Connecting a Design Framework for Service-oriented Systems with UPPAAL model-checker

Predrag Filipovikj

June, 2013
Master’s thesis in Computer Science 30 ECTS
Supervisor: Aida Čaušević
Examiner: Cristina Seceleanu

MÄLARDALEN UNIVERSITY
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING
VÄSTERÅS
SWEDEN
Abstract

In the context of Service-Oriented Systems (SOS), services represent loosely coupled discrete units that can be created, invoked, composed and decomposed upon a client request. In such a setting, where complex systems are composed out of services based on the client request, ensuring the expected level of Quality-of-Service (QoS) becomes a difficult task. In systems built on service-oriented principles, the formal specification of both functional and extra-functional system behavior, service availability, compatibility and interoperability between different services and systems have become important issues. To be compliant with the new features, the REMES language has been extended towards SOS with new constructs that have been given formal semantics. In this thesis, we propose transformation rules, definitions and techniques for transforming these new constructs into Timed Automata (TA) counterparts to facilitate the formal analysis. Also, we present an extension to an existing REMES SOS IDE toolset for performing an automated transformation of the REMES SOS models into the TA framework suitable for the formal analysis with the UPPAAL model-checker. The contribution from our work is on two fronts: a) define transformation rules for all of the constructs specific for the REMES SOS modeling and b) prototype implementation of the transformation rules as an extension add-on to the already existing IDE for modeling SOS to perform the automated transformation. The benefit of performing an automated transformation of the REMES SOS models in TA is twofold. First, by automating the transformation process, the process of validation of the models becomes faster. Second, we considerably reduce the influence from the human factor in the entire process, and at the same time lower the risks of introducing errors into the systems in the phase of creating the formal model. Additional benefit from the automated process is that the SOS designer does not have to be a verification expert in order to be able to verify the modeled system.
Acknowledgements

I would like to express my deepest gratitude to my examiner Cristina Seceleanu and my supervisor Aida Čaušević. This thesis would not have been possible without their great support and guidance. Thank you for your patience, knowledge and believing in me in the most critical situations.

I would also like to thank Professor Sasikumar Punnekkat and other personal involved in the EUROWEB project for giving me the opportunity to be a part of Mälardalen University. I would like to thank Professor Ivica Crnkovic for giving me the best tips and tricks for success. Thank you for the believing in me and giving me the opportunity to play on the big stage.

Last but not least, I would like to thank my parents, Verica and Filip Filipovikj for making me the person that I am today. You have been the greatest support throughout my life and everything that I have accomplished I owe to you.

Västerås, June 2013
Contents

1 INTRODUCTION .................................................................................................................. 3
2 PRELIMINARIES .................................................................................................................. 5
  2.1 SERVICE-ORIENTED SYSTEMS .................................................................................... 5
  2.2 REMES: A RESOURCE MODEL FOR EMBEDDED SYSTEMS ........................................ 6
  2.3 TIMED AUTOMATA ....................................................................................................... 12
  2.4 MODEL-CHECKING ..................................................................................................... 13
  2.5 UPPAAL ......................................................................................................................... 15
3 PROBLEM ANALYSIS AND TRANSFORMATION RULES FOR TRANSFORMING
   REMES SOS MODES INTO TIMED AUTOMATA .............................................................. 16
  3.1 PROBLEM ANALYSIS .................................................................................................... 17
  3.2 TRANSFORMATION OF REMES ATOMIC MODE INTO TIMED AUTOMATA ................... 18
  3.3 TRANSFORMATION OF REMES COMPOSITE MODE INTO TIMED AUTOMATA ............... 21
  3.4 TRANSFORMATION OF REMES PARALLEL COMPOSITION INTO TIMED AUTOMATA .... 26
  3.5 TRANSFORMATION OF REMES AND/OR MODES INTO TIMED AUTOMATA ................. 27
4 IMPLEMENTATION .............................................................................................................. 36
  4.1 EXTENSION FOR THE REMES SOS IDE ................................................................. 36
  4.2 STATIC STRUCTURE OF THE EXTENSION TOOL ......................................................... 37
  4.3 FUNCTIONAL ASPECT OF THE EXTENSION TOOL ....................................................... 38
    4.3.1 Atomic mode translator ......................................................................................... 40
    4.3.2 Atomic submode translator ................................................................................... 41
    4.3.3 Composite mode translator .................................................................................. 43
    4.3.4 Synchronized REMES mode translators .............................................................. 46
  4.4 INTEGRATION WITH THE REMES SOS IDE ............................................................ 49
  4.5 DISCUSSION ON THE APPROACH ............................................................................. 50
5 AN ILLUSTRATIVE EXAMPLE ............................................................................................ 51
  5.1 SERVICE REPOSITORY ................................................................................................. 52
6 RELATED WORK ................................................................................................................ 64
7 CONCLUSIONS AND FUTURE WORK ............................................................................. 67
8 REFERENCES ...................................................................................................................... 69
List of Figures

Figure 2.1 A Composite REMES service ................................................................. 7
Figure 2.2 Examples of TA .................................................................................. 12
Figure 2.3 Schematic view of the model-checking approach [6] ......................... 14
Figure 3.1 REMES IDE user workflow diagram .................................................. 17
Figure 3.2 Initial system automaton (global trigger) ........................................... 19
Figure 3.3 A REMES atomic mode ..................................................................... 19
Figure 3.4 Transforming a REMES atomic mode into TA framework ............... 19
Figure 3.5 Transforming a REMES atomic submode into TA ......................... 20
Figure 3.6 A REMES composite service composed out of two submodes and one conditional connector ................................................................. 22
Figure 3.7 A timed automaton corresponding to a REMES composite mode ... 22
Figure 3.8 A timed automaton generated from transforming a non-lazy sub-mode in the Composite service depicted in Figure 3.6 .................. 23
Figure 3.9 An additional automaton used to trigger the edges between the non-lazy mode and the synchronization location as soon as the guard becomes active 24
Figure 3.10 Parallel REMES composition ........................................................... 26
Figure 3.11 Timed automaton corresponding to an atomic REMES mode inside parallel composition ................................................................. 27
Figure 3.12 AND REMES mode ....................................................................... 28
Figure 3.13 OR REMES mode .......................................................................... 29
Figure 3.14 Transforming REMES atomic modes synchronized with the Init/Entry location of the AND/OR mode ................................................. 30
Figure 3.15 Transforming a REMES atomic mode synchronized at the end of execution of the AND synchronized mode ........................................... 32
Figure 3.16 Transforming a REMES atomic mode synchronized with the end of execution of the OR synchronization mode ....................................... 32
Figure 3.17 Transforming REMES atomic modes inside AND/OR mode not connected to the input or the output pin of the AND/OR mode ................. 35
Figure 4.1 Structure of a UPPAAL document .................................................... 37
Figure 4.2 Hierarchy of translator classes .............................................................. 39
Figure 4.3 Extension tool integration into the main context menu ............................................. 49
Figure 5.1 ATM machine service compositions ........................................................................ 52
Figure 5.2 A service composition for the ATM system modelled in the design framework. .......................................................................................................................... 54
Figure 5.3 Login REMES composite mode ............................................................................... 55
Figure 5.4 Timed automaton generated from the LOGIN REMES composite mode ........ 55
Figure 5.5 MONEY_WITHDRAW REMES composite mode .................................................... 57
Figure 5.6 The timed automaton generated from the MONEY_WITHDRAW REMES composite mode. .......................................................................................................................... 57
Figure 5.7 ACCOUNT_INFO REMES mode ............................................................................ 58
Figure 5.8 ERROR REMES mode .......................................................................................... 59
Figure 5.9 Timed automaton corresponding to ACCOUNT_INFO REMES atomic mode. .......................................................................................................................... 59
Figure 5.10 Timed automaton corresponding to ERROR REMES atomic mode .......... 59
Figure 5.11 AND synchronized mode composed of BALANCE_DISPLAY and PRINT_RECEDE atomic modes. .......................................................................................................................... 60
Figure 5.12 Timed automaton corresponding to the BALANCE_DISPLAY AND synchronized atomic mode. .......................................................................................................................... 61
Figure 5.13 Timed automaton corresponding to the PRINT_RECEDE AND synchronized atomic mode. .......................................................................................................................... 61
Figure 5.14 LOGOUT REMES atomic mode .......................................................................... 62
Figure 5.15 Timed automaton corresponding to the LOGOUT REMES mode .................. 63
1 Introduction

The requirements and features that modern software systems have to deliver to the end users grow exponentially with the number of incorporated applications and scenarios into them. At the same time, one should be able to reduce the development time of new applications through a software reusability and componentization. The Service-Oriented Systems (SOS) paradigm provides a design solution for constructing large scalable systems based on collections of independent discrete software modules called services. The idea behind using small reusable units is to provide assets for building large and complex systems out of small building blocks. The benefits from following the SOS approach are evident through the decreased complexity and improved code reusability within the systems. In many of the cases the practice of the SOS paradigm can lead to lowering the costs while building and maintaining such systems. Additional advantage from following the SOS paradigm is the fact that systems/services become available on users’ demand, meaning that the users can invoke, compose, and decompose services within the system according to their needs.

To address such needs, an extension of REMES, an already existing resource-aware timed behavioral modeling language has been proposed [5]. To support designers to use the REMES behavioral language for modeling SOS – design framework called REMES SOS IDE for behavioral modeling of services has been developed [7]. The current set of features of the framework includes Graphical User Interface (GUI) for modeling SOS in a form of editable diagrams, a textual dynamic compositions service language and an automated process for verifying the correctness of the created model. The framework, however, lacks support for the formal analysis and verification of the created SOS models. The importance of the formal methods comes to focus when building complex software systems, as their aim is to establish correctness of an underlying mathematical precise model. The goal is to enhance the feature set of the REMES SOS IDE by adding a mechanism that will provide automated support for the transformation of the REMES model into a formal model suitable for the formal analysis. To provide such functionality, we define a set of transformation rules for transforming the existing REMES SOS modes into TA model. Henceforward, based on the defined set of rules, an extension tool that automates the process of transformation has been developed. In the process of defining the transformation rules of the SOS models into TA, we have adopted the
transformation rules defined by Orlić et al. [9] for transforming REMES Embedded Systems (ES) models into Priced Timed Automata (PTA) formal framework, and modified them to be compliant with the newly introduced SOS constructs of the REMES language. However, the set of the formal definitions does not fully comply with all of the REMES SOS supported models. To overcome the issues we have extended the definition set by introducing formal definitions for transformation rules to support REMES SOS specific constructs that are not present in the REMES language for modeling ES.

As a part of the thesis work, we propose a prototype implementation of an extension add-on to automate the process of transforming the REMES SOS models into the TA formal model suitable for the formal analysis with the UPPAAL model-checker. The tool is based on the set of defined transformation rules and techniques implemented in Java programming language, integrated as an extension to the existing REMES SOS IDE. This extension add-on provides all the required assets to support automatic transformation of the REMES SOS models into TA. To prove our concept and to validate our research contributions, we will consider a real-world example of an Automated–Teller Machine modeled as a REMES SOS system. By applying our tool to the REMES SOS model, we will generate UPPAAL compliant TA model suitable for model-checking.

The thesis is organized as follows. In Chapter 2 we present the background information about SOS, the REMES behavioral language, and the fundamentals of model-checking together with TA formal framework description and UPPAAL model-checker. In Chapter 3 we will formulate the problem of the thesis and present an overview of all transformation definitions used to transform the REMES SOS modes into TA. Chapter 4 provides details about the implementation of the extension and its integration into the existing REMES SOS IDE. In Chapter 5 we apply our extension tool to a real-world example in order to validate the contributions. Chapters 6 and 7 conclude the thesis by presenting the related work and conclusion remarks together with some ideas about the future improvements of the framework.
2 Preliminaries

In this chapter we provide preliminary explanations of the service behavioral modeling, tools, and concepts on which the REMES behavioral framework is based. We start this chapter by explaining the SOS paradigm in more details as the fundamental basis for this work. Than we present an overview of the REMES modeling language used for behavioral modeling of services and service compositions followed by the description of the TA formal framework. Later on, we present more details on model-checking, and finally we conclude this chapter with a description of the UPPAAL toolset.

