion to Rebeca takes place. Hence, these correctness attributes are inherently used for establishing that the models are translatable to Rebeca.
Moreover, the necessity for the evaluation of the mapping procedure is crucial for establishing confidence and identifying its applicability scope. This section gives an answer to the last research question:

1. **RQ3:** What is the applicability of the proposed mapping procedure and its proper substantiation?

6.1 Pre-Conversion Validation

The basis for pre-conversion validation is grounded on the definition of correctness attributes (or correctness rules) for establishing the correctness/compatibility of source UML models. Here, we need to consider semantic correctness regarding Rebeca concepts and what is achievable in Rebeca. Also, closely related merit of correctness is design correctness with respect to the design pattern used to convey the information within diagrams. Nevertheless, we include this category as part of the semantic correctness, as it also deals with boundaries of Rebeca language and therefore can be attributed to its semantics. Considering we already established a comprehensive mapping procedure, that includes detailed semantics of the UML appropriate for target Rebeca models, we will indeed use these UML semantics as the main objects for the creation of such correctness attributes (Table 5.2). Hence, these correctness attributes can be seen as the approximation of the models within the scope of the defined mapping procedure and its boundaries.
As the result of the identification of such correctness attributes, we strive to provide definite correctness rules that convey the design pattern for modeling of the considered UML diagrams within the scope of the proposed detailed mapping procedure and its semantics.
To capture the correctness rules in a readable and traceable manner, we provide a table (Table 6.1) in which for each applicable semantic in the Table 5.2 we document a correctness rule associated with it. This new table shares the same first two columns of the associated Table 5.2, including both Reference ID and UML concept to Rebeca concept, supplied with an additional column Correctness attribute whose purpose is self-explanatory.
Additionally, any semantics that are not covered by the mapping procedure are excluded by definition. This basically means that they need to be filtered out by the

pre-conversion validation or approximated within the scope of the mapping procedure (and included semantics), if applicable. Ideally, the source UML semantics should not exceed the defined semantics in the mapping procedure. Hence, these defined semantics governs the applicability scope that is discussed further in Section 6.2 on a practical example.

Ref. ID	**UML concept -> Rebeca concept**	**Correctness attribute**
1	Class -> reactive class	Queue length has to be defined along with the class name: ClassName[queueLength]
2	Regular attribute ->regular state variable	The regular attribute has to be defined with constraint [1], after a definition of the type, that symbolizes the singular value attribute. It can also be defined without any constraint after the definition of type. The type of attribute has to be compatible with Rebeca's type (excluding ExtType).
2	Array attribute ->array state variable	The array attribute has to be defined with constraint [N], after a definition of the type, where N is greater than 1 and symbolizes the array. The type of attribute has to be compatible with Rebeca's type (excluding ExtType).
3	Operation -> message server	Operation (representing message server) has to be marked as public and it cannot have a return statement as it is void by definition.
3	Operation -> local method (return)	Operation (representing local method with return statement) has to be marked as private.
3	Operation -> local method (void)	Operation (representing void local method) has to be marked as private.
3	Operation -> constructor	Constructor operation has to be marked as public and it cannot have a return as it is void by definition. The name has to coincide with the Class name. A constructor has to be included in Rebeca, in at least one of the reactive classes, as it serves its purpose being the initial message server (containing initial message call), that sets things going. Additionally, it is used to initialize state variables.
3	Constructor operation argument -> constructor argument and assignment to corresponding state variable	Constructor argument has to be named as follows: argumentName = stateVariableName + Arg If constructor contains any additional behavioral logic (i.e. message calls, etc.), then the sequence diagram has to be constructed for the corresponding constructor, excluding the initialization of state variables with arguments.
4	Association -> Known rebec	Known rebecs defined by associations should match the known rebecs defined in the sequence diagram and the known rebecs defined by relations in the object diagram.
6	Object argument -> Main - constructor argument	Each argument has to be referenced to the corresponding attribute in the class diagram and the value has to be specified.
7	Object relation -> Main - known rebec argument	Known rebecs defined by relations should match the known rebecs defined in the sequence diagram and the known rebecs defined by associations in the class diagram.

8	Combined Fragment of type ALT -> conditional statements (if/else)	Before the condition statement inside the first Interaction Operand, we should not specify any keyword. It is by definition accepted that the first interaction operand in ALT fragment refers to the if statement. We only specify a condition. The second ALT fragment, by default, takes the word else and no additional alterations should be made in the else. Also, else is optional and we can have one Interaction operand representing specification of if statement. A condition is built as a combination of four paradigms including variable, value, conditional logical operator, conditional comparison.
9	Combined Fragment of type ALT (logical operator) -> conditional logical operator	Conditional logical operators have to be one of the following: logical AND (&&), logical OR (||), negation (!). The first two have to be placed between two conditions while the last is placed before condition or group of conditions to negate their value.
10	Combined Fragment of type ALT (conditional comparisons) -> conditional comparisons	Conditional comparison operators have to be one of the following: equality comparison (==), inequality comparison (!=) or following operators (<, >, <=, >=).
11	Message call of method of a different class -> call of message server of a different class	The source lifeline has to be defined as a class so that in the lifeline name it contains only the class name. The target lifeline has to be defined as an object of a specific class so that in the lifeline name it contains both name of the class and name of the object, as follows: objectName:ClassName
12	Message call of method of the same class -> call of message server of the same class	The message call has to be modeled with a recursive arrow or a message call towards the same class as initiating.
12	Message call of method of the same class -> call of local method of the same class	The message call has to be modeled with a recursive arrow or a message call towards the same class as initiating.
13	Combined Fragment of Type Loop with only boolean condition (excluding range) -> While loop	Any conditional statement can be modeled in a while loop. But, keep in mind, that certain conditions may result in the infinite loop (refers commonly to static condition clauses but also certain dynamic clauses).
14	Combined Fragment of Type Loop with the specification of range (excluding explicit iteration variable) -> For loop	The comma-separated start and end range value has to be specified. Considering that the iteration variable is not specified, then the default name (i) is used. In this case, it is crucial not to use the variable with the same name inside the loop (to avoid overriding its value). Also, when multiple conditions are nested, the name of the iteration variable is incrementally created, so that inside each nested loop except the first, iteration variable gets an incremented number to its name (i.e. i, i2, i3, i4 etc.).
14	Combined Fragment of Type Loop with the specification of range and iteration variable -> For loop	The specified name of the iteration variable should not be used to define variables inside the loop (including other iteration variables of nested loops) to avoid overriding its value.

15*	Assignment of values to attributes (global) -> Setter method	There is no explicit information to be provided on the UML side for the creation of setter methods as they are automatically created when mapping of the state variables is performed. Invoking the setter method or initiating a message call has to be done by asynchronous call to the same class, with the name of the setter method call as follows: setStateVariableName(value)
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (regular assignment)	Inline regular assignment has to be defined in the form, as follows: set(variable, value) This has to be defined as a message call to the same class as initiating. Instead of a value, we can also pass a casted value (in the form: *(CastType) value*) as well as non-deterministic expression (in the form: *?(value1, value2,.. valueN)*). Same applies for all inline assignment variations.
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (Array assignment at position)	Inline array assignment at the specified position has to be defined in the form, as follows: set(array[position],value) This has to be defined as a message call to the same class as initiating. Regular assignment is possible here but then an array has to be passed as value.
16	Inline assignment of values to attributes (local and global) -> Inline assignment of value (assignment with specified assignment operator)	Inline assignment with the specified operator has to be defined in the form, as follows: set(variable, value, assignmentOperator) Operators has to coincide with: = (default), +=, -=, *=, /=, %=. This has to be defined as a message call to the same class as initiating.
17	Declaration of local attributes ->Declaration of local variables (regular local variables without initialization)	Declaration of regular local variable has to be defined in the form, as follows: declare(variableName, Type) This has to be defined as a message call to the same class as initiating.
17	Declaration of local attributes ->Declaration of local variables (regular local variable with initialization	Declaration of regular local variable with initialization has to be defined in the form, as follows: declare(variableName, Type, value) This has to be defined as a message call to the same class as initiating. Instead of a value, we can also pass a casted value (in the form: *(CastType) value*) as well as non-deterministic expression (in the form: *?(value1, value2,.. valueN)*).
17	Declaration of local attributes ->Declaration of local variables (local arrays)	Declaration of local array has to be defined in the form, as follows: declare(variableName, Type[N]) This has to be defined as a message call to the same class as initiating. Declaration of local array with value assignment is achieved in the same manner as for regular local variables. However, in this case, an array is expected as the value.
18	Initiating message call from generic sender -> Identification of message server or local method	Name of the initiating lifeline (source of the message) has to be sender (it is static and unchangeable). The name of the message call and its arguments has to coincide with the operation (in the class diagram), that is modeled. The target lifeline has to be defined as a class so that in the lifeline name it contains only the class name.

