Mälardalen University
School of Innovation, Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science (60 credits) in Computer Science with Specialization in Software Engineering 15.0 credits

A FORMAL ANALYSIS FRAMEWORK FOR EAST-ADL ARCHITECTURAL MODELS EXTENDED WITH BEHAVIORAL SPECIFICATIONS IN SIMULINK

Paolo Shestani
psi17001@student.mdh.se

Vasja Çollaku
vcu17001@student.mdh.se

Examiner: Professor Marjan Sirjani
Mälardalen University, Västerås, Sweden

Supervisor: Post Doc. Raluca Marinescu
Mälardalen University, Västerås, Sweden

January 16, 2019
Abstract

Model-Driven Development is a development approach which is being used frequently in the automotive context in order to design models. EAST-ADL is an architectural language which models systems according to their architectural features, whereas Simulink is a tool environment which models systems according to their behavior. In this thesis work, we propose a set of transformation rules that take into consideration the EAST-ADL architectural model details and the behavioral specifications in Simulink, and generate a formal model, which can be verified UPPAAL model checker. Moreover, we implement these proposed transformation rules in a tool that automates them. The transformation rules proposed in this thesis work would be implemented for every EAST-ADL file with Simulink behavior specifications, generated by the MetaEdit+ tool. Properties like timing constraints, triggering and hierarchy in both EAST-ADL and Simulink have been considered by the transformation rules. Finally, the Brake-by-Wire case study is used to validate the tool and assess the mapping of the elements.
Acknowledgments

The most important gratitude belongs to our supervisor, Raluca. Thank you, Raluca! We started this journey together in a peculiar way and we did not make it any easier for you, but you still helped us and guided us through everything. You did not give up on us and was a real mentor.

Thank you professor Elinda! You had faith in us and believed that we would make it, and we did not let you down. You lead us toward a bright future and will always be grateful to you!

Thank you Mlardalen University! It was an amazing experience being a student here. We are thankful of what you taught us and the opportunities that you offered us, so we hope to have made you proud and that we can be good ambassadors.

A more personal touch...

I would like to express my very profound gratitude to my family for providing me with unfla-
ing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you. - Paolo Shestani

Thank you mum and dad and chuli! You are the only ones who have my heart. Thank you for walking this path with me. It has always been the four of us and that is the thing I am most grateful of. I am who I am today just because of you. I love you. Thank you Odeta for being a cousin, a teacher, a supervisor, a guider. You were a lighthouse when I was in the shadows and that, I shall never forget. Last but not least, I thank myself. Good job, Vasja, you did it! Now, go and conquer the world. - Vasja Çollaku
# Table of Contents

1 Introduction .............................................. 8  
   1.1 Problem Formulation ................................. 9  

2 Related Work ........................................... 10  
   2.1 Formal Analysis of EAST-ADL models .................. 10  
   2.2 Formal Analysis of Simulink models .................. 10  
   2.3 Reflection on the state-of-the-art of formal analysis of EAST-ADL and Simulink models ................. 11  

3 Preliminaries ........................................... 12  
   3.1 EAST-ADL ............................................ 12  
      3.1.1 EAST-ADL Notations ............................... 12  
      3.1.2 Modeling Tool .................................... 13  
   3.2 Simulink ............................................. 14  
      3.2.1 Simulink notations ............................... 15  
   3.3 Model Checking ....................................... 15  
      3.3.1 Timed Automata Notations ......................... 16  
      3.3.2 UPPAAL ........................................... 17  

4 Research Method ......................................... 20  
   4.1 Research Method steps ............................... 20  
   4.2 Threats to Validity ................................... 22  

5 Transformation Rules ..................................... 23  
   5.1 Proposed Transformation Rules ....................... 23  
   5.2 Hierarchy ............................................ 26  
      5.2.1 EAST-ADL without hierarchy ..................... 26  
      5.2.2 EAST-ADL with hierarchy ......................... 26  
      5.2.3 Simulink without hierarchy ...................... 28  
      5.2.4 Simulink with hierarchy ......