2.1 Service-oriented systems

The constant growth of the complexity in the modern software systems can be observed as a consequence of different factors such as the increasing number of functionalities added to support more features, high quality standards imposed by the modern engineering practices, mobility, the increasing degree of distribution, etc. The service-oriented paradigm has emerged as a result of efforts to address the problem of increasing complexity of the systems. It has evolved from the object-oriented and component-based concepts, based on two main principles: (i) modularization and (ii) composition. The concept of modularization focuses on reduction of the system complexity by splitting the high level functionalities into smaller separate modules of behavior, while the process of composition provides means for recomposing the separate behavioral units in different manners to efficiently reproduce the behavior of the system.

The basic functional units of the SOS architecture are services. Number of definitions about services can be found in the literature, and one of the most precise has been given by Broly et al. [2], stating that a service can be defined as “a set of functions provided by a software or system to client software or system, usually accessible through application programming interface”. The aim of introducing the concept of services is to support low-cost but yet secure and reliable software systems. The services are enhanced with interface information that enables them to be published, invoked, coupled and decoupled on demand. They are designed and built as discrete pieces of code that enable code reuse. Due to the reusability properties of the services the need for developing software components each time a new business process arises has been significantly reduced. Each service has its
behavioral description and publicly exposed interface information. The separation of concepts enables service users to consume service functionality exposed by the public interface, without knowing about their implementation. Although the service implementation is hidden from the outside, it is available to the service developers. The publicly exposed interface provides service specific information like time-to-serve, service capacity, etc. The implemented discovery mechanisms enable developers to significantly reduce the cost for developing new systems, by finding and reusing existing services.

Number of service-oriented concepts exists nowadays. The basic notion of the service-oriented approach such as mechanisms for service discovery, predicting the performance, reliability and orchestration and choreography are supported by vast majority of them. Among the most popular approaches for modeling SOS systems are: BPEL4WS [3], BPEL [3], BPMN [15], WS-CDL [19], and others. The common downside for the most of the approaches is the lack of support for formal analysis or the process of getting the analysis model is long and tedious. More details on how formal modeling and analysis features have been introduced into these frameworks compared to our approach will be elaborated into the related work section.

2.2 REMES: A Resource Model for Embedded Systems

A dense-time state based hierarchical modeling language called REMES [1] is used to model the functional and extra-functional behavior of SOS. The set of features that the language supports makes it suitable for describing the resource-wise behavior of the interacting components and services. The REMES language provides support for two main classes of resources, which depending on the type of consumption can be continuous such as energy or discrete such as memory.
The internal behavior of REMES system is described by modes. Depending on their structure, the modes can be categorized as atomic i.e. modes that does not contain any submodes (AtomicMode1 and AtomicMode2 in Figure 2.1), or composite modes that contain arbitrary number of submodes in their internal structure (CompositeService in Figure 2.1). REMES behavioral language supports transfer of both data and control information. To facilitate the data transfer in the system, REMES provides the data interface, while the control in the system is passed via the control interface. The data interface consists of local and global variables that can be of type boolean, natural, integer, array, clock, etc. The clock type is a special type of continuous variable evolving at rate 1, used to express the timing properties of the system. The control interface of a mode (see Figure 2.1) consists of one or more entry and exit points.

The control flow in a composite mode is represented by a set of dedicated edges interconnecting the control points of the modes. The execution of a composite mode is performed in a sequence of discrete steps. There are two types of actions supported by the REMES language – discrete and delay/timed actions. The delay/timed actions
represent the solution to the differential equations that annotate the models as they describe the continuous behavior of the mode. In Figure 2.1 the AtomicService2 atomic REMES mode has the urgent attribute (marked with U) set to true. With this flag set to true, the mode becomes an urgent mode and it is executed and exited without any delay. The discrete actions are present in a form of annotations on the edges and they are performed instantaneously. Execution of discrete actions result in a change of the executing mode, as the outgoing edge starting from the mode exit point has been taken.

Discrete actions can be formalized as tuples of type $A = (g, S)$ where the $S$ is the action body which can be a statement such as assignment, conditional statement, etc., or a sequence of statements that are executed only when the edge has been taken. The action $A$ is enabled and may be executed only when the action guard $g$ is evaluated to true. The time for execution of a particular mode can be controlled with a predicates over a continuous variables called invariants (expression $y \leq b$ in AtomicMode1 in Figure 2.1). Invariants are boolean expressions over system variables used to control the execution/delay time of a particular mode. In addition to atomic modes and edges, a REMES composite mode can also contain conditional connectors (an element decorated with a letter C in Figure 2.1). Conditional connectors allow selection of one of the many possible outgoing edges, based on their respective guards. For a discrete action to be executed, a corresponding guard must be evaluated to true. In case when none of the guards of the discrete actions evaluates to true, no discrete action is executed and the system remains in the current mode. In situations when more than one of the guards evaluates to true, the next discrete action for execution is selected non-deterministically.

REMES language uses special global variables of type clock to model the timed behavior of the system. The clock variables are of continuous nature evolving at rate 1 (variables $x, y$ in Figure 2.1). If decorated with one, the mode can be exited as soon as the corresponding invariant stops to hold.

To provide the constructs that support SOS modeling, Čaušević et al. have introduced the SOS features into the initial REMES behavioral language [5]. As a result of their work, the initial REMES language has been extended with a service behavioral description that provides means for modeling and analysis of services and service compositions.
The basic functional unit for modeling REMES SOS models is service. A service in REMES is represented with a mode (further in the text we will use the terms REMES mode and REMES service interchangeably). Just as in REMES behavioral language for modeling ES, the services in REMES for SOS modeling can be either atomic or composite. To enable the service to be published and invoked, a list of attributes has been exposed at the interface of the REMES service. As depicted in Figure 2.1, every service (atomic or composite) is decorated with the following attributes:

- **capacity** – specifies the maximum number of requests (messages) that can be received, handled and served by the service per time unit;
- **time-to-serve** – indicates the worst-case time in which service will receive a request and will respond to it;
- **service precondition** – a predicate that must evaluate to true in order for service to be evoked and executed;
- **service postcondition** – a predicate (Pre: $\Sigma \rightarrow \text{Bool}$, Pre $\equiv$ (PreInit $\lor$ PreEntry)) that must evaluate to true in order for the service to be invoked. The polymorphic type of state $\Sigma$ includes both the local and global variables, and the initial predicates PreInit and PreEntry;
- **service type** – an attribute that determines the type of the service mode. In the current version of the REMES language the only supported service types are: web service, network service and database service;
- **service status** – indicates the current status of a service. Currently supported values are: idle, active and passive (not invoked).

In order to facilitate the service manipulation, the REMES language for modeling SOS has been enriched with interface operations given by a pre- and postcondition specification. REMES behavioral modeling language supports interfaces for creating and deleting services, creating and deleting service lists as well as adding, removing and replacing services from a service list. The following two examples show interfaces for creating and deleting services:

- **Create service**: create service_name

  
  $\{pre\} : \text{service}_\text{name} = \text{NULL}$

  $\text{create} : \text{Type} \times N \times N \text{“passive”} \times (\Sigma \rightarrow \text{bool}) \times (\Sigma \rightarrow \text{bool}) \rightarrow \text{service}_\text{name}$
- **Delete service**: \(\text{del service}_\text{name}\)
  
  \[
  \begin{align*}
  \text{[pre]} : & \quad \text{service}_\text{name} \neq \text{NULL} \\
  \text{del} : & \quad \text{service}_\text{name} \rightarrow \text{NULL} \\
  \text{[post]} : & \quad \text{service}_\text{name} = \text{NULL}
  \end{align*}
  \]

One possible way for new services to be created is through composition of already existing services which are often represented as independent and distributed functional units. The newly composed services must fulfill the imposed requirements which may be continuously evolving as a result of the dynamic composition of the services. To support the dynamic aspects of services in REMES, a new textual hierarchical language for dynamic service composition (HDCL) has been introduced. This new hierarchical language supports modeling of sequential, parallel and nested services. The Dynamic Composition Language (DCL) and Hierarchical Dynamic Composition Language (HDCL) can be described through the formulas (1) and (2):

\[
\begin{align*}
\text{DCL} := (s_{\text{list}}, \text{PROTOCOL}, \text{REQ}) & \quad (1) \\
\text{HDCL} := ((\text{DCL}, \text{PROTOCOL}, \text{REQ})^+, \text{PROTOCOL}, \text{REQ})^+, ...) & \quad (2)
\end{align*}
\]

According to the formulas (1) and (2), the HDCL facilitates infinite degree of nesting of DCL. The positive closure operator indicates that HDCL is a result of a nesting one or more DCLs. The type of binding between the services with the hierarchy is defined by the PROTOCOL as:

\[
\text{PROTOCOL} ::= \text{unary} \text{operator service}_\text{name} | \text{service}_m \text{binary} \text{operator} \text{service}_n
\]

(3)

The requirement \text{REQ} is a predicate that includes both the functional and extra-functional properties of a composition. Through the requirement predicate, one can define the capability, characteristics, or quality of service of the system. The unary and binary operators present in the PROTOCOL definition are defined as follows:

\[
\begin{align*}
\text{unary} \text{operator} ::= \text{exec} - \text{first} & \quad (4) \\
\text{binary} \text{operator} ::= ; | || | || \text{SYNC} - \text{and} | || \text{SYNC} - \text{or} & \quad (5)
\end{align*}
\]
If we consider a situation where two services $s_1$ and $s_2$ are placed into a list denoted with $s_list$ and we also assume that their execution starts at some given point of time, than the semantics of the unary and binary protocol operators and the correctness of the composition in situation where $s_i \ Pre_i$ represents the strongest postcondition of $s_i$, $i \in \{1, 2\}$ with respect to $Pre_i$ is defined as following:

- **Exec-first** – for a given composition, specifies which service should be executed first. The formal model below shows an example where service $s_1$ should execute first and that service $s_2$ can become active only when execution of $s_1$ is complete and its postcondition has been established.

$$status_{s1} = \text{active} \land status_{s2} = \text{idle} \ Post_{s1} \rightarrow (status_{s2} = \text{active}) \quad (6)$$

If we have a list of services, the executing of service $s_1$ fist can be defined as:

$$\text{Exec – first } s_1 \ \Delta \ s_1 \ \lnot \ \text{gs}_1 \ (s_2 \ Binary\_operator \ ... \ Binary\_operator \ s_n) \quad (7)$$

- **Sequential composition** – represent composition of two services that are executed uninterrupted in a sequence. The correctness condition of the sequential composition is:

$$(s_p.s_2.(s_p.s_1.\ Pre_{s_1}) \rightarrow \ Post_{s_2}) \land (\ Post_{s_2} \rightarrow \text{REQ}) \quad (8)$$

- The correctness of a **Parallel composition** ($s_1 || s_2$) is defined as:

$$(s_p.s_1.\ Pre_{s_1} \lor s_p.s_2.\ Pre_{s_2}) \rightarrow \text{REQ} \quad (9)$$

*Parallel composition with synchronization:* services denoted by S-AND belong to an AND mode that needs to synchronize their execution at the end of the execution.

Considering that, “and” synchronization of those services is defined as follows:

$$\forall s_1, s_2 \in S - \text{AND} \rightarrow (\forall \ n \ now \ \cdot \ status_{s_1} = status_{s_2} = \text{active}) \land (\ start_{s_1} + \ TimeToServe_{s_1} = \ start_{s_2} + \ TimeToServe_{s_2}))$$

The condition for checking the correctness of “and-AND” synchronization is formalized as:

$$(s_p.(s_1 || \ SYNC - \ and \ s_2). \ Pre_{AND} \rightarrow (\ Post_{s_1} \land Post_{s_2})) \land (\ Post_{s_1} \land Post_{s_2} \rightarrow \text{REQ}) \quad (11)$$
2.3 Timed automata

Timed Automaton [10] represents a finite state machine extended with a finite number of real-valued clocks, introduced to measure the time between occurring events. The basic functional units of timed automaton are locations interconnected between each other with edges. As depicted in Figure 2.2 an automaton can be composed of one or more locations (L0 and L1 in Figure 2.2). Considering the automaton presented in Figure 2.2 b), one can notice that it is composed out of two control locations (L0, L1), two clock variables (x, z), and a synchronization action (synch!). One of the locations (L0) has been marked as starting point and it is regarded as the initial location. The other location in the automaton (location L1 in Figure 2.2 b)) has been decorated with a boolean expression, which represents the invariant of the location. It defines the maximum allowed time that the automaton is allowed to spend in that particular location. For illustration the automaton given in Figure 2.2 b) is allowed to stay in location L1 as long as the invariant \((x<=7 \&\& z<=2)\) holds.