Table 6.1: Pre-conversion validation correctness rules

The provided Table 6.1, contains a detailed list of correctness attributes for each mapping element. Against these rules, source UML models have to be assessed in order to establish their correctness and ultimately proceed to the next stage where actual transformation is conducted. The objective of the pre-conversion validation is the case in which the source UML models coincide with the correctness attributes. The practical example, upon which we do the transformation based on the mapping procedure, in post-conversion validation is validated by these rules, until all of them successfully passed, before transformation was done.

6.2 Post-Conversion Validation

After the detailed mapping procedure is created, the necessity for its validation on applicability scenarios is obvious. Notably, the process in which the transformation is done and target models acquired and validated can be done differently depending on the available resources (i.e. time) and other factors.
The post-conversion validation is organized as a manual transformation of source UML models to Rebeca models that is performed by two subjects on a uniform example. We are investigating the possibility for variations in the target Rebeca models, although we expect the variations to be minor. From our understanding, the pre-conversion validation should equalize the source UML models and significantly reduce the possibility for different interpretations. This is expected to propagate and eliminate the variations in target Rebeca models. Hence, the examples are given validated by the correctness attributes of the pre-conversion validation. The subjects involved were responsible for performing the manual transformation by following the principles of the mapping procedure and obtaining the target Rebeca models. These models are then inspected to check if they reflect the source UML models and identify any defects or unintended behaviors. Also, we check for variations in two comparative model transformations. The subjects had no prior knowledge of Rebeca which we perceive as a good thing because they have to rely entirely on the mapping procedure to do the transformation, therefore reducing the effect of the previous knowledge. However, we could argue that the general programming knowledge in similar languages to Rebeca, could potentially influence the results. It is important to note that this study, conducted with the help of external subjects, is not empirically structured and this could affect negatively the replicability and validity of the results. This is caused by the lack of resources (i.e. time, number of involved subjects, etc.).
In addition to this example, we also provide another example that shows how complex behavioral logic can be captured within the sequence diagram and used for the complete transformation by applying the mapping procedure. This example, however, is performed independently without the involvement of external subjects. This section is organized into four subsections. In the first subsection, we provide the validated source UML models as an example on which the transformation is done. In the second, we document detailed transformation and reflect upon the acquired target Rebeca models. In the third, we provide an additional example, focused on complex behavioral logic. Finally, in the fourth, we discuss the applicability of the mapping procedure and potential improvements.

6.2.1 Practical Example - Validated Source Models

In this section, we provide validated source UML models, that are used for manual transformation, handled independently by the subjects. As aforementioned, these models are provided validated by the correctness rules of the pre-conversion validation with reasons to establish the common ground for both independent transformations. This enables us to perform precise inspections in the acquired target models and better understand the source of any possible defects and variations.
The given train-controller example represents a two-way railroad with a single-track bridge in between, upon which the access is controlled by traffic lights, managed by a controller. For modeling, we used a Class diagram, Object diagram and Sequence diagram considering that all three are required for detailed transformation, in order to capture both structure and behavior in the models.
Structure is captured within Figures 6.1 and 6.2 whereas behavior is captured within Figures 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8. The structural semantics (from class and object diagrams) are used for establishment of the skeleton application while behavioral semantics (from sequence diagrams) introduce an additional layer of logic, resulting in the meaningful target models or set of Rebeca semantics to be obtained.

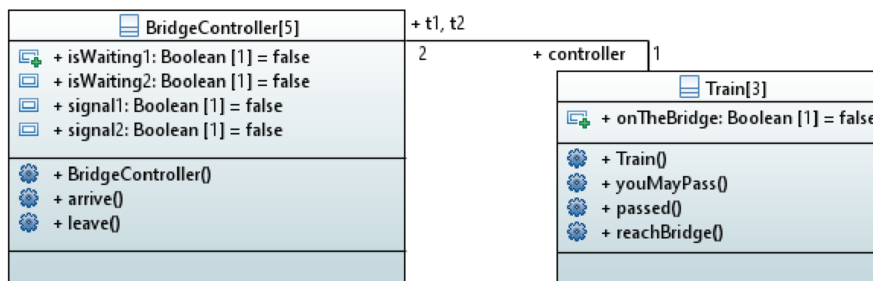

Figure 6.1: Class diagram - structural source

Figure 6.2: Object diagram - structural source

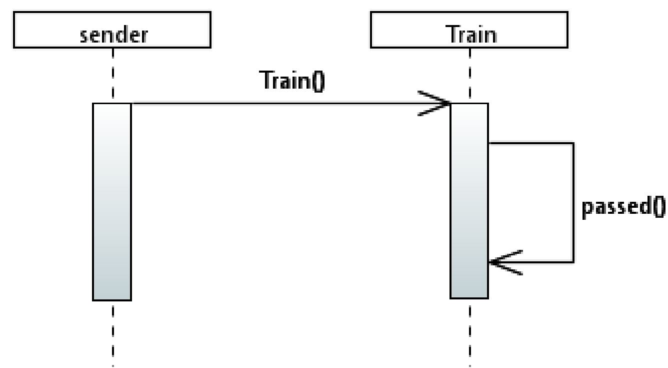

Figure 6.3: Sequence diagram - Train constructor - behavioral source

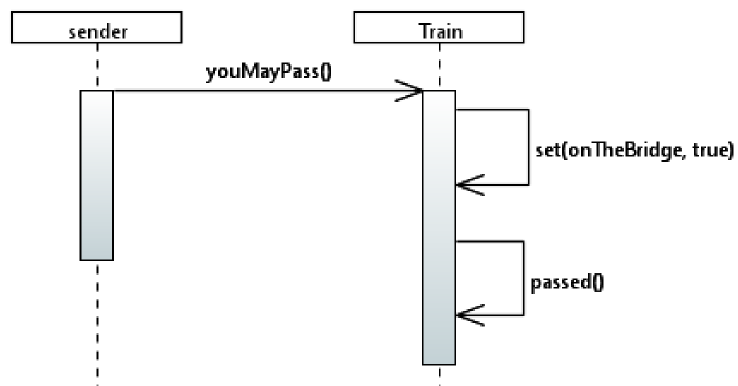

Figure 6.4: Sequence diagram - Train youMayPass method - behavioral source

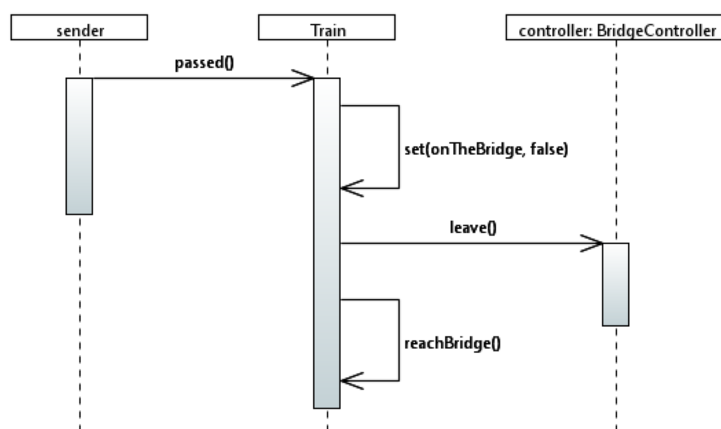

Figure 6.5: Sequence diagram - Train passed method - behavioral source

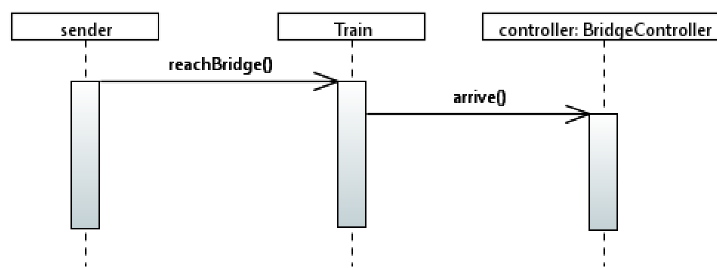

Figure 6.6: Sequence diagram - Train reachBridge method - behavioral source

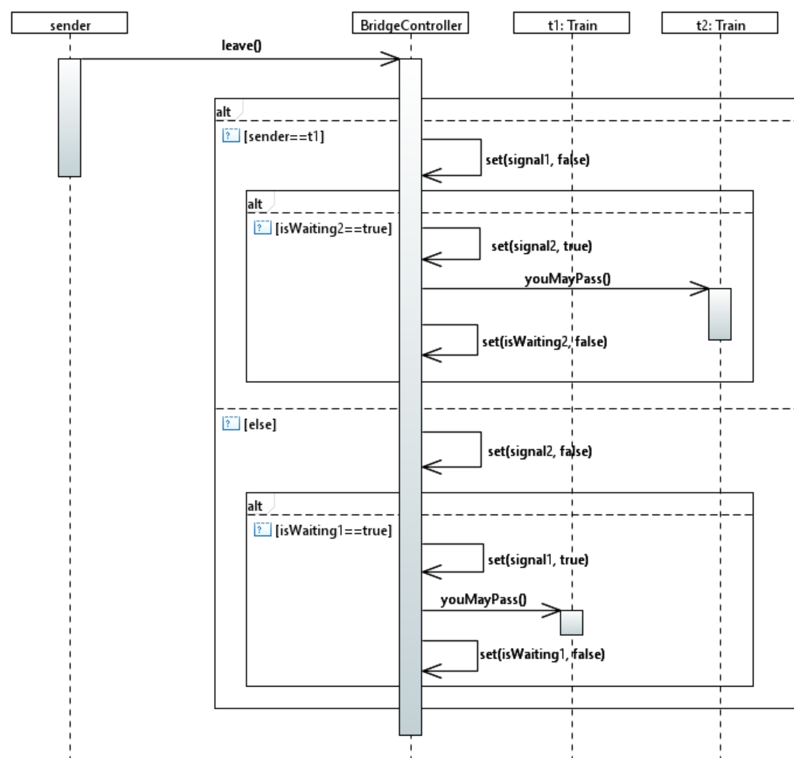

Figure 6.7: Sequence diagram - BridgeController leave method - behavioral source

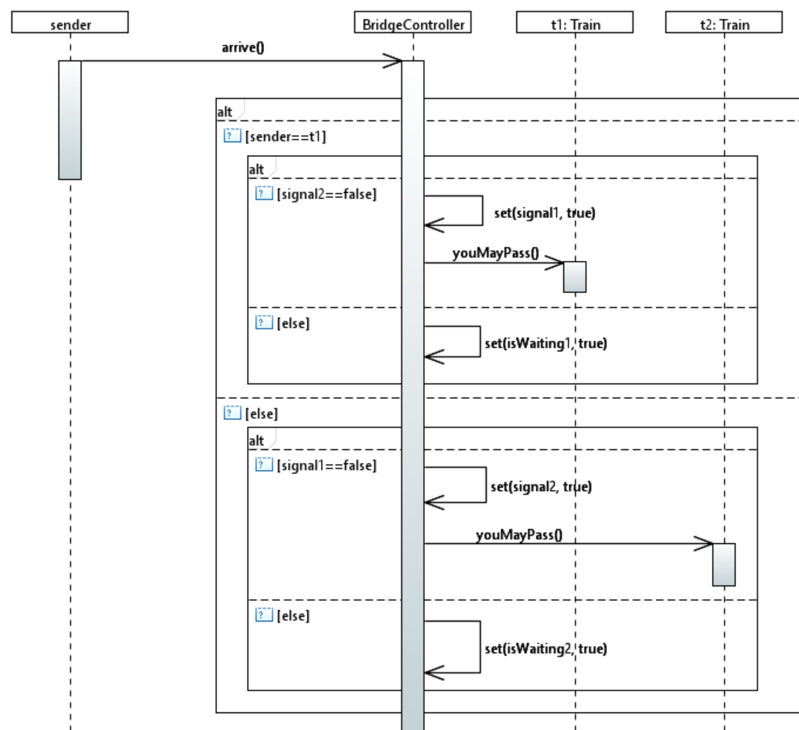

Figure 6.8: Sequence diagram - BridgeController arrive method - behavioral source

6.2.2 Manual Transformation - Acquiring Target Models

The two subjects performed the transformation on the given example (Section 6.2.1), based on the guidelines of the proposed detailed mapping procedure. The results (obtained Rebeca models) by two subjects are deemed as significantly similar as the only differences between them are minor syntactical mistakes caused by typing errors in the names of the message servers, constructors, etc.. Additionally, one of the subjects did not include the queue sizes as part of the reactive class declarations, while the other misinterpreted the message calls, representing inline assignment, as method invocations and performed initialization of a state variable inside the *statevars* container instead of constructor, even though appropriate documentation for all of these cases is included as part of the mapping procedure. Notably, these variations are insignificant in terms of the validity of the proposed mapping procedure and can be easily eliminated by the automated model transformation. We provide generated Rebeca models as two Listings 2 and 3, that portray the models transformed by two independent subjects (marked as Subject 1 and Subject 2). The minor syntactic mistakes in both transformations are marked with orange color while other, more serious, issues are marked with red. The last Listing 4 represents the defect-free transformation. As follows:

```
reactiveclass BridgeController {
  knownrebecs {
    Train t1;
    Train t2;
  }
  statevrs {
```

```
    boolean isWaiting1;
    boolean isWaiting2;
    boolean signal1;
    boolean signal2;
  }
  BridgeController() {
    isWaiting1 = false;
    isWaiting2 = false;
    signal1 = false;
    signal2 = false;
  }
  msgsrv Arrive() {
    if (sender == t1) {
      if (signal2 == false) {
        signal1 = true;
        t1.youMayPass();
      }
      else {
        isWaiting1 = true;
      }
    }
    else {
      if (signal1 == false) {
        signal2 = true;
        t2.youMayPass();
      }
      else{
        isWaiting2 = true;
      }
    }
  }
  msgsrv Leave() {
    if (sender == t1) {
      signal1 = false;
      if (isWaiting2 == true) {
        signal2 = true;
        t2.youMayPass();
        isWaiting2 = false;
      }
    } else {
      signal2 = false;
      if (isWaiting1 == true) {
        signal1 = true;
        t1.youMayPass();
        isWaiting1 = false;
      }
    }
  }
}
reactiveclass Train {
  knownrebecs {
    BridgeController controller;
  }
  statevrs {
    boolean onTheBridge;
  }
```

```
  Train() {
    onTheBridge = false;
    self.passed();
  }
  msgsrv youMayPass() {
    onTheBridge = true;
    self.passed();
  }
  msgsrv passed() {
    onTheBridge = false;
    controller.leave();
    self.reachBridge();
  }
  msgsrv reachBridge() {
    controller.Arrive();
  }
}
main {
  Train t1(controller):();
  BridgeController controller(t1, t2):();
  Train t2(controller):();
}
```