Transition between locations can occur through the control edges. Considering the Figure 2.2 it is evident that edges have been decorated with boolean conditions called guards (e.g. \(x > 2\) in Figure 2.2 a)), which must evaluate to true in order for an edge to be taken. In addition to guards, edges can be enhanced with actions and assignments. In most of the cases the assignments are used to reset the clock variables or to update the other data variables in the system (\(x := 0\) in Figure 2.2 a)). For example, if we refer to the edge between the locations L1 and L0 in automaton
presented in Figure 2.2 b) we can notice that during the transition both of the clock variables have been reset.

The state of the system is represented though the current location and the values of the clocks while the system is in that location. In TA system, occurring transitions can be either discrete corresponding to a change of the location of the executing automaton or delayed transitions which model the passage of time.

Also, the timed automaton can be executed in parallel for a given synchronization function representing connected network of TA. One such TA network is given in Figure 2.2 where the separate automata synchronize between each other by passing and receiving synchronization actions, given by synch! and synch?, respectively.

### 2.4 Model-checking

Model-checking also known as model exploration is one of the verification techniques used to establish correctness of a system in early design phase. The other verification techniques on this level are simulation and testing which perform exhaustive experiments over a restrictive set of model or real system scenarios, respectively.

Model-checking is an automated verification technique used to explore all the possible states in the system, based on the brute-force algorithm. The basis of the model-checking techniques is the abstraction of the system under consideration called the model. Baier et al. [6] refer to the importance of the model as following: “Any verification using model-based techniques is only as good as the model of the system”. By traversing all the system states in a systematic way and applying the model-checking technique one can verify if the model truly satisfies certain property. The brute-force manner of checking the models is suitable for handling small state set spaces (up to $10^9$ states), while for larger state space sets (with $10^{20}$ states or more) the model-checking technique has to be based on clever algorithms and tailored data structures [6]. In the process of traversing the state space, the model-checking technique is capable of discovering errors which is very hard to detect with the process of emulation, testing or simulation.
Model-checking technique is suitable for checking both the qualitative and quantitative properties of the underlying system. Qualitative approach aims at validating some of the properties of the system such as correctness, reachability, fairness, safety, and etc. Number of situations can be explored by checking the timing properties of the underlying system. Some of the situations include checking whether a deadlock is possible in the system, and if so, will it occur after some time while the system is running? If the response has been received, has it been received within the predefined time frame? Also, due to the fact that the system properties must be precisely defined, the model-checking technique can find potential flaws into the documentation like inconsistencies or ambiguities in the defined properties.

The behavior of the system or what the system should do is described with the property specification, while the model description addresses on how the system behaves. In the process of system checking, the model-checker examines all the possible system states and checks whether the desired property has been satisfied. If a state that violates some property has been reached, the model-checker is capable of providing a counter example, showing how the system can be lead into that state. The automated checking process allows constant recheck of the model. This can be very useful if some of the features of the model under consideration have been changed after it has been previously verified. Once the model has been corrected, it can be validated again by reapplying the process of model-checking to uncover potentially introduced errors.

Figure 2.3 Schematic view of the model-checking approach [6].
2.5 UPPAAL

UPPAAL is a set of tools based on constraint-solving and on-the-fly techniques for modeling, simulation and verification of real-time systems [8]. It is used for systems that can be modeled as a collection of non-deterministic processes, have a finite structure and real-value clocks and communicate through channels and/or variables. The UPPAAL model-checker is the most suitable for application areas where the timing aspects are critical. By exploring the state space of the system, the UPPAAL model-checker checks the invariant and reachability properties.

UPPAAL toolset consists of three main parts: description language, simulator and a model-checker. The description language serves as a modeling language for describing the system as a network of TA extended with data variables. The language is non-deterministic guarded command language that supports set of data types. Some of the currently supported data types in the UPPAAL language are integer, clock, channel, boolean, etc. The simulator and the model-checker provide interactive automated analysis of the system that has been modeled. Due to the exhaustive dynamic behavior coverage of the system for detecting errors during the model-checking phase, the entire process of verification the model can be very expensive. To overcome this problem, the UPPAAL simulator supports dynamic execution of the system during the design phase. This feature of discovering errors prior the model-checking process minimizes the risk of design errors during the model-checking phase.

UPPAAL supports both textual and graphical formats for the description language. The textual format of the UPPAAL provides basics for a high-level programming language for the TA. In addition to the textual format the graphical interface provides features for system definition in a form of network of TA. In order to provide the WYSIWYV\(^1\) feature of the toolset, the compiler performs transformation of the system description in graphical format into textual format.

The UPPAAL model-checker supports verification and validation of temporal properties, such as safety and liveness, specified in a subset of TCTL (timed

\(^1\) WYSIWYV – What You See Is What You Verify.
computational tree logic) [32] [33]. The UPPAAL toolset simulator visualizes counter examples produced in the process of model-checking and visually points to model errors before the tool performs full formal verification. The UPPAAL TA extends the native TA with notions of bounded integer variables, binary and broadcast channels, and urgent and committed locations.

So, in the next chapter we will elaborate the problem definition of the thesis and we will present the theoretical background in the form of transformation rules for transforming the REMES SOS constructs which are contributions of the thesis. Through the set of transformation rules we provide the necessary means for transforming the REMES SOS models into TA formal framework.

3 Problem analysis and transformation rules for transforming REMES SOS modes into Timed Automata

As described in the introduction, the goal of the thesis is to provide support for automated transformation of the REMES SOS models into TA framework suitable for the formal analysis. To accomplish this, we need to provide a theoretical support for transforming all the REMES SOS designing constructs that have been added to the original REMES language. In order to deploy proposed theoretical solution and enable designers to use it, we need to provide prototype implementation in a form of an extension add-on to the REMES IDE for modeling SOS.

In this chapter we will give more detailed overview of the problem that we aim to solve. For that purpose, we will establish the theoretical basis in a form of transformation rules for transforming the REMES modes for SOS modeling into TA suitable for formal analysis. As the REMES SOS language is based on REMES language for behavioral modeling of ES, the newly defined transformation rules are in compliance with the transformation rules established for the REMES ES models. This set of definitions presented in [9] contains transformation rules for transforming atomic, composite and parallel composition REMES modes into PTA. In this thesis, we propose a modification to the transformation rules to comply with TA, and we also formally define transformation rules for AND and OR synchronized modes typical just for the REMES SOS behavioral language. As a result from our work, we provide all the necessary means to support the formal verification of the newly introduced constructs of the REMES language for modeling SOS.
The chapter is divided into five subsections. In the first subsection we will present the problem definition of the thesis. Second, we present the transformation process for the atomic REMES modes. Third, we define how to transform the REMES SOS composite modes. Later on, in section four we provide the necessary means to facilitate the transformation of parallel compositions of REMES modes into network of TA, and we close the chapter by introducing formal definitions for transforming AND/OR synchronized modes into TA framework.

### 3.1 Problem analysis

![Figure 3.1 REMES IDE user workflow diagram](image)

The problem definition can be presented through a simple scenario performed by the system designer in the process of designing the system. Figure 3.1 depicts a scenario with a typical workflow for designing a SOS model using the REMES tool. The complete workflow has been presented as a set of activities performed by the system designer. If we focus on the workflow, it is evident that the complete set of actions has been divided into two branches of execution. The main branch that spans from top to bottom is composed of actions that are mostly design-oriented. All of these actions are currently supported by the IDE, meaning that they can be automatically performed. On the other side, there is a branch of execution connected with a dashed
line to the main branch. These actions must be performed in order to create formal model of the underlying REMES SOS model. They are marked with gray color to indicate that they are not supported by the IDE, meaning that currently they must be performed by a human verification expert. The problem is however, that in most of the cases the system designer is not a verification expert, so additional actor is required to manually perform these actions in order to formally verify the model by using model-checking technique. Also manual transformation requires substantial amount of time, especially for large and complex systems. Note that the model-checking action is specially marked as it is performed by the UPPAAL tool and is out of scope of our problem.

In order to bridge the gap between the system modeling and the formal analysis of the system, this thesis focuses on defining a set of transformation rules, definitions and techniques that can be implemented as an additional feature to the IDE. To enrich the feature set of the REMES IDE without modifying the core functionalities by any means, the transformation rules has been implemented as an extension addon. By following this approach, the existing core functionalities of the tool are retained, while new features in a form of an individual module have been added to the host environment. There are number of reasons that motivate the adoption of the UPPAAL toolset as a platform for performing model-checking on REMES models. UPPAAL is an academic tool that can be used freely and without restrictions in the academic research projects, and second it is suitable for modeling the REMES behavioral models as finite state machines with a support for the timing aspect. To build a formal model suitable for model-checking with the UPPAAL tool, the REMES SOS model has to be transformed into TA as it is the input language to the model-checking tool.

3.2 Transformation of REMES atomic mode into Timed Automata

An atomic mode, also regarded as service (see Figure 3.3) in REMES language is defined as a standalone REMES mode that does not contain any submodes in its internal structure. In contrast to this, if an atomic mode exists inside a composite mode, e.g. the AtomicSubMode1 inside the CompositeService depicted in Figure 3.6, then it is called atomic submode.
The timed automaton obtained as a result from transforming a REMES atomic mode given in Figure 3.3 is presented in Figure 3.4. The automaton is composed out of two locations and two edges. The location named as Start is the initial location, whereas
the other location corresponds to the atomic mode that is being transformed. The edge between the Start and the AtomicService location is used for synchronizing the automaton execution with the system start-up activated on broadcast synchronization action marked as $\text{start? Start} \rightarrow \text{AtomicService}$ in Figure 3.4. The purpose of the trigger is to activate the execution of the TA corresponding to the modes in the system, and it is broadcasted from the global trigger automaton presented in Figure 3.2. The second transition represents a looping transition decorated with an activation synchronization channel $a1$ ($\text{AtomicService} \xrightarrow{a1!} \text{AtomicService}$) broadcasted by the current mode to reactivate itself and synchronize with the other modes in the system.

The mode properties such as the invariant and the urgent flag are translated into properties of the corresponding location. The other properties of the REMES mode (constants, variables and resources) are in fact the declarations of the variables used by the particular mode. They are translated into corresponding declarations of the timed automaton represented by a Template object in UPPAAL.

![AtomicMode](image)

**Figure 3.5 Transforming a REMES atomic submode into TA.**

As shown in Figure 3.5 the transformation of a REMES atomic submode results in a single location of a timed automaton corresponding to a REMES composite mode. Similar to the REMES atomic mode, the mode properties of the REMES atomic submode are transferred to the corresponding location. The location becomes decorated with the same invariant as the REMES atomic mode, and it can also be marked as Urgent or Committed depending on the flags of the submode.

The variables and constants declared in atomic modes are mapped into local variables and constants of the resulting timed automaton. In contrast to this, the resource variables are translated into global resource variables. The currently imposed constraint is that there will be only one resource variable in the system and that it will be of type integer.
The formal definitions for transforming the REMES ES atomic modes and submodes have been presented by Orlić et al. [9]. In our work we have adopted the transformations and give them the necessary means to support SOS distinctive features of the modified REMES language.