Listing 2: Subject 1 - obtained Rebeca models

```
reactiveclass BridgeController(5) {
  knownrebecs {
    Train t1;
    Train t2;
  }
  statevars {
    boolean isWaiting1;
    boolean isWaiting2;
    boolean signal1;
    boolean signal2;
  }
  BridgeController() {
    isWaiting1 = false;
    isWaiting2 = false;
    signal1 = false;
    signal2 = false;
  }
  msgsrv arrive() {
    if (sender == t1) {
      if (signal2 == false) {
         set(signal1, true);
        t1.youMayPass();
      }
      else {
        set(isWaiting1, true);
      }
    }
    else {
      if (signal1 == false) {
        set(signal2, true);
```

```
        t2.youMayPass();
      }
      else{
            set(isWaiting2, true);
      }
    }
  }
  msgsrv leave() {
    if (sender == t1) {
      set(signal1, false);
      if (isWaiting2 == true) {
        set(signal2, true);
        t2.youMayPass();
        set(isWaiting2, false);
      }
    } else {
      set(signal2, false);
      if (isWaiting1 == true) {
        set(signal1, true);
        t1.youMayPass();
        set(isWaiting1, false);
      }
    }
  }
}
reactiveclass Train(3) {
  knownrebecs {
    BridgeController controller;
  }
  statevars {
    boolean onTheBridge = false;
  }
  Train() {
    self.passed();
  }
  msgsrv youMayPass() {
    set(onTheBridge, true);
    self.passed();
  }
  msgsrv passed() {
    set(onTheBridge, false);
    controller.leave();
    self.reachBridge();
  }
  msgsrv reachBridge() {
    controller.arrive();
  }
}
main {
  Train t1(controller):();
  Train t2(controller):();
  BridgeController controller(t1, t2):();
}
```

Listing 3: Subject 2 - obtained Rebeca models

```
reactiveclass BridgeController(5) {
  knownrebecs {
    Train t1;
    Train t2;
  }
  statevars {
    boolean isWaiting1;
    boolean isWaiting2;
    boolean signal1;
    boolean signal2;
  }
  BridgeController() {
    signal1 = false;
    signal2 = false;
    isWaiting1 = false;
    isWaiting2 = false;
  }
  msgsrv arrive() {
    if (sender == t1) {
      if (signal2 == false) {
        signal1 = true;
        t1.youMayPass();
      }
      else { isWaiting1 = true; }
    }
    else {
      if (signal1 == false) {
        signal2 = true;
        t2.youMayPass();
      }
      else{ isWaiting2 = true; }
    }
  }
  msgsrv leave() {
    if (sender == t1) {
      signal1 = false;
      if (isWaiting2 == true) {
        signal2 = true;
        t2.youMayPass();
        isWaiting2 = false;
      }
    } else {
      signal2 = false;
      if (isWaiting1 == true) {
        signal1 = true;
        t1.youMayPass();
        isWaiting1 = false;
      }
    }
  }
}

reactiveclass Train(3) {
  knownrebecs {
    BridgeController controller;
  }
```

```
  statevars {
    boolean onTheBridge;
  }
  Train() {
    onTheBridge = false;
    self.passed(); }
  msgsrv youMayPass() {
    onTheBridge = true;
    self.passed(); }
  msgsrv passed() {
    onTheBridge = false;
    controller.leave();
    self.reachBridge(); }
  msgsrv reachBridge() {
    controller.arrive(); }
}

main {
  Train t1(controller):();
  Train t2(controller):();
  BridgeController controller(t1, t2):();
}
```

Listing 4: Defect-free transformation Rebeca models

6.2.3 Capturing Rich Behavioral Concepts - Example

Besides the presented example, we also provide an additional example, to further substantiate the mapping procedure. The example represents a commit problem where entities (marked as nodes) are expected to commit upon performing an action. Moreover, if any of the nodes disagrees then the action is aborted which implies *AND* logical operator between the commits. This problem is solved in a way that one of the nodes is a listener and others are emitters. The listener collects all the messages (commits and aborts) from other nodes as well as its own message. In case if any abort message is received, then the final message that is sent to all nodes is an abort message. Otherwise, the final message to be sent is a commit message. This example is more complex than the previous and is therefore used to show how complex behavioral logic can be modeled using the proposed mapping procedure. However, external subjects were not involved as part of the transformation, applied on the provided UML models. Hence, we only provide the manual transformation that is done independently. Additionally, as in the previous example, we provide pre-validated source UML models. As follows:

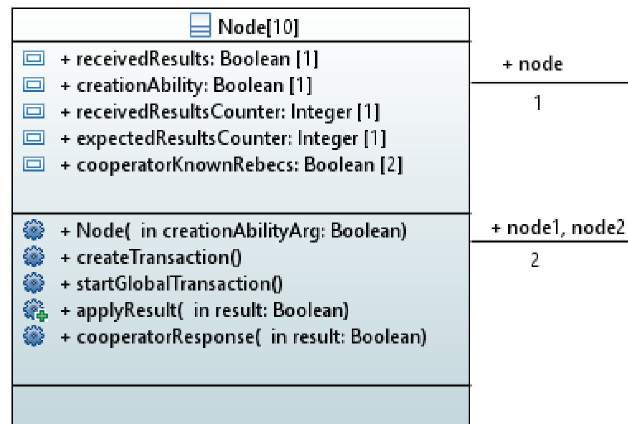

Figure 6.9: Class diagram - Node class - structure

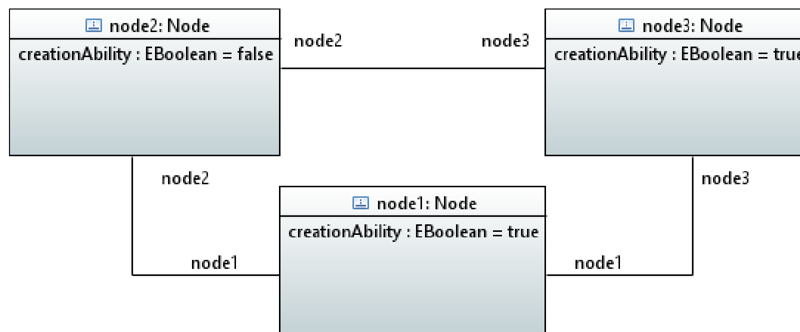

Figure 6.10: Object diagram - Node instances - structure

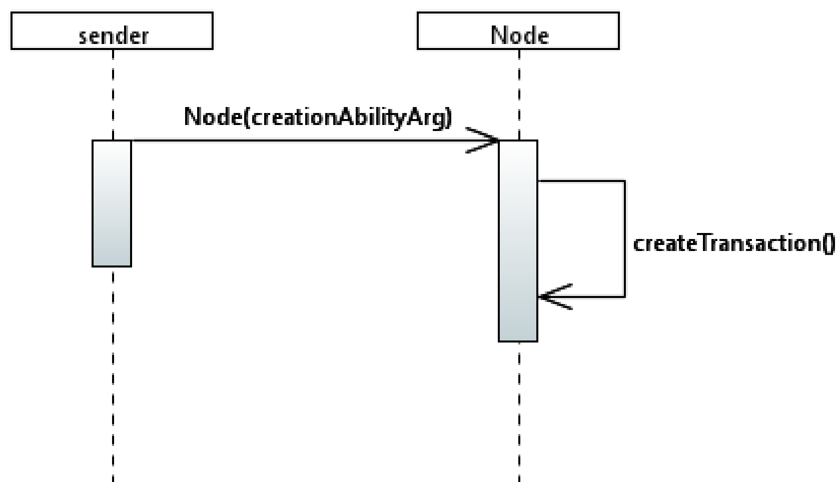

Figure 6.11: Sequence diagram - Node constructor - behavior

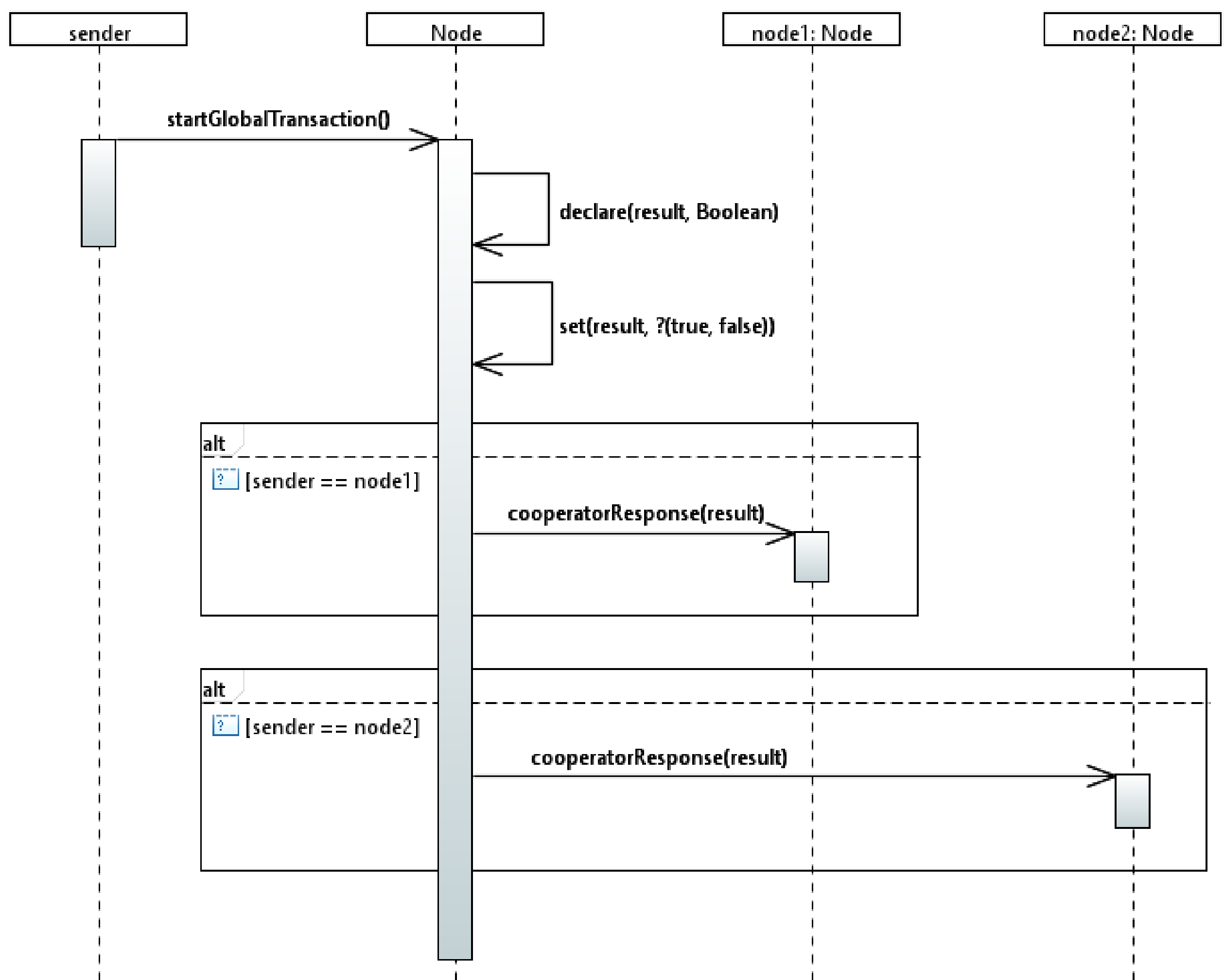

Figure 6.12: Sequence diagram - Start global transaction method - behavior

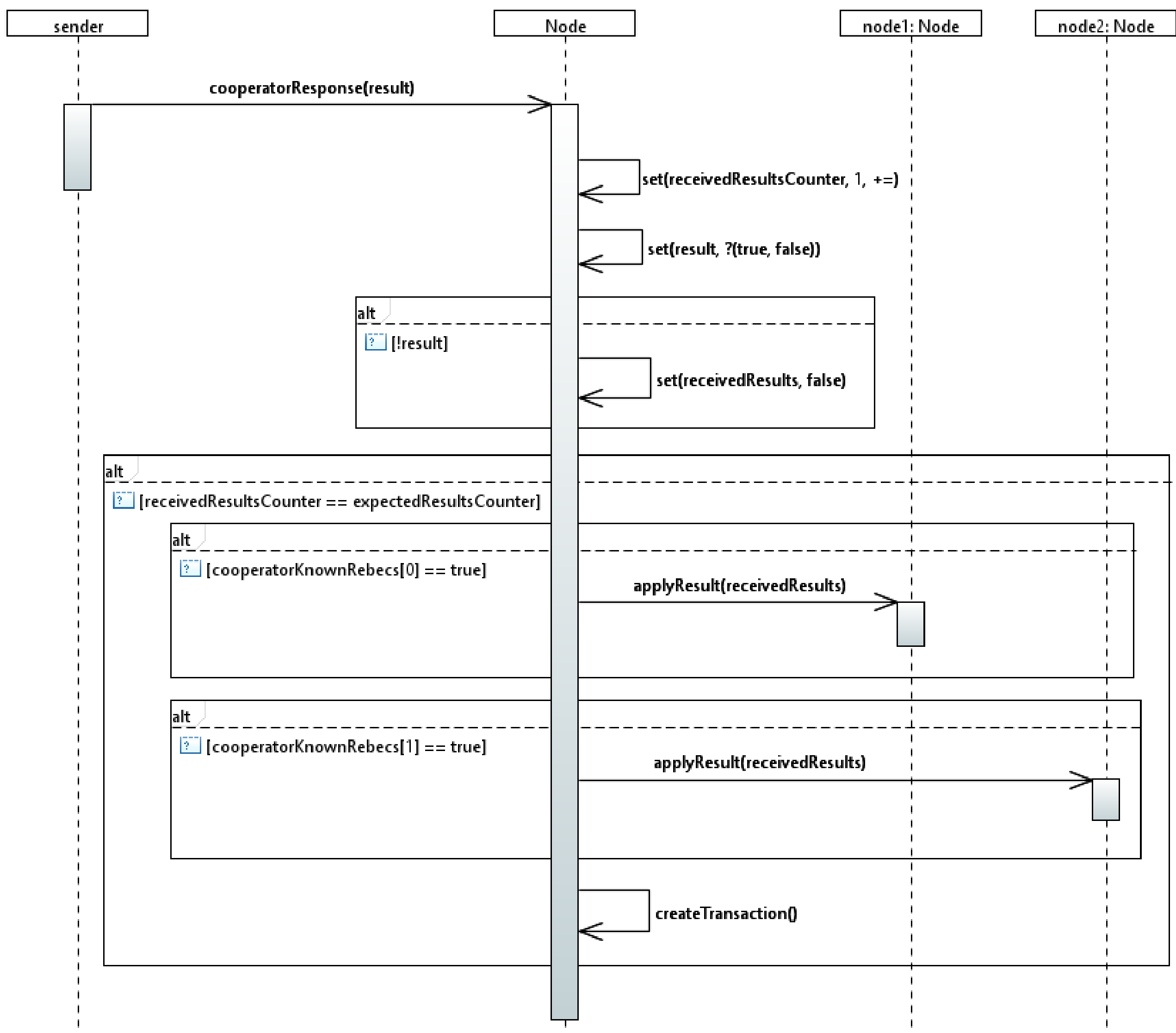

Figure 6.13: Sequence diagram - Cooperator response method - behavior

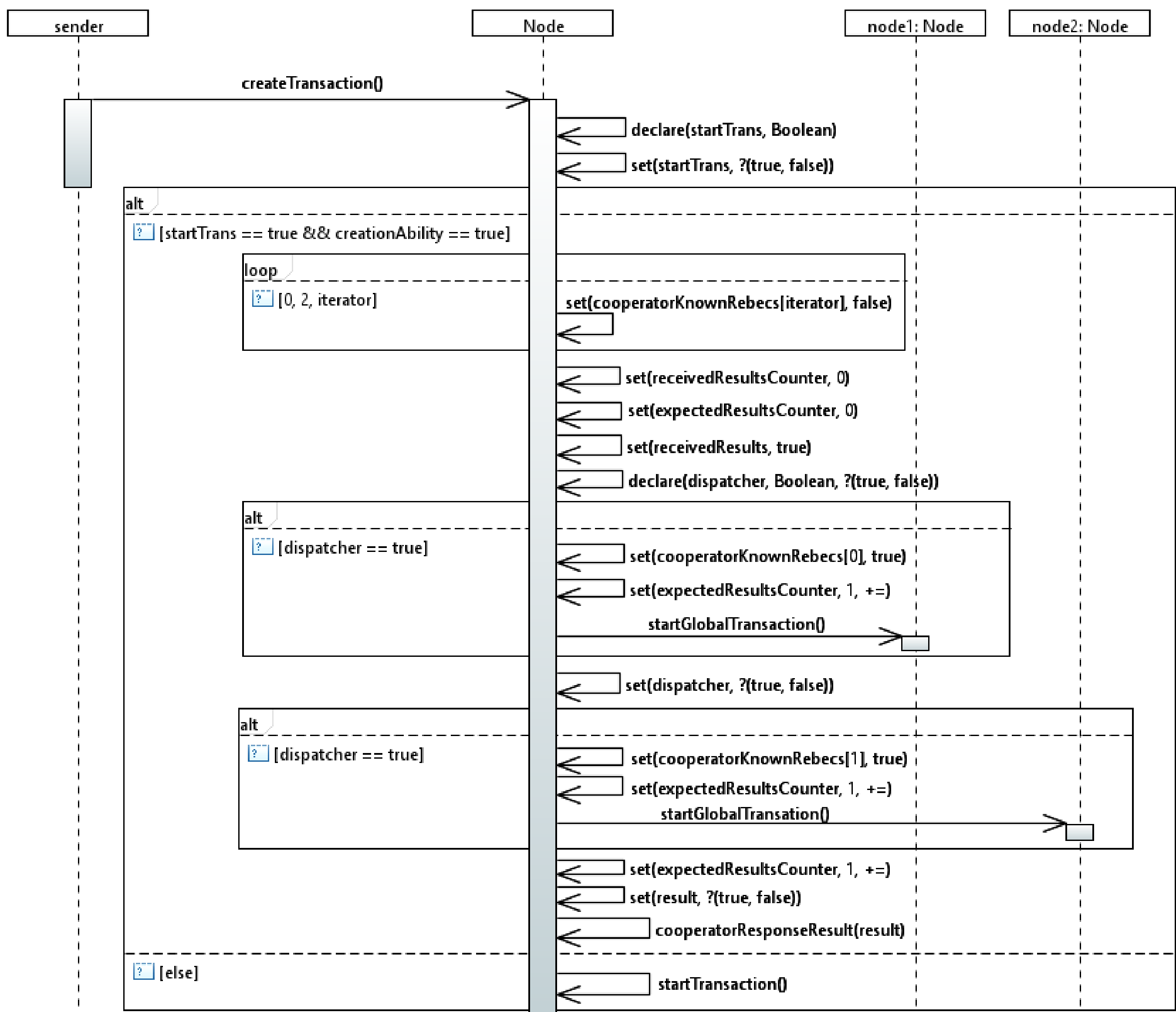

Figure 6.14: Sequence diagram - Create transaction method - behavior

We provide generated Rebeca models as part of the Listing 5. As follows:

```
reactiveclass Node(10) {

  knownrebecs {
      Node node1;
      Node node2;
  }

  statevars {
    boolean receivedResults;
    boolean creationAbility;
    int receivedResultsCounter;
    int expectedResultsCounter;
    boolean[2] cooperatorKnownRebecs;
```

```
}

Node(boolean nodeCreationAbilityArg) {
  creationAbility = nodeCreationAbilityArg;
  self.createTransaction();
}

msgsrv createTransaction() {
  boolean startTrans;
  startTrans = ?(true, false);
  if (startTrans == true && creationAbility == true) {

    int iterator;
    for(iterator = 0; iterator < 2; iterator = iterator + 1){
      cooperatorKnownRebecs[iterator] = false;
    }

    receivedResultsCounter = 0;
    expectedResultsCounter = 0;
    receivedResults = true;
    boolean dispatcher = ?(true, false);

    if (dispatcher == true) {
      cooperatorKnownRebecs[0] = true;
      expectedResultsCounter += 1;
      node1.startGlobalTransaction();
    }

    dispatcher = ?(true, false);
    if (dispatcher == true) {
      cooperatorKnownRebecs[1] = true;
      expectedResultsCounter += 1;
      node2.startGlobalTransaction();
    }

    boolean result;
    expectedResultsCounter += 1;
    result = ?(true, false);
    self.cooperatorResponse(result);
  }
  else {
    self.createTransaction();
  }
}

msgsrv startGlobalTransaction() {
  boolean result;
  result = ?(true, false);

  if (sender == node1) {
    node1.cooperatorResponse(result);
  }
  if (sender == node2) {
    node2.cooperatorResponse(result);
      }
}
```

```
  msgsrv cooperatorResponse(boolean result) {
    receivedResultsCounter += 1;
    if (!result){
      receivedResults = false;
    }

    if (receivedResultsCounter == expectedResultsCounter) {
      if (cooperatorKnownRebecs[0] == true) {
        node1.applyResult(receivedResults);
      }
      if (cooperatorKnownRebecs[1] == true) {
        node2.applyResult(receivedResults);
      }

      self.createTransaction();
    }
  }

  msgsrv applyResult(boolean result) {

  }

}

main {
  Node node1(node2, node3):(true);
  Node node2(node3, node1):(false);
  Node node3(node1, node2):(true);
}
```

Listing 5: Complex example with rich behavioral logic

6.2.4 Results, Applicability and Potential Improvements of Mapping Procedure

From the given results, in the form of transformed models, we can derive conclusions regarding the applicability of the mapping procedure. Moreover, as part of the applicability analysis, we also strive to identify and propose potential improvements that can be done to further elevate the mapping procedure.