### 3.3 Transformation of REMES composite mode into Timed Automata

A composite mode represents a REMES mode which internal structure is composed out of an arbitrary number of atomic REMES modes and conditional connectors. Compared to the transformation of atomic services, the process of transforming composite services into timed automaton is performed in phases, as the process requires translation of the composite mode connection points into locations and elimination of the conditional connectors from its internal structure. The transformation of composite REMES services into timed automaton is performed as following:

- translate the composite service connection points;
- translate the composite service submodes;
- remove the conditional connectors from the composite service inner structure.
Figure 3.6 A REMES composite service composed out of two submodes and one conditional connector

Figure 3.7 A timed automaton corresponding to a REMES composite mode
Figure 3.7 depicts the timed automaton obtained as a result from transforming the REMES composite mode given in Figure 3.6. In the process of transforming the composite mode connection points into locations, following rules apply: (i) a connection point is mapped into a location only if it has input and/or output edges and (ii) the output connection point locations (Exit and Write) must always be marked as urgent. The second rule for transforming the composite mode connection points is introduced to satisfy the “run-to-completion” semantics of the REMES behavioral language and prevent the timed automaton to stay in those locations.

The automaton stays in Start location waiting for the system start-up triggered by the \textit{start} broadcast synchronization action (Start $\rightarrow$ InitPoint). If the Init connection point of the composite mode has been mapped into a location in the resulting timed automaton, then a transition is constructed to connect this location with the Start location. In the opposite case i.e. when the Init connection point does not map into a location, the initialization transition is created between the Start and the location corresponding to the Entry point of the composite mode. The edge between the InitPoint location and AtomicSubMode2 location in Figure 3.7 corresponds to the edge from the Init connection point of the composite service and the AtomicSubMode2 submode. The same applies to the edge between the EntryPoint and the AtomicSubMode1 location. If we focus on the edge between the WritePoint location and the AtomicSubMode1 location, we can notice that this edge has been decorated with exactly the same properties as the edge from EntryPoint to the AtomicSubMode1. Note in Figure 3.6 that this edge does not exist in the REMES SOS behavioral model. It has been additionally introduced into the automaton to ensure the internal execution of the composite mode as long as the write edge guard holds. By the time the guard on the write edge stops to hold, the mode is exited in the next execution cycle through the Exit connection point. The return edge from the ExitPoint to the EntryPoint location is introduced in the automaton with the purpose to reset it and enable its reactivation after the execution has finished.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.7.png}
\caption{A timed automaton generated from transforming a non-lazy sub-mode in the Composite service depicted in Figure 3.6.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.8.png}
\caption{A timed automaton generated from transforming a non-lazy sub-mode in the Composite service depicted in Figure 3.6.}
\end{figure}
The submodes within the REMES composite service are translated according to the transformations rules defined earlier in Chapter 3. The timing properties of the REMES language support specifying the execution time of a mode. To denote the execution time of a REMES mode, a predicate expression over continuous variables called invariant is used. The composite mode presented in Figure 3.6 contains a submode (AtomicSubMode1) that has not been decorated with an invariant expression. Also, the urgent flag of the mode has been set to false. Atomic submodes that are not decorated with an invariant expression and have the urgent flag set to false are classified as *non-lazy modes*. While the system is in one of these modes, time is allowed to pass until at least one of the guards of the output edges evaluates to true. Because of that, the transformation of the non-lazy modes differs from the transformation rules for the atomic submodes defined in Chapter 3. The automaton presented in Figure 3.8 is corresponding to a non-lazy mode, and it is composed of a location representing the submode and an additional committed location for each of the mode’s output edges.

![Image of an automaton](image)

*Figure 3.9 An additional automaton used to trigger the edges between the non-lazy mode and the synchronization location as soon as the guard becomes active.*

In Figure 3.8 we can see how the non-lazy mode named AtomicSubMode1 placed inside the composite mode in Figure 3.6 has been translated into timed automaton. In the same figure we can notice that the edge between the submode and the Exit connection point of the composite service is split into two parts with an additional committed location between. The first part of the newly created edge is decorated with the same synchronization action and guard as the original edge. The second part of the edge is decorated with the same statement present on the original edge. Also, an additional timed automaton (see Figure 3.9) with one location has been introduced.
into the system in order to provide synchronization actions. The depicted transformation of non-lazy REMES SOS mode has been done in compliance with the formal definition specified in [9].

Observe that the synchronization actions in the timed automaton presented in Figure 3.7. The AtomicService3? synchronization action on the EntryPoint output edges comes from the system’s architecture and it ensures the synchronization of the composite service execution upon entrance with the other modes in the system. The other synchronization actions are used to synchronize the composite service (timed automaton) with other services (timed automata) when it finishes execution. These synchronizations can be observed at the output edges of the WritePoint and they are used to trigger the execution of the modes connected to this connection point of the composite mode. When the composite mode does not synchronize its execution with the other modes in the system, i.e. it is not connected to any other mode through its WritePoint, the output edges originating from WritePoint location are not decorated with synchronization actions, like the automaton presented in Figure 3.8.

Notice that the timed automaton presented in Figure 3.7 does not contain conditional connectors. According to the formal transformation definition for the REMES composite modes specified in [9], the conditional connectors have been replaced with a set of edges defined as $E_{cc} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} e_{ij}$, where $i$ and $j$ denote the number of input and output edges of the conditional connector. Each of the new edges has been constructed as combination of the guards and the statements from the conditional connector’s input and output edges. The edge between the InitPoint and the AtomicSubMode1 locations is produced as a combination of the input edge of the conditional connector and the output edge going towards the submode. The guard at the edge is constructed as a combination of the guards of the combined input and output edges of the conditional connector ($y < 3 \&\& z > 2$ in Figure 3.7). In this fashion we eliminate the conditional connectors, while retaining the functionality of the resulting TA model.

Note that the definition for replacing the conditional connectors within a REMES composite mode does not provide the full compatibility. Numerous examples can be presented where the transformation rules for conditional connectors do not result in a correct TA. To avoid erroneous results, we suggest that the resulting TA should be checked by a human verification expert prior the model-checking process in order to validate its correctness.
3.4 Transformation of REMES parallel composition into Timed Automata

A parallel composition is represented by a set of REMES modes, be they composite or atomic placed between parallel connectors (AtomicService3 and AtomicService4 in Figure 3.10). The parallel composition was introduced in REMES language with a function to synchronize the execution of the REMES modes [5]. The resulting transformation from REMES parallel composition is a network of TA, where each individual timed automaton corresponds to a REMES mode within the composition. To demonstrate how the REMES parallel connection is translated into a network of TA, we present the transformation of one REMES mode from the parallel composition. The timed automaton resulting from the transformation of the AtomicService3 (see Figure 3.10) placed in the parallel connection is presented in Figure 3.11.
If we observe the Figure 3.11 one can notice that the presented timed automaton corresponding to the AtomicService3 mode matches to great extend with a timed automaton generated for REMES atomic mode placed outside a parallel composition. The only difference between the two automata is on the initialization edges where the automaton has been synchronized with AtomicService1 and AtomicService2 modes, placed outside the parallel composition. This affects only the REMES modes that are directly synchronized with the connectors of the parallel composition. The transformation of the modes that are not directly connected with the output edge of the parallel connector or placed between other REMES modes is not affected by the parallel connectors, as they are translated according to already defined rules for REMES modes.

The difference in transformation of the composite REMES modes directly connected to the parallel connector is on the initialization edge. Similar to the atomic REMES modes, the transformation created initialization edge for each of the REMES modes to which the composite mode has been synchronized through the parallel connector.

### 3.5 Transformation of REMES AND/OR modes into Timed Automata

To synchronize the behavior of services, REMES language for modeling SOS introduces a special type of modes called AND and OR synchronized mode [5]. According to the semantic definition, from the view point of an external observer the modes synchronized within the AND/OR mode finish their execution simultaneously. The key difference between the two types of synchronized modes is at the start of the execution of the modes within. In a case of AND synchronized
mode, by default it is assumed that all of the synchronized submode lanes will start their execution at the same point of time. This is due to the fact that the AND synchronized mode does not allow guards at the incoming edges. By submode lane we assume a chain of connected REMES services starting from the input connection point of the synchronized mode and ending up in the output connection point of the same. In contrast to this, the OR synchronized mode allows guards on its incoming edges; therefore the start of the execution of the submode lanes is directly dependent upon the guards. Due to this dependency, the simultaneous execution of the OR synchronized compositions cannot be guaranteed. The AND and OR synchronized modes modeled in REMES SOS IDE are shown in Figure 3.12 and Figure 3.13, respectively.

Similar to the transformation of the parallel compositions, the result from transforming the AND/OR synchronized modes is a network of TA, where each automaton is corresponding to a particular REMES mode existing within the AND/OR mode. As defined by Čaušević et al. [5] AND and OR modes have similar behavior. Direct implication from this great similarity between the synchronized modes is reflected on the transformation rules, which are also similar to a great extent. The differences occur due to the different input and output synchronization of the modes.

![AND Mode Run](image)

*Figure 3.12 AND REMES mode*
The resulting TA network corresponding to a synchronized mode is produced by transforming all the individual atomic REMES modes placed within the AND/OR synchronized mode. Depending on the synchronization upon the start and the end of the execution, the set of atomic REMES modes within AND/OR mode can be divided into three groups. The first group represents the modes that need to be synchronized at the end of the execution of the AND/OR mode as they are directly synchronized with the output connection point e.g., modes named Service 5 and Service 6 in Figure 3.12 and Figure 3.13, respectively. The second group represents the modes that are synchronized with the input connection point of the synchronized mode; Service 1 and Service 2 in Figure 3.12 and Figure 3.13 belong to this group of modes. The third group of modes (also referred as the middle layer) are the ones that are not directly synchronized with the AND/OR mode connection points. Typically in synchronized REMES modes, these modes are placed between the modes synchronized with input and the modes synchronized with the output connection points of the AND/OR mode. Service 3 and Service 4 in Figure 3.12 and Figure 3.13 belong to this group of modes.
The above introduced classification of the modes inside synchronized modes is crucial for the implementation, as different transformation rules apply for the different groups of modes depending on their synchronization. Figure 3.14 depicts the resulting TA produced from transforming the REMES atomic modes synchronized with the Entry connection point of the AND/OR mode. From the presented TA we can see that these atomic REMES modes have been transformed in compliance with the rules for transforming atomic REMES modes introduced earlier in Chapter 3. In this particular case, the synchronization channel present on the initialization edge (atomicservice1_synch in Figure 3.14 a) and b)) has been used to synchronize the modes’ execution with another mode placed outside the synchronized REMES mode. The connection between these two modes is the input connection point of the synchronized mode, through which the first mode within the REMES synchronized mode is aware of the modes outside the composition. At the same edge, there is a difference in a statement set between the modes placed inside AND and OR synchronized modes. In Figure 3.14 a) corresponding to the mode placed in an AND synchronization mode one can notice that the statement set is empty i.e., the initialization edge has not been decorated with any statements. In contrast to this, it is evident that the same edge of the timed automaton presented in Figure 3.14 b) is corresponding to the atomic REMES mode within OR synchronized mode has been decorated with a statement \((O R M o d e 3 _ { S Y N C \_ V A R \_ V A L U E } = O R M o d e 3 _ { S Y N C \_ V A R \_ V A L U E } + 1)\) that increments the counter variable by one. The role of the counter variable is to keep track of the submode lanes that have
started with execution. This feature is an essential part of the mechanism used for avoiding deadlock situations when translating OR composition modes into TA network.

The section continues as following. First we present a formal definition for transforming REMES atomic modes directly connected to the Entry connection point of the AND/OR mode. Secondly, we provide formal definition for transforming atomic REMES modes connected to the Exit connection point. We conclude the chapter with explanation about transforming the middle layer REMES modes inside AND/OR mode by using existing transformation rules presented earlier in Chapter 3.

If we assume that $N_M$ is the set of all modes that are part of the AND/OR mode and are directly synchronized with the Entry point of the AND/OR mode, we can then formally define the transformation into a network of TA in **Definition 1**.

**Definition 1.** Let $N_M$ be the set of atomic modes of the AND/OR mode that are connected to the Entry point of the AND/OR mode denoted with $M_{AO}$. The transformation $r2t \equiv N_M \rightarrow N_T$ transforms these modes into a network of timed automata. The transformation $r2t$ of a set $N_M$ is performed by transforming the individual modes $M \in N_M$ into corresponding automata $a_m \in A_M$, respectively, where $|N_T| = |A_M|$. Additionally, we define a “counter” variable $M_{AO\_SYNC\_VAR} \in V_o$, where $V_o$ is a set of data variables of the AND/OR mode. The variable keeps track of the number of the invoked submode lanes within an AND/OR mode: when a submode lane within the AND/OR mode is invoked, the counter is incremented.

The transformation of the modes synchronized at the exit of the synchronization REMES mode (see Service 5 and Service 6 in Figure 3.12 and Figure 3.13) has to provide synchronization with the corresponding network of TA for atomic modes synchronized at the end of the execution. Figure 3.15 and Figure 3.16 present timed automaton obtained as a result of transforming REMES atomic modes synchronized upon the end of the execution of AND and OR modes, respectively.
Figure 3.15 Transforming a REMES atomic mode synchronized at the end of execution of the AND synchronized mode.

Figure 3.16 Transforming a REMES atomic mode synchronized with the end of execution of the OR synchronization mode.
The mechanism added to the timed automaton in order to provide a synchronization of modes at the end of the execution of an AND/OR mode consists of an integer variable (ORMode3_SYNC_VAR in Figure 3.16) and an additional committed location (location decorated with C in Figure 3.16). The integer variable has been introduced into the system to count the number of the exit atomic modes that have finished execution. This is very important, as only the last mode that executes within the synchronized mode is broadcasting a signal on synchronization channel in order to invoke the execution of the next modes in the system.

According to the formal definition [5], AND/OR synchronized modes are allowed to operate in one of two possible regimes: and or max. The operating regime captures the behavior of the AND/OR modes upon finishing the execution. If a particular AND/OR mode is operating in and regime, it is expected that all exit atomic modes finish execution simultaneously, while if the synchronized mode operates in max regime the synchronized mode finishes its execution when the last mode in the synchronized composition completes its execution. Our proposed synchronization mechanism is tailored to satisfy both of the synchronization regimes.

To ensure the synchronization at the end of execution, the synchronization edge connecting the location corresponding to the mode (OrModeName_Service6 in Figure 3.16) and the initial location (location marked as Start in Figure 3.16) of automaton has been split in two parts with a committed location (location decorated with C Figure 3.16). On the transition between the location corresponding to the atomic mode and the committed synchronization location the counter variable (OrMode3_SYNC_VAR Figure 3.16) has been updated, indicating that the current mode has finished its execution. There are two possible execution paths between the committed location and the initial location of the automaton i.e. there are two reset edges for the automaton. On both of the edges it is tested whether the mode finishes execution before, or it finished last or simultaneously with the other modes. If we go back to the Figure 3.12 and Figure 3.13 again we can notice a difference in the statement for comparing the value of the variable between the AND and OR synchronization modes. In the AND mode the counter variable is compared with a number equivalent to the number of executing compositions, while in the OR mode the same variable is compared to a zero value constant, after decrementing it first. This is the variable that was introduced in the transformation of the modes connected to the input connection point of the synchronization mode. The purpose of this
variable is to count the number of modes that have started the execution before the synchronized mode finishes its execution. This feature is crucial for the OR synchronized mode since it allows guards at the output edges from the input connection point. Without this variable, there is a possibility that some of the guards on the input edges will not evaluate to true thus resulting in not invoking some of the input modes, so the submode lanes following it would never be executed. This situation can consequently lead to deadlock at the exit of the AND/OR mode.

The AND synchronized modes are not prone to a deadlock problem, due to the fact that all of the input modes are executed simultaneously and all of the executing submode lanes are expected to finish in some point of time, avoiding a possible deadlock situation at the end of the execution.

If we assume that Nm is the set of all modes that are part of the AND/OR mode, and that are connected directly to the Exit point of the AND/OR synchronized mode, we can then formally define the transformation into a network of TA in Definition 2.

**Definition 2.** Let $N_M$ be the set of atomic modes of the AND/OR mode that are connected to the Exit point of the AND/OR mode denoted with $M_{AO}$. Assuming that $M \in N_M$, then the transformation $r2t \triangleq N_M \rightarrow N_T$ transforms these modes into a network of timed automata. The transformation $r2t$ of a set $N_M$ is performed by transforming the individual modes $M \in N_m$ into corresponding automata $a_m \in A_M$, respectively, where $|N_T| = |A_M|$. Additionally, we define a “counter” variable $M_{AO\_SYNC\_VAR} \in V_t$, where $V_t$ is a set of data variables of the AND/OR mode. The variable keeps track of the number of the invoked submode lanes within an AND/OR mode, (that is, the counter is decremented at each invocation), and when it reaches 0, the AND/OR mode finishes its execution by broadcasting a synchronization action $M_{AO\_SYNC}$.

The third group of REMES modes within AND/OR synchronized mode are the ones that are not synchronized with the connection points of the AND/OR synchronized mode. These modes are placed between the modes synchronized with Entry connection point and the modes synchronized with Exit connection point of the AND/OR synchronized mode (see Service 3 and Service 4 in Figure 3.12 and Figure 3.13). For these modes, the transformation rules for transforming atomic REMES mode apply without any changes. The timed automaton obtained from transforming the modes is presented in Figure 3.17.
In this chapter we have introduced all the theoretical rules required to perform transformation from REMES SOS modes into TA framework suitable for the formal analysis with UPPAAL tool. In the next chapter we will give a complete overview of the defined transformation rules and their implementation as an extension add-on to the REMES IDE to perform an automated generation the TA formal model. Step-by-step we will go through each phase of creating the tool, from defining the static structure until reaching the complete functionality. We will wrap up the presentation of the extension tool by presenting how the formal verification module has been added to the IDE and how designer can use its functionality.
4 Implementation

In this chapter we present details about the implementation of the REMES SOS IDE extension that provides features for transforming the REMES SOS models into TA suitable for analysis with the UPPAAL model-checking tool. As a core functionality of the extension tool, we explain the implementation of the transformation rules presented Chapter 3 in Java programming language.

The chapter is divided in two parts. In the first part, we present the detailed description of the internal structure, both the static and the functional aspect of the extension implementing the transformation rules. In the second part of the chapter we describe how the extension tool has been integrated into the host REMES SOS IDE environment to support transformation of the REMES SOS modes into TA.

4.1 Extension for the REMES SOS IDE

The basic requirement for the extension tool is to provide some type of a communication channel between the already existing REMES service-oriented design framework and the UPPAAL model-checker. To support that, the extension must provide means for transforming the REMES SOS modes into TA framework and store them in a format readable for the UPPAAL toolset. The implementation of the extension tool is based upon two fundamental aspects: (i) data structures that replicate the internal structure of an UPPAAL document and (ii) a functional behavior of the tool defined by the transformation rules.

The extension tool source code has been distributed through three Java packages. The main package\textsuperscript{2} contains all the relevant classes used by the REMES SOS design framework. This package contains adapter classes that expose the transformation functionality to the IDE tool. Classes replicating the UPPAAL data (static) structure are located in the models\textsuperscript{3} package. The core functionality of the extension that implements the transformation rules is distributed into the transformations\textsuperscript{4} package.

\begin{itemize}
\item[2] org.service.modelchecker.uppaal.
\item[3] org.service.modelchecker.uppaal.models.
\item[4] org.service.modelchecker.uppaal.transformations.
\end{itemize}
4.2 Static structure of the extension tool

In order to support the transformation of the REMES SOS modes into TA compatible with the UPPAAL model-checker, the extension tool has to provide the static structure that replicates data models from both UPPAAL and REMES SOS framework. Having the extension tool integrated as an add-on into the REMES IDE for modeling SOS enables it to use both the static structure and the functionalities of the host IDE. Opposite to this, to be able to export the REMES SOS modes into UPPAAL compliant TA, a structure that replicates the UPPAAL exported documents has to be created.

![Figure 4.1 Structure of a UPPAAL document.](image)

The diagram given in Figure 4.1 shows the structure of a UPPAAL document. This class diagram has been extracted from the XML file generated by the UPPAAL tool, and it captures all of the core entities upon which the internal structure of the UPPAAL compliant TA is built. Our extension add-on will create a file with the same exact structure as shown in the diagram, which can be used like an input to the UPPAAL tool. The root element of the document is the NTA element. The NTA
container element holds the definition for the global variables and constants of the UPPAAL TA model, as well as the instantiation of the separate timed automaton. The root element of the model is composed out of an arbitrary number of Template entities, each corresponding to a separate timed automaton. The internal structure of each Template object is composed out of Locations entities interconnected with edges represented by Transition objects. Both the locations and the edges can be decorated with properties, which are stored into objects of type Label.

In the REMES extension tool, we have recreated the structure of the UPPAAL system, by creating corresponding classes for all of the UPPAAL entities. The data classes corresponding to the UPPAAL entities are stored in the models package. This replication of all the entities in the UPPAAL exported system is critical to support the early design decision to serialize the UPPAAL TA model by using XML annotations. By recreating the UPPAAL data model, it is possible to automate the process of serialization and deserialization of system into XML files by using third parity libraries such as JAXB [4].

4.3 Functional aspect of the extension tool

In the following section we describe the functional aspects of the extension for transforming REMES SOS modes into an UPPAAL compliant TA. The implementation of the functionalities is based on the transformation rules presented in Chapter 3.

For each of the REMES SOS modes a separate transformation engine class that implements transformation rules for that particular REMES SOS mode has been created. The detailed class diagram of the translator classes is depicted in Figure 4.2.
Based on the characteristics of the transformation rules for REMES SOS modes, we have identified five major classes of REMES mode translators: translator for REMES atomic modes called AtomicServiceTranslator, translator for REMES atomic submodes – AtomicModeTranslator and translators for each of the composite REMES modes (composite, AND and OR REMES modes). Since the atomic REMES modes (atomic modes and submodes) do not share similarities in the transformation process, separate transformer classes have been created for them with different functional characteristics. The composite modes on the other side, share much similarities between each other. From the viewpoint of transforming the REMES SOS composite modes, all of them act like wrappers over atomic modes, with some distinctive individual characteristics. To improve the code reusability of the extension tool we have created the BaseCompositeModeTranslator class to hold all of the shared functionalities of the composite mode transformation classes. The specific functionalities of a particular composite mode have been implemented in the corresponding mode translators. Additionally translators for transforming the REMES atomic modes placed inside an AND/OR synchronized modes have been added as separate entities. In the following subsections, we dissect each of the transformation classes and explain their functionalities in detail.
4.3.1 Atomic mode translator

*AtomicServiceTranslator* class holds the implementation of the transformation rules for the standalone REMES SOS atomic modes, as presented in Chapter 3. Due to the simplicity of the transformation rules the class that wraps the functionality contains only three properties and one publicly exposed method which is the actual implementation of the transformation rule. To be able to extract all the relevant information for the service that is currently being translated, the instantiation of a translation object is allowed only by using the constructor that accepts the identification of the mode and the context in which it has been created. The context in which the REMES modes are created is from the base type *GraphPinScene*, which in the implementation of the REMES IDE toolset it also acts as a container for the REMES SOS modes.

The first step in the process of transforming the REMES atomic service is to create a new timed automaton represented by a *Template* object. This newly created automaton by default contains only the initial location marked as Start. Once the *Template* object has been created, we will further enrich it with a new location object that will correspond to the REMES atomic service that we are currently working on. Before adding the location object to the *Template*, it will be given the same name and identification as the corresponding REMES service. This is very important for the later phase of the transformation when the edges between the REMES modes are translated, as the algorithm uses the identifications of the modes to indicate the starting and the end point of the transition. The service properties are directly transferred as location properties in a form of *Label* objects. This means that invariant of the service becomes invariant of the location and also the location is marked as urgent if the corresponding flag of the mode has been set to true. The declaration of the variables, constants and the resources is added to the definition section of the UPPAAL compliant TA represented by a *Template* object.

When transforming the data properties it is very important to map them into a proper context i.e., map them either as local or global variables. The variable properties of the atomic mode are translated into local variables of the timed automaton of corresponding type. An important notice is that the extension does not validate the syntax of the modes, so the correctness is not guaranteed. The resource variable is translated into global variable of type integer. The imposed limitation is that only one resource variable of type integer is allowed in the system. The resource variable
is always mapped into a global variable called \textit{cost}. During the transformation process the data variables are stored in object of type \textit{DefinitionParsingResut}, which stores the already transformed data types in an array of clock, integer, constants, channel and broadcast channel. When the mode has been translated, the \textit{DefinitionParsingResut} property of the resulting transformation is mapped into a string variable by calling the \textit{ParseDefinitions()} method and added to the definition section of the corresponding timed automaton.