To make conclusions about applicability, we need to investigate the level of detail of transformation on the given examples as well as the Rebeca semantics that were not included in the mapping procedure and the supplying reasons for such decision. Starting by analyzing the given examples, it is obvious that we have a low-level of detail that goes all the way to the most rudimentary behavioral blocks of the Rebeca and provides corresponding semantics in the UML. Some semantics that we specifically introduced in the sequence diagram (as for example inline assignment and declaration of variables) could arguably be denoted as too detailed for the function of the sequence diagram. On the other side, we strongly believe that the sequence diagram should encompass this information within, for multiple reasons that we will mention shortly. Notably, an idealistic solution to this problem would be that UML specification provides adequate modeling elements for these semantics

in the sequence diagram. We also investigated other types of behavioral diagrams (i.e. activity diagram, state-machine diagram) that could potentially be used to model these semantics, although this endeavor ended unsuccessfully. Accurately speaking, we didn't find any behavioral diagrams that provided adequate semantics to be used for such purpose, at least not in the fashion that would grant us the ability to reasonably justify the use of additional types of diagrams. Conclusively, the introduction of additional semantics in the sequence diagram seemed to be the most elegant and adequate solution to this problem. The reasons for such decision are various from increased readability and understandability to avoidance of additional types of diagrams, that coherently increase complexity as well as development and maintenance time. These new semantics did not significantly affect the readability and quality of the sequence diagram while complementing it with important low-level semantics for obtaining meaningful and complete information in the Rebeca.
We can also notice that the provided mapping is able to capture most of the semantics of a Core Rebeca resulting in a complete transformation without a necessity to add any information subsequently. That being said, there are certain semantics available in Rebeca, that were not covered by the mapping procedure. We list them and for each of them we provide reasons for excluding them as well as a possible solution, that is in the scope of the proposed mapping procedure and can be used to fully or partially replace them. As follows:

1. Switch condition
   (a) Justification: Switch conditions are not commonly used. Moreover, many good practices discourage the usage of switches as they can significantly reduce readability. On the other side, if we were to provide mapping for switch condition it would only be possible to capture this information, up to some extent, with the usage of ALT Combined fragment, used for mapping of regular if condition. Having one UML semantic that is used for multiple semantics in the Rebeca can become confusing and messy, as we would need to establish different use cases that would be mapped with each of them.
   (b) Possible solution: We can use regular if condition to fully replace the switch condition.
2. Increment / Decrement (++, –)
   (a) Justification: Increment and decrement cannot be simply modeled within the sequence diagram as it lacks necessary semantics. Moreover, the simplicity of increment and decrement is the main reason for its usage across different languages including Rebeca. However, we don't include this in the mapping to avoid introducing too many low-level semantics.
   (b) Possible solution: We can use inline assignment to fully replace the increment and decrement.
3. Bit-wise operators

(a) Justification: Bit-wise operators are rarely used and most often replaced by logical operators as these are much more efficient and cut the execution time for evaluating the value of conditions (true or false) due to 'short-circuit' logic. These are some of the reasons why they are not included in the mapping.