The last step in the process of transformation of REMES atomic mode is the process of synchronization of the mode execution with the other modes in the system. The mode has to be synchronized both upon the start and on the end of the execution. At the beginning of the execution, the mode is synchronized with a set of modes referred as \textit{predecessors}. This set contains all the modes that are source of the input edges of the mode under transformation. The function responsible for finding the predecessors of a particular mode is provided by the \textit{Utils} static class, which holds all the functionalities common for the implementation. For each of the modes in the set of predecessors an edge between the initial location and the location corresponding to the mode itself is created. This edge enables the execution of the current timed automaton to be triggered by any of the modes in the predecessors set.

After synchronizing the mode with the preceding modes, the transformation has to provide mechanism for synchronization of the timed automaton with the modes following the current one. To enable this, according to the rules presented in Chapter 3 for transforming atomic REMES modes into a network of TA a looping edge broadcasting synchronization channel has been created. If the mode from its output connection point is connected to more than one REMES mode, the created synchronization will be through \textit{broadcast channel}. If the mode has only one output connection the synchronization will be through \textit{binary UPPAAL channel}. In case when the mode does not have output connections, no synchronization channel is added on the looping transition.

\textbf{4.3.2 Atomic submode translator}

Transformation of the atomic REMES submodes is supported by the \textit{AtomicModeTranslator} class. The translator object is created with a constructor that requires the identification of the submode, identification of the parent composition mode of the submode and a \textit{GraphPinScene} object i.e. the context in which the
parent composite mode has been created, which is used to find all of the siblings of the composite mode.

The transformation of the REMES submodes differs from the other REMES SOS modes, because it is the only mode that does not result automaton represented by a Template object. The transformation of the REMES submodes results in a set of locations, edges and definitions which are stored in an object of type PartialTransformationResult. This is due to the fact that REMES submode is translated into one or more locations of a timed automaton corresponding to the parent composite mode. The resulting PartialTransformationResult object is very similar to the Template object as it holds almost identical information. The difference is in the semantics, which means that by creating this type of object at the end of the transformation process, it is assumed that the produced result has to be merged with the parent timed automaton represented by a Template object.

The process of transforming the atomic REMES submode starts by creating a corresponding location for the translating REMES mode. Once created, the location has been decorated with all of the required properties – the invariant of the mode becomes invariant of the location, the name of the mode is the new name of the location and the same identification as the mode is transferred to the location. The data properties of the mode are parsed and directly translated into declaration format of the Template object. After transforming the mode into a location, the transformation process continues with a test whether the submode belongs to the class of non-lazy modes. This is done by checking the urgent flag of the mode and comparing the invariant with an empty expression. If the mode has not been marked as urgent and the invariant predicate is an empty expression, according to the rules presented in Chapter 3 for transforming the non-lazy modes the timed automaton is enriched with a synchronization location for each of the outgoing edges. The synchronization location is a committed location which has been decorated with a similar name as the mode location. Our proposed naming convention is to add the suffix “_Sync” (eg. ModeName_Sync) to the mode identification. If the non-lazy mode has more than one outgoing edge, the transformation adds index at the end of the name of the committed location. The edge between the mode location and the synchronization action is additionally decorated with the non-lazy synchronization channel. If there are multiple outgoing edges from the non-lazy mode, all of them will be decorated with the same synchronization action. Instead of creating a looping
transition for each of the non-lazy submodes in a given composite mode, the timed automaton is enriched with only one looping edge sufficient to synchronize all the non-lazy modes in a given composite mode.

The last step in the process of transforming the REMES atomic submodes is to translate the output edges. During this step the output edges of the atomic submode are translated into a transition objects readable for the UPPAAL model-checker. In this stage of transforming the atomic submodes we are performing the elimination of the conditional connectors from the composite mode internal structure. This is accomplished by inspecting the target node of each of the atomic mode output edges. If the target mode is another atomic submode a new Transition object corresponding to the original edge is created. In a case when the target node of the edge is a conditional connector the transformation process enters in a recursive mode, and as a result it returns the set of output edges of the conditional connector. Each of the edges in the new set is then combined with the edge from the submode to the conditional connector. The source of the newly created transitions is the atomic submode preceding the conditional connector and the target node is the atomic submode that is target node of the corresponding output edge of the conditional connector. The process of transforming the edges into transitions combines the guards, actions and assignments of the original edges.

After the transformation of the atomic mode is complete, the PartialTransformationResult object is returned and the data present in it is merged into the parent timed automaton.

### 4.3.3 Composite mode translator

CompositeModeTranslator class is derived from the base composite translation class called BaseCompositeModeTranslator. The base class holds the properties and functionalities shared between the composite mode translators in the system. Also, the AND and OR synchronized mode translators are derived from this base class. Before presenting the details about the implementation of the CompositeModeTranslator class, we will give more detailed overview of the base class for all the composite modes.

The BaseCompositeModeTranslator class was introduced into the system to achieve better overall performance of the code. In the first place, we think about the code reusability and creating the hierarchy of classes to achieve simplicity in the
The BaseCompositeModeTranslator class is an abstract class, and it is used as a template for creating concrete implementations for transforming different types of composite modes. Even though the class has been created as an abstract, it holds the implementation of the shared functionalities between all the composite mode translators. For that purpose in the base BaseCompositeModeTranslator class we have put the implementation for translating the composite mode pins into locations of resulting timed automaton. Beside the implementation of the shared code in the BaseCompositeModeTranslator, the abstract class has one abstract method called TranslateMode() which returns a list of Template objects. The list of Template objects as a result object has been enforced by the transformation of the synchronized modes where a network of TA has been returned - one automaton for each of the atomic modes that has been synchronized.

The CompositeModeTranslator class holds the implementation of the transformation rules for the composite REMES mode composed of REMES atomic modes and conditional connectors. As defined in Chapter 3, the transformation of the composite modes is done in steps as following:

- translate composite mode pins;
- translate atomic REMES submodes into a corresponding timed automaton;
- remove the conditional connectors.

Translation of the atomic submodes inside the composite mode is performed by the AtomicModeTranslator class. The process of translating the inner modes from the viewpoint of the CompositeModeTranslator represents iteration through all of the submodes in the composite mode and creating a translator class for each of them. To get the list of child modes placed within the composition mode we use the getNodes() method provided by the GraphPinScene object of the CompositeMode class. This method returns a set of objects consisting out of composite mode pins, atomic submodes and conditional connectors inside a composite REMES mode.

The process of transformation of the composite mode continues by iterating through each of the objects in the list. In this phase of transformation it is important to determine the type of the child object in order to create the adequate translator class. The type of a mode is determined by using the MatchNode() method from the NodeMatcher class. The process of matching the mode returns NodeMatcherResult object which holds the information about the type of the matched mode. After the
type of the mode has been determined, the composite mode translator can create the corresponding translator class and call the \textit{TranslateMode} method which will return a set of locations, transitions and data variables from the translated submode. In the later phase of the transformation process this partial result is further parsed and merged into a timed automaton corresponding to the composite mode.

After the inner submodes of the composite mode have been translated into corresponding timed automaton, the next step is to synchronize the execution of the current automaton within the rest of the network. According to the definition for transforming composite modes presented in Chapter 3, we try to construct initialization edge in the resulting timed automaton between the Start location and the location corresponding to the Init connection point of the composite mode. If the Init connection point of the composite mode has not been translated into location, the transformation creates transition corresponding to the edge between the Start location and the location corresponding to the Entry connection point of the composite service. An initialization edge will be created for each mode to which the current composite mode has to be synchronized. In contrast to the atomic REMES modes which have only two connection points, the composite modes have four connection points, from which two are input and two output points. To obtain the set of modes to which the timed automaton corresponding to the composition mode has to be synchronized, the \textit{CompositeModeTranslator} passes the synchronization point identification to the \textit{GetCompositeModeConnectionPointSynchronizations()} method of the \textit{Utils} class. The class method returns a list of all the modes directly connected to that connection point. If the composite connection mode does not have any input connections, the edge will be decorated with a synchronization channel broadcasted from the global trigger automaton.

When the initialization synchronization of the timed automaton has been provided, depending on whether the mode has been synchronized to other modes in the system through its output synchronization points, the transformation adds synchronization action on the reset transition. The reset transition is created between the locations representing Entry and the Exit connection points of the composite mode. In case when the REMES composite mode has not been connected to other modes through the Exit synchronization point the synchronization action will not be added on the transition. If the Exit connection point of the REMES composite mode has exactly one output connection a binary synchronization channel will be added to the
transition channel. The last possible situation is when the Exit synchronization point has more than one outgoing edge. In this case the synchronization channel added for synchronization will be of type broadcast.

The final step in the transformation process is to remove the conditional connectors from the resulting timed automaton, and it is being handled by the \textit{AtomicModeTranslator} translator class. Due to the fact that the conditional connectors are part of the submode set, in the process of iteration through the submodes of a composite mode if a given submode is of type conditional connector the \textit{CompositeModeTranslator} just ignores the mode.

\subsection*{4.3.4 Synchronized REMES mode translators}

The \textit{ORModeTranslator} and \textit{ANDModeTranslator} classes provide the necessary means for transforming the AND/OR synchronized modes into a network of TA. Creation of translator objects for an AND/OR synchronized modes is possible by providing the identification of the synchronized mode and the context object in which the mode has been created. According to the transformation rules presented in Chapter 3, the transformation of synchronized modes produces network of TA. The primary difference in the semantics of the both synchronized modes is that the submodes inside the OR mode are of type \textit{RemesAtomicServiceORWidget} while in the AND mode are of type \textit{RemesAtomicServiceANDWidget}. To follow up the idea of code reusability and maintenance, we have created separate translator classes for both types of submodes present in the AND/OR mode. The \textit{RemesAtomicServiceANDWidgetTranslator} and \textit{RemesAtomicServiceORWidgetTranslator} are the classes that provide the underlying support for transforming AND/OR mode submodes according to Definition 1 and Definition 2 presented in Chapter 3.

The transformation process of both AND and OR modes is similar to a great extent, with only difference in the type of the submodes being transformed. For that purpose we present general overview of how the transformation rules for AND/OR modes have been implemented into code. The transformation of the synchronized modes starts with creating either \textit{ANDModeTranslator} or \textit{ORModeTranslator} class by providing identification and the \textit{GraphPinScene} object. The translator produces the network of TA by iterating through all of the submodes and calling adequate methods for transforming each of the submodes into a corresponding timed
automaton. Beside the service modes, the connection points of the AND/OR mode have also been implemented as child submodes. Since the transformation rules define that the synchronization mode itself should not enrich the resulting network of TA by any means, the synchronization point submodes have been ignored in the process of transforming the synchronized modes.

There are two java classes\(^5\) that provide the functionality of transforming REMES atomic modes inside a synchronized composition, that can be of type RemesAtomicServiceANDWidget and RemesAtomicServiceORWidget. The implementation and functionality of the both transformers for the synchronized REMES atomic modes match to great extent, with the difference that RemesAtomicServiceORWidgetTranslator have been enriched with additional variables used for synchronization.

The translator objects for synchronized REMES atomic modes can be instantiated by providing the identification of the REMES atomic mode, the identification of the synchronized mode wrapping the REMES atomic mode and the GraphPinScene object in which the synchronized mode has been created. After the translator object has been created, the synchronized mode transformation class calls the method for transforming the REMES atomic mode. According to the definition for transforming the REMES atomic modes, inside the synchronized mode the atomic modes can be divided into three major groups – REMES atomic modes connected to the output connection point that needs to be synchronized at the end of the execution, REMES atomic modes synchronized with the input connection point of the AND/OR mode and atomic modes that are not directly synchronized with either of the AND/OR mode connection points.

The transformation of the REMES modes that have to be synchronized with the Entry connection point of the AND/OR mode is done according to the Definition 1 presented in Chapter 3 for transforming REMES atomic modes. To find the modes on the external side of the synchronization point to which the REMES mode has to be synchronized, we use the method GetGlobalSyncServices() provided by the

\(^5\)The classes that provide the functionality are the RemesAtomicServiceANDWidgetTranslator RemesAtomicServiceORWidgetTranslator class, respectively.
synchronized REMES atomic mode translator class. The special case is when the synchronization mode is the first mode in a REMES service composition. In that case, the composition mode does not have any input edges and consequently to that, the REMES modes inside the AND/OR mode that are synchronized directly to the input synchronization point does not have any external synchronization to trigger their execution. The execution of the input synchronization REMES modes is triggered by the global trigger automaton synchronization. If the REMES atomic mode is directly connected to the output connection point of the synchronized mode, according to the Definition 2 the transformation must provide an additional mechanism for synchronizing the result automaton of that output REMES mode with other automatons within the TA network produced by Definition 1. The synchronization regime in which the synchronization mode operates does not influence the structure and the behavior of the proposed mechanism. As presented in Figure 3.15, the mechanism consists of additional committed location and a variable to count the current number of executed service compositions connected to the output connection point of the AND/OR mode called synchronization variable. The committed location on one side is connected with the location corresponding to the REMES atomic mode while on the other side is connected to the initial location via two edges. The definition imposes that the edges from the committed location to the initial location of the automaton have to be decorated with guards. One of the guards compares if the value of the synchronization variable with the total number of executing paths, and the other guard expression evaluates the opposite case. At the end of the transformation, a synchronization broadcast channel is added to the edge that evaluates when the current mode is the last output synchronized mode that has to be executed. The broadcast channel is used to synchronize the synchronized composition with the other modes in the system.
4.4 Integration with the REMES SOS IDE

The developed extension add-on for transforming the REMES SOS modes into TA has been designed to provide straightforward integration with the host IDE environment. To support seamless integration, all of the functionalities of the extension tool have been exposed through the `UppaalModelCheckerConnector` class.

![Extension tool integration into the main context menu.]

The extension functionalities have been integrated to the host IDE environment by creating new option in the context menu, as depicted in Figure 4.3. The designer can generate TA model of the previously designed REMES SOS model by selecting the adequate option menu.

To be able to provide some degree of independence from the host environment, the `UppaalModelCheckerConnector` class can be instantiated only by providing the root scene of the environment represented by the `GraphPinScene` object. This `GraphPinScene` root scene contains all the elements in the REMES SOS model. The `UppaalModelCheckerConnector` generates the resulting TA by iterating through all of the child elements of the root scene and translating each of them into a corresponding timed automaton.

After the REMES SOS model has been transformed into a TA network, the resulting transformation is then serialized directly into a XML document by using the JAXB library. The document is saved on the file system in a file previously selected by the user. Once exported, the XML file is ready to be imported into UPPAAL tool in order to be verified.
4.5 Discussion on the approach

Although our work relies on transformation rules defined in [9], the novelty that we have introduced is that transformation rules defined in our thesis have been adopted to work with TA framework instead of PTA framework. Further we have formalized and implemented new transformation rules for the AND and OR modes, which are specific for the SOS specification and represent a powerful mechanism for synchronizing the behavior of modes in the system.

So, therefore our approach facilitates the formal analysis of a richer class of systems compared to the previous work [9]. The entire implementation is fully compliant with the theoretical rules. The existence of the development tool for modeling REMES SOS models has had a big influence on the technical decisions for the implementation. The imposed constraint of enriching the IDE tool and not developing stand-alone tool from the scratch was possible because we had the source code of the original IDE. Another imposed constraint in the implementation part was that the core functionalities of the tool were not supposed to be altered or changed. All these constraints have resulted in developing a separate module in a form of an extension add-on for providing the formal modeling support. We have exposed the functionality of the module by using adapter classes which were used as a bridge between the host environment and the extension add-on. Later, the tool was integrated in the IDE in such a way that it provides straightforward usage for the system designers. To validate our work we have exemplify the described contribution on a real world problem in Chapter 5.

This chapter presents the technical details about our solution to introduce formal verification support to the REMES IDE. We have completed the part focusing on the problem analysis, theoretical foundation and the practical implementation of our proposed mechanism. In order to present and validate the ideas and contributions from our work, in the next chapter we present an illustrative example in a form of SOS model of an Automated-Teller Machine.
5 An illustrative example

In this section we give an example through which we present the functionalities of the extension for connecting the REMES design framework with the UPPAAL toolset for formal verification and validation. For that purpose we will consider an Automated-Teller Machine (ATM) similar to the one used present the capabilities of the REMES design framework [7]. The example has been modified in order to show the newly introduced features of the extension tool. The ATM machine represents a typical case of safety critical real-time system characterized with a high degree of complexity and elaborates interactions with different hardware resources. The example depicts a typical scenario for an ATM system in which:

- customer inserts card;
- the ATM machine ask user to enter pin code;
- ATM machine verifies the pin and logs the user in the system;
- ATM machine displays account information;
- user enters sum to withdraw from the ATM;
- the sum is checked and the adequate action is taken;
- ATM prints ticket and shows new state of the balance;
- and so on.

The interaction between the customers and the ATM machine has many possible scenarios and this is just one typical case. Other scenarios may include actions where users enter invalid pin code, invalid amount for withdrawal, etc. The various user scenarios can be achieved through different composition of services available in the service repository presented in Figure 5.1. As presented in the same figure, the services can be invoked and composed in different ways to comply with the users’ needs.
5.1 Service repository

![Figure 5.1 ATM machine service compositions](image)

The available services are contained within the service repository (see Figure 5.1) represented as a collection unit that enables users to obtain and modify the interfaces of the services to which they have access. Through the service interface the invocation and flexibility of the services has been provided to the users. In the following illustrated example we present a simple scenario of user interacting with the ATM machine. The scenario includes some basic functionalities of the ATM such as login, account check, withdrawing money, display of errors occurring during the process and logout. Having the specific goal of presenting the transformation process, the modeled scenario does not include the entire set of functionalities of an ATM machine such as the communication between the ATM machine and the bank. The SOS model under consideration consists out of two composite modes, three atomic modes, AND synchronized mode and parallel connection points.

For the initial step in our defined scenario the users are required to prove their identity. The LOGIN composite service (see Figure 5.3) captures this functionality of the ATM machine. It consists out of two atomic modes called INSERT_CARD and TYPE_PIN. Once the user inserts card into the machine, the INSERT_CARD service validates its presence and sets variable card to 1 (true) to indicate the cart is in the machine. The user is than redirected to a new screen where pin code is required to
Once the user enters a valid pin code, the TYPE_PIN service will set the pin and log variables to 1 indicating that correct pin code has been entered and that the user has been successfully logged into the system.

The ACCOUNT_INFO mode in Figure 5.7 is presented as an atomic REMES mode. It can be executed only if the user is logged in (log == 1). This service shows the user account information together with all the completed and pending transactions as well as the current balance of the bank account.

The MONEY_WITHDRAW composite service depicted in Figure 5.5 captures the functionality of users withdrawing money from the ATM machine. It is composed out of two atomic modes TYPE_SUM and GIVE_SUM. The users can use this service only if they have been previously logged into the system (log == 1). The TYPE_SUM service waits for fixed amount of time for user to enter the desired sum. Once entered, the sum is placed into a global variable called sum and the execution of the scenario continues with the GIVE_SUM atomic mode. Before entering the GIVE_SUM atomic service which performs the function of giving the amount of cash to the user, the composite service makes validation of the entered sum versus the current state of the balance. Only in a case of a positive balance the GIVE_SUM service is executed. In every other scenario, the execution proceeds directly to the exit connection point of the MONEY_WITHDRAW composite mode i.e. the ATM does not issues cash to the user.

The ERROR atomic service (see Figure 5.8) deals with the incorrect information provided by users upon the login procedure. The mode has been marked as urgent in order to process the incorrect login information once it has been evoked.

FINALIZE_TRANSACTION AND synchronized mode in Figure 5.11 represents the machine behavior after the user has passed the procedure of withdrawing money. It consists out of two atomic modes BALANCE_DISPLAY and PRINT_RECEDE responsible for displaying the new account balance and recede print after the transaction.

LOGOUT service (see Figure 5.14) is the last service in the scenario. It is responsible for terminating the connection (interaction) with the user and ejecting the card back.
Figure 5.2 A service composition for the ATM system modelled in the design framework.
To explain the transformation of each of the modes in the composition presented in Figure 5.2, in this example we will follow the transformation process as it iterates through the modes one by one. First in a queue for transformation is the composite REMES mode named LOGIN. As presented in Figure 5.3 the inner structure of the mode is composed of two atomic modes named INSERT_CARD and TYPE_PIN.

![Figure 5.3 Login REMES composite mode](image)

Illustrated in Figure 5.4 is the resulting timed automaton generated from transforming the LOGIN REMES composite mode presented in Figure 5.3. According to the transformation rules for transforming composite REMES modes defined in Chapter 3, the LOGIN composite mode is translated into a timed automaton where
the submodes are translated into locations of the automaton. Since the LOGIN composite mode is the first mode in the composition, its execution has been triggered by the global triggering signal depicted as \textit{start?}. Once the execution of the automaton has been triggered, the execution proceeds from Start to the EntryPoint location. If we focus on the synchronization points of the composite mode (see Figure 5.4), it is clear that only the Entry and the Exit synchronization points of the composite mode have been translated into locations. The reason behind this is that the other two connection points (Init and Write) of the composite mode do not have input or output connections associated with them. From the EntryPoint location the automaton execution continues into the \textit{INSERT\_CARD} location. The location has the same name as the REMES atomic mode and the invariant of the mode is directly translated into the invariant of the location. Observe the outgoing edge from the \textit{INSERT\_CARD} location in Figure 5.4. It contains the same guard as the original edge in the REMES model. Through this edge the execution of the timed automaton continues to the next location which is \textit{TYPE\_PIN}. The Exit output synchronization point has been translated into a corresponding location, which according to the definition for transforming composite modes has been marked as urgent (ExitPoint decorated with \textit{U} in Figure 5.4). The edge from the ExitPoint location to the input EntryPoint location represents the reset edge used to reset the automaton. Since the LOGIN REMES composite mode is directly synchronized with other REMES modes upon exit, the edge is additionally decorated with a synchronization channel. As presented in Figure 5.4 the LOGIN mode is synchronized to more than one other mode upon exit, so according to the UPPAAL semantic that synchronization channel must be defined as broadcast.

Next to transform are the modes placed inside the parallel connection. As defined earlier by the transformation rules in Chapter 3, the result from transforming a parallel connection is a network of TA. In the above presented example, there are four modes named \texttt{BALANCE\_DISPLAY}, \texttt{MOONEY\_WITHDRAW}, \texttt{ERROR} and \texttt{FINALIZE\_TRANSACTION} that are connected in parallel. In the remaining of section we present the resulting TA network produced from these four modes inside the parallel connection and give more detailed explanation how the result has been influenced by the parallel connection.
Figure 5.5 MONEY_WITHDRAW REMES composite mode

![Diagram of MONEY_WITHDRAW REMES composite mode]

Figure 5.6 The timed automaton generated from the MONEY_WITHDRAW REMES composite mode.

The timed automaton corresponding to the MONEY_WITHDRAW REMES mode (see Figure 5.5) is illustrated in Figure 5.6. From the resulting automaton we can see that the execution of the mode has been triggered by the LOGIN composition mode. Although the mode is not directly connected to the LOGIN mode, it has been activated through the parallel composition connector. Since arbitrary number of REMES modes are allowed to have the same name, we use their identifications to
name the synchronizations. By practicing this, we guarantee that the modes are properly synchronized. Specific for this composite mode is the presence of a conditional connector in its internal structure. As depicted in Figure 5.5 the conditional connector is placed between the \textit{TYPE\_SUM} and the \textit{GIVE\_SUM} modes, and its output edges are decorated with guards. According to the definitions for removing conditional connectors from the REMES composite mode inner structure presented in Chapter 3, the conditional connector has been replaced by a set of edges. The new set of edges is composed by permuting all the input and output edges of the conditional connector and merging them together. In our example, we can see that the \textit{TYPE\_SUM} location (see Figure 5.6) is directly connected to the \textit{GIVE\_SUM} and the ExitPoint locations. Depending on the current state of the balance and the amount that user wants to withdraw, the execution will either continue in the \textit{MONEY\_WITHDRAW} location, or it will skip this location and directly go to the ExitPoint location. Because the composite mode is followed by other modes in the system (see Figure 5.2), the reset edge between the ExitPoint and EntryPoint locations is decorated with synchronization channel.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{ACCOUNT_INFO.png}
\caption{ACCOUNT\_INFO REMES mode}
\end{figure}
The TA corresponding to the ACCOUNT_INFO (Figure 5.7) and ERROR (Figure 5.8) modes are presented Figure 5.9 in and Figure 5.10, respectively. Since they are connected in parallel with the MONEY_WITHDRAW mode, both are activated at the same point of time by the same broadcast channel. The TA has been generated in
compliance with the transformation rules for transforming REMES atomic modes into timed automaton presented in Chapter 3. Each automaton consists of one location representing the mode (e.g. ERROR location in Figure 5.10) and an initial location (e.g. Start location in Figure 5.10). Since both of the modes are connected to another mode in composition through the parallel connection the reset edges are decorated with synchronization channels.