(b) Possible solution: We can use logical operators to almost fully replace bit-wise operators.

4. Ternary conditions

   (a) Justification: The ternary condition is not used generally and it is highly advised against their use, as they can significantly lower readability of conditional statements and therefore code. This is especially true if ternary conditions are longer then the specified number of characters.

   (b) Possible solution: We can use regular if condition to fully replace the ternary condition.

5. Break and Continue clauses

   (a) Justification: Continue clause is not included due to the lack of adequate modeling semantics to be used for mapping. Hence, UML needs to provide additional semantics for mapping to be viable. On the other side, there is a possibility to specify a break combined fragment that can be used similarly as the break in Rebeca. In this case, the break fragment consists of the specification of condition which, if satisfied, leads to the break of the enclosing statement. However, the introduction of break further reduces the readability of the sequence diagram. Considering that the same end results can be achieved by the usage of regular conditions, the break seems to be unnecessary.

   (b) Possible solution: Modeling and structuring conditional statements in an appropriate way can be used to partially replace the function of continue and break clauses. In this case, however, the loops will not be interrupted (as in the case of break) and will continue to iterate through the specified range, although the end result will be the same.

Along with these semantics, it is important to note that Timed Rebeca is not included as part of the mapping procedure. Although we will shortly provide some information regarding the Timed Rebeca concepts and how they can be modeled with certain UML semantics, available in the sequence diagram. Timed Rebeca introduces temporal logic as an extension of Core Rebeca for modeling and verification of time-critical systems. Timed Rebeca specifies several semantics, used to represent temporal logic. As follows:

1. Delay

   This semantic is used as a standalone concept and it represents a passing of time during the execution of a message server. Basically, it specifies temporal constraint preceding the next object of execution. It can be used for different

use cases and one would be to compensate for irregular or unexpected events that might occur under some circumstances.

2. After

   This semantic is not used as a standalone concept but in conjunction with a message server call. It is used to represent the time that elapses before delivery of message server to its receiver.

3. Deadline

   This semantic is not used as a standalone concept but in conjunction with a message server call. It is used to represent timeout or maximum time for which the message remains valid. Basically, it is a time to serve the message server call after which it is no longer valid.

For the provided temporal concepts in the Rebeca, we attempt to identify closely related UML concepts that could be used for the mapping. With that in mind, we discovered that in the UML, constraints are commonly used for specifying timing requirements upon the messages. They can denote the timing of one message or the duration between multiple messages. We identified the following semantics as part of the sequence diagram with respect to the timing constraints and we provide possible mappings with the aforementioned temporal Rebeca concepts.

1. State invariant

   A state invariant is a run-time constraint or an explicit requirement assigned to the specific element, typically placed on the lifeline with a temporal logic specified in the curly braces. It can be used for specifying a timeout constraint for the specific message calls. Besides, it can also be used for establishing other types of requirements in addition to timing, although that is not of interest for this work.

   (a) Mapping: Often defined in the form $\{t{=}{=}time\}$. Can be used for mapping with the *deadline* specification in the Rebeca ($\{t{=}{=}time\} \xrightarrow{\text{mapped}}$ *deadline(time))*.

2. Duration constraint

   A duration constraint is used between two objects, called start and end object, and represents a constraint on the duration between them. Besides, it can also be placed on a single message call and represents a communication delay constraint on the message. It is commonly used for delaying the execution of the consecutive objects or the single object by specifying a duration requirement between two message calls or on the single message call, respectively.

   (a) Mapping: Defined between two consecutive message calls in the form $\{minValue..maxValue\}$, where minValue and maxValue represent mininum and maximum duration. Can potentially be used for mapping with the *after* specification in the Rebeca ($\{minTime..maxTime\} \xrightarrow{\text{mapped}}$ *after(maxTime))*, or with the *delay* specification ($\{minTime..maxTime\}$ $\xrightarrow{\text{mapped}}$ *delay(maxTime)*.

3. Time Constraint

   A time constraint identifies a constraint that applies to a single object on the sequence diagram. It is not entirely compatible for mapping with the temporal Rebeca concepts. However, it has certain properties which makes it a potential candidate for mapping.

   (a) Mapping: Defined in the form $\{t..t+n\}$.Can potentially be used for mapping with the *deadline* specification ($\{time..time+n\} \xrightarrow{\text{mapped}}$ *deadline(time+n))* or less possibly with the *delay* specification ($\{time..time+n\} \xrightarrow{\text{mapped}}$ *delay(time+n))*.

As we can see, some of the mappings are shared between the concepts. Hence, each timing constraint in the sequence diagram can be used to represent at least one timing concept in the Rebeca. However, we noticed a possible shortcoming with respect to the available semantics for the mapping with *delay* concept in the Rebeca, although, it seems that *after* and *delay* share some similarities, which leaves a possibility of using a duration constraint, in the UML, to represent both depending on the use case. For example, if a target message call in the specified duration constraint is a representation of the message server invocation, we could map it as an *after* clause and, for other scenarios, we could use a *delay* clause. Notably, this is just one possible suggestion and alternatives are not excluded.
Along with Timed Rebeca, it is also important to mention Probabilistic Timed Rebeca. Probabilistic Timed Rebeca is nothing more than a Core and Timed Rebeca from a modeling perspective, with a slight difference in the way non-deterministic expressions are specified. Notably, with the current specification of the mapping procedure, there is no limitation regarding probabilistic semantics used in Rebeca, which essentially means that you can also define non-deterministic expressions with probabilities instead of regular values. Concluding with Timed Rebeca and Probabilistic Timed Rebeca, it becomes clear that we strive to include all aspects of the Rebeca in an attempt to provide a comprehensive mapping procedure that should ultimately enable us to model various Rebeca concepts in the UML. Moreover, the results obtained by transformation, show that the mapping procedure is substantial enough to capture all the semantics of most Rebeca models, leaving few to none manually added concepts. This consequentially portrays its wide applicability scope. This section gives an answer to the last research question with respect to applicability:

1. **RQ3:** What is the applicability of the proposed mapping procedure and its proper substantiation?

   (a) **Answer:** This cannot be simply answered in a few words, as the answer is actually conveyed by complete transformation on the given examples and profound discussion conducted in Section 6.

7 Discussion and Limitations

In this section, we discuss the research questions and designated answers as well as the emerged solutions and their limitations.
To the best of our knowledge, the research conducted in this thesis is the first attempt to comprehensive conceptual mapping between a sub-portion of UML and Rebeca, with the goal of enabling formal verification early in the design process. The identification of Rebeca concepts, which preceded the iterative process, has been achieved by studying literature and available examples. This was then followed by the iterative mapping between the identified Rebeca and UML concepts. The main outcome is a comprehensive mapping procedure presented in Tables 5.2 5.3, that directly answers the first two research questions RQ1 and RQ2 (1.1). Hence, the comprehensive mapping procedure has a dual nature of accomplished results. Not only that it provides adequate mappings between a sub-portion of UML and Rebeca but it accomplishes this while preserving minimality of the included UML diagrams. After the mapping is created, in order to establish compatibility of the source UML models with the mapping procedure and therefore ensure that we are in respective domain, we provide an extensive list of correctness attributes, for each mapping in Table 5.2. This is accomplished as part of the pre-conversion validation, and the list of correctness attributes is presented in Table 6.1. Moreover, the assessment of the applicability of the proposed mapping procedure was the next big step that was completed, showing that the mapping procedure is indeed comprehensive and able to cope with different use cases focusing on Core Rebeca. This provides an answer to the last research question RQ3 (1.1).
Nonetheless, the identification of limitations regarding the researched subject is crucial to establish scenarios that could threaten the validity of the findings. The identified limitations range from low to moderate. Minor limitations, denoted as low, are limitations with respect to not included concepts. These include switch condition, increment/decrement, bit-wise operators, ternary condition and break/-continue clauses. They are designated as low considering they can be fully replaced by concepts that are in the scope of the mapping procedure, therefore limiting their negative impact to the minimum. More serious limitations, denoted as moderate, include possible issues related to the minimalist approach regarding the subset of UML modeling concepts as well as the assessment of the mapping procedure (in post-conversion validation). Minimality-driven approach to reduction of types of UML diagrams and their consisting information, indeed shows a decrease in the time and cost for building and maintaining the modeled system. However, reducing the number of behavioral UML diagrams might cause too many low-level details in a single sequence diagram leading to low readability and therefore opposite result from the intended. To deal with this issue, we suggest reducing the complexity of message servers and local methods, that these sequence diagrams essentially represent, by breaking down the functionality into multiple smaller cohesive units. This, in turn, increases readability, traceability and therefore maintainability of the modeled system. Regarding the assessment of the mapping procedure that is done in the post-conversion validation, we identified two major limitations:
The first limitation is due to the lack of implementation for the proposed mapping

procedure, in a form of automated model transformation tool. In that case, the automated tool would be running the transformation and afterward validation on different applicability scenarios with the purpose to establish the correctness of the target models (runnable in Rebeca and reflecting the source UML models). Although we can estimate the applicability scope, based on the mapping procedure and performed manual transformations, the automation of both the mapping procedure and pre-conversion validation of source UML models could potentially lead to other significant findings through more precise and extensive analyses. Regarding the justification for the lack of implementation, we state that this decision is made mainly on the basis of available resources for conducting the research. We analyzed both alternatives (automated and manual transformation), and came to realization if such implementation of a model transformation tool was done in this thesis, it would negatively affect the quality and extensiveness of the mapping procedure by decreasing. These conclusions are made by listing time constraints that would be consumed by the implementation of a model transformation tool and attempting to integrate them in the existing time plan for the duration of the thesis. The new time plan constrained by approximately half, the available time for the creation of the mapping procedure. In this new time plan, we were lacking approximately 2 months, for the detailed mapping procedure and implementation to be complete. This lead us to prioritize and focus mainly on the quality and extensiveness of the mapping procedure as well as the validation phases, leaving the implementation part for future improvement.

The second limitation is with respect to how the evaluation of the mapping procedure is done in post-conversion validation. While the two subjects are involved as part of the first applicability scenario in doing the manual transformations and certain results are obtained, the work is not organized as a structured empirical study (i.e. experiment). On the other side, certain aspects of an experiment are present (i.e. controlled variables as experience of the subjects and provision of the same pre-validated source models, etc.). However, it lacks proper design and execution phases as well as the quantitative analysis of the obtained results. This is therefore identified as the limitation regarding the evaluation of the mapping procedure.

Notably, with respect to the UML concepts that are Papyrus-centric and not provided in the original specification of the UML by OMG, we did not identify any of such concepts in the provided mapping procedure.

8 Conclusion and Future Work

In this section, we present a conclusion of our work as a summary and potential directions for future research related to this topic.
The work in this thesis explores the syntax of the Rebeca language and attempts to provide mapping, for each identified Rebeca concept, with a corresponding concept in the UML. Considering the semantic richness of the Rebeca, it is not unusual that certain concepts do not have a compatible mapping pair in the UML. These concepts and all other cases, for which we do not provide a mapping, are disregarded as the limitations of the mapping procedure. The proposed mapping procedure, to the best of our knowledge, is the first attempt to comprehensive conceptual mapping between a sub-portion of UML and Rebeca including both structural and behavioral concepts. In the end, we proposed correctness attributes, as part of the pre-conversion validation, with reasons to establish the common ground for modeling of the included UML diagrams. These correctness attributes are accountable for making the source UML models compatible for application of the mapping procedure. The mapping procedure is validated by transformation on the defined UML models by two selected subjects. The results of this transformation show the wide range applicability of the mapping procedure and serve as evidence that asserts its comprehensiveness.
The work that is done as part of this thesis can be extended in the following ways. The main extension of this work would be the implementation of the proposed mapping procedure that inherently enables automated model transformation based on the provided UML models. Additionally, considering the importance of the pre-conversion validation, it would be wise to implement the automated validation of the source UML models based on the proposed list of correctness attributes. Notably, this would significantly optimize the success of model transformation and obtaining correct Rebeca models by prevention towards using incompatible UML models in the transformation. Besides, other extensions could potentially include more comprehensive mapping procedure that covers the concepts that are not included in this thesis and/or optimizes the already included mappings as well as correctness attributes in the pre-conversion validation. Finally, Rebeca is a fast evolving language which opens a door for potential future extensions in the mapping procedure to reflect the latest and greatest Rebeca concepts whether it means updating the existing or introducing the new ones.

References

[1] A. Jafari, E. Khamespahanh, H. Hojjat, Z. Sabahi Kaviani, and Sirjani, "Rebeca user manual," 2016.

[2] A. Nguyen-Duc, "The impact of software complexity on cost and quality - A comparative analysis between open source and proprietary software," *CoRR*, vol. abs/1712.00675, 2017. [Online]. Available: http://arxiv.org/abs/1712.00675

[3] F. Alavizadeh and M. Sirjani, "Using uml to develop verifiable reactive systems." 01 2006, pp. 554–561.

[4] R. B. France, S. Ghosh, T. T. Dinh-Trong, and A. Solberg, "Model-driven development using uml 2.0: promises and pitfalls," *Computer*, vol. 39, pp. 59–66, 2006.

[5] S. Fatemeh Alavizaedh, A. hashemi nekoo, and M. Sirjani, "Reuml: a uml profile for modeling and verification of reactive systems," 09 2007, pp. 50–50.

[6] M. Broy and M. Cengarle, "Uml formal semantics: lessons learned," *Software and System Modeling*, vol. 10, pp. 441–446, 10 2011.

[7] D. Firesmith. (2017) Multicore processing. [Online]. Available: http://insights.sei.cmu.edu/sei_blog/2017/08/multicore-processing.html

[8] C. T. Lopez, S. Marr, H. Mössenböck, and E. G. Boix, "A study of concurrency bugs and advanced development support for actor-based programs," *CoRR*, vol. abs/1706.07372, 2017. [Online]. Available: http://arxiv.org/abs/1706.07372

[9] B. Storti. (2015) The actor model in 10 minutes. [Online]. Available: http://www.brianstorti.com/the-actor-model/

[10] J. E. Smith, M. Kokar, and K. Baclawski, "Formal verification of uml diagrams: A first step towards code generation," 11 2001, pp. 224–240.

[11] W. Mcumber and B. Cheng, "A general framework for formalizing uml with formal languages." 01 2001, pp. 433–442.

[12] I. Ober, "An asm semantics of uml derived from the meta-model and incorporating actions," vol. 2589, 05 2003.

[13] A. Evans, R. France, K. Lano, and B. Rumpe, "The uml as a formal modeling notation," 09 2003.

[14] M. Kersten, J. Matthes, C. Fouda Manga, S. Zipser, and H. Keller, "Customizing uml for the development of distributed reactive systems and code generation to ada 95," vol. 23, 09 2002.

[15] G. Booch, J. Rumbaugh, and I. Jacobson, "Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series)," *J. Database Manag.*, vol. 10, 01 1999.

[16] N. Ceta. (2018) All you need to know about uml diagrams. [Online]. Available: https://tallyfy.com/uml-diagram

[17] (2018) Rebeca modeling language. [Online]. Available: http://rebeca-lang.org/

[18] M. M. P. Kallehbasti, "Scalable formal verification of uml models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 847–850.

[19] K. Lano and J. Bicarregui, "Formalising the uml in structured temporal theories k. lano, j. bicarregui," 04 2002.

[20] S.-K. Kim and D. A. Carrington, "A formal mapping between uml models and object-z specifications," in *Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, ser. ZB '00. London, UK, UK: Springer-Verlag, 2000, pp. 2–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=647284.722952

[21] E. Sekerinski, "Graphical design of reactive systems," in *B*, 1998.

[22] Object management group. [Online]. Available: https://www.omg.org/