**Figure 5.11** AND synchronized mode composed of BALANCE_DISPLAY and PRINT_RECEDE atomic modes.

After the transaction has been processed and the requested sum has been issued to the user, the ATM machine displays the new balance and prints the recede (ticket). In our example we assume that these two atomic operations are AND synchronized (see Figure 5.11). Both of the services are placed within AND synchronized mode which is set to operate in a **max** regime. This means that the execution of both services will start simultaneously and the execution of the synchronized composition will finish when the slower mode finishes execution.
The TA corresponding to the BALANCE_DISPLAY and PRINT_RECEDE atomic REMES modes placed inside AND synchronized mode are presented in Figure 5.12 and Figure 5.13, respectively. The only difference between the two automatons is the name of the location representing the atomic mode and the name of the
corresponding committed locations. The start of the execution of both automatons is synchronized with the previous REMES mode in composition i.e. MONEY_WITHDRAW (see Figure 5.2). Due to the fact that both of the modes are connected to the output connection point of the AND mode (see Figure 5.11) their execution needs to be synchronized with the execution of the AND mode. The synchronization mechanism of the timed automaton presented in Figure 5.12 and Figure 5.13 consists of an additional committed location and a variable that counts the number of executed modes. Since the synchronized mode is of AND type, it is guaranteed that both of the modes will start their execution in the same point of time. Because of this, the guard on the edge between the committed and the Start location (see Figure 5.12 and Figure 5.13) compares the number of executed modes with a constant. The constant value is calculated and it corresponds to the number of atomic modes directly connected to the Exit connection point of the AND mode. Depending on the value of the counter variable, the corresponding transition will be activated. The guard expression ANDMode_SYN_VAR >= 2 (in Figure 5.12 and Figure 5.13) will evaluate to true for the mode that is executing last in the submode lane. Upon the execution of the transition corresponding to the previous guard, the automaton will broadcast synchronization action to signal the end of execution of the AND synchronization mode and to trigger the execution of the next mode in the composition.

Figure 5.14 LOGOUT REMES atomic mode
Figure 5.15 Timed automaton corresponding to the LOGOUT REMES mode.

Figure 5.15 depicts the timed automaton generated for the LOGOUT REMES mode presented in Figure 5.14. The logout is the last mode executed in the example scenario. The distinctive feature of this mode is that it has been directly connected to a parallel connector. Due to that, the initialization of the timed automaton is different compared to the other automata in the system. In the Figure 5.15 we can clearly see that the initial timed automaton consists of three edges, each decorated with activation synchronization from the REMES modes preceding the current mode. Although the LOGOUT mode is not directly connected with the rest of the modes, the activation has been transmitted through the parallel composition connector. Observe the reset looping edge of the LOGOUT mode in Figure 5.15. Since the LOGOUT mode is the last mode in the composition (see Figure 5.2), it is not connected with other modes upon exit. Because of this, the looping reset edge has not been decorated with synchronization actions.

By applying our prototype implementation to an example SOS model we presented and validated its functionalities. We have shown how REMES SOS model has been transformed in phases into a TA framework suitable for model-checking with the UPPAAL tool. In Chapter 5, we present the related work by comparing different approaches for transforming behavioral models into formal models.
6 Related work

The fact that the SOS services are published, invoked, composed and destroyed at the user demand brings the problem of ensuring the QoS into focus. In the literature, there are several SOS approaches that facilitate formal analysis of SOS in order to ensure the QoS, and in most of the cases this process is not straightforward. In this section we elaborate a number of approaches that are focusing on providing assets for the formal analysis to the current frameworks for modeling SOS.

To our knowledge, there are no other approaches that focus on providing formal analysis support to the REMES SOS language, so we will focus on a number of different related research endeavors that aim to introduce formal modeling and verification to other SOS modeling frameworks described in Chapter 2. Most of the below listed methods rely on several techniques for transforming SOS models into forms suitable for formal analysis, therefore we will compare our work to some of the most relevant of such methods. Due to its features that enable behavioral modeling of services as extended hierarchical state-machines, REMES is amenable to automatic generation of analysis models as TA networks, while guaranteeing high level of correctness of the resulting formal model.

The SENSORIA research project [18] has been focused on describing a pattern language for service engineering augmented with the formal analysis, transformations and dynamicity. The aim of the project is to provide a method through which models described using high level modeling languages [30] [29]. Based on the work published by the SENSORIA project, Bocchi et al. [20] have provided an extension to the SRML, a modeling language for service-oriented computing with primitives that allow capturing, modeling and analyzing typical time-related service-level agreements. The developed framework provides mechanism for capturing different kinds of time delays that occur during the service provision. In their work, the authors attempt to provide a method for model validation against a number of time constrained requirements and possibly provide structural improvements of the system in order to meet the specified requirements. To enable this, the approach facilitates transformation of models specified in SRML into Performance Evaluation Process Algebra (PEPA) [21] that enables quantitative analysis of timing properties. Compared to our work, the framework developed by Bocchi et al. is available only in a form of guidelines for human experts to follow in
order to create formal mode. Although it provides well defined mechanism for
capturing the multiple functionalities like describing the behavior of the SRML
design elements, compared to our approach the framework still lacks implementation
meaning that it does not provide an automated support for the formal analysis to the
system designer.

The lack of the formal analysis support of BPMN as one of the most popular
approaches for modeling SOS has been in focus of many research projects. Several
approaches have been introduced to bridge this gap and provide the formal analysis
features to the BPMN. The method proposed by Dijkman et al. [31] defines
semantics for a large subset of BPMN models by mapping the BPMN models into
plain Petri Nets. The developed solution provides a high level of expressiveness by
formalizing the basic building elements of the BPMN as well as the formalization of
the synchronization of the execution. In contrast to our approach where we offer
transformation for the complete set of the REMES SOS constructs into the formal
framework (into TA), the approach of Dijkman et al. does not support formalization
of a number of BPMN elements. On the other side, this approach is very similar to
ours as it focuses on providing theoretical foundation for transformation and then
building a prototype implementation to automate the process and valorize the
contributions. Another difference from our approach is that they have developed
stand-alone tool not integrated into a modeling tool, thus the level of automated
transformation is not the same as ours. To overcome the limitations of the previous
approach, Prandi et al. [14] provide translation of BPMN models into a Calculus of
Orchestration of Web Services (COWS) [16] that can be used as an input to the
PRISM model-checker [17]. The approach offers a mechanism for translating both
the BPMN elements and their behavior, be that synchronized or not. There are
number of similarities such as the high level of expressiveness of the formal modes
between this and our approach. However, the main difference from our approach is
that Prandi et al. have provided only a framework to the system designers and
verification experts to consult in order to achieve high degree of correctness and
performance of the formal model. The approach is not supported by a tool that would
provide an automated transformation of the BPMN models into COWS.

Bryans et al. [26] have proposed an approach to enable algorithmic translation from
BPMN into Event-B [27] – a widely used formal language for supported by the
Rodin platform [28] extensible through various plugins that provide various analysis
features like simulation and model-checking, as well. The approach provides coverage for a large portion of the set of features and it can be fully automated. This approach is very similar to ours as it supports the formal modeling of the core elements of the BPMN as well as for the orchestration of the services. Similarities with our approach can be found in the technical part of the approach, as the authors are considering to develop a plugin to an existing platform in order to provide automated transformation.

Foster et al. [23] present an approach that facilitates transformation of web service compositions in a form of BPEL4WS services into a Finite State Processes (FSP) which is later translated into labeled transition systems suitable for model-checking using various tools. The approach introduces technique for model-checking service compositions under specific resource constraints. In its native form the approach provides assets for high accuracy in mapping BPEL process into FSL as well as the orchestration activities. With the recent upgrade a support for mapping interactions to resource allocation activities has been introduced. Similar to our proposed approach, the transformation from BPEL4WS into FSP has been automated and added as a plugin to an already existing tool which supports editing of BPEL and FSP models. This approach, however, has a limited representation of the underlying model semantics and it is based on a number of assumptions. The other aspect in which this tool differs from ours is that it is applicable only under a number of resource constraints.

V. Raffe [22] introduces an approach for analyzing complex systems specified by Graph Transformation Systems (GTS) using scenario based model-checking techniques. The technique is based on the previous work of Baresi et al. [25] for style-based approach for modeling SOS, and it transforms GTSs with different properties into a BIR1 language that serves as an input of the Bogor model-checker [24]. Although the approach offers optimizations in analyzing different critical scenarios before the implementation, it still has limitations. The biggest drawback of the approach is that it does not represent a general approach due to the lack of support for generating and analyzing every property of a given model, which is not a case in our approach due to the properties of the REMES language.

Nakajima [12] has used the Web Service Flow Language (WSFL) [11] to provide means for describing the Web service aggregation. The concept of aggregation provides assets for creating web service compositions composed out of many
services, where each of the services in the composition is possibly provided by a different provider. By using the WSFL language, Nakajima has generated description flow for the service composition. With the description flow, the verification of service compositions is becoming very similar to verification of business flow that can be verified using the SPIN model-checker [13]. The verification process is performed in two steps as follows: first the service composition flow written in WSFL is translated into PROMELA process which serves as an input of the SPIN model-checker; second the operational semantics of the model are encoded by the PROMELA process. The proposed model provides mechanisms to model properties such as reachability, deadlocks, or application specific progress properties. The drawback of this approach compared to ours, is that the WSFL specification document does not provide information about rules on the use of the control and data links. This opens a door for introducing errors into the WSFL model description. This can affect the newly created models to behave differently depending on the link data.

7 Conclusions and Future work

In this thesis we have presented an approach for automated transformation of REMES SOS models into a TA suitable for verification with the UPPAAL tool. To support that, the definition set defined by Orlić et al [9] for transforming REMES ES models has been modified to comply with the TA framework and enhanced with additional transformation rules that define how the REMES SOS specific constructs should be transformed into TA. We have also implemented the transformation rule set into an existing REMES SOS IDE, thus enabling the designers to automatically generate TA models. By automating the process of transforming REMES SOS models into TA, one can expect faster way to get to the verification and validation of a given behavioral model. Although the process does not require direct assistance from outside, it is highly desirable that the model is verified by a verification expert before performing a model-checking with the UPPAAL tool.

In addition to the extension for transforming REMES SOS models into UPPAAL compliant TA, we have also adjusted the GUI and added new backend features of the REMES SOS IDE. To make the visual representation of the REMES SOS models more intuitive and similar to the graphical representation of the REMES ES models, as well as number of changes regarding the synchronization points of the REMES SOS
modes and the visualization of their properties have been made. The most significant feature introduced in the original IDE is the support for direct export of REMES SOS modes into vector graphics.

The current state of the implemented tool provides a communication channel between the REMES SOS and UPPAAL tool in a form of XML file. The process of creating the formal model and storing it into a XML file is fully automated, but the process of validating the model is not. For that purpose, to be able to visualize the generated TA, the verification expert must start instance of the UPPAAL verification tool and import the generated TA. The future work will be focused towards including the REMES verification action in the REMES IDE. When provided with this feature, the REMES IDE tool will become a complete tool for designing SOS systems with automated support for the system designers in the process of formal verification of the REMES SOS models.
8 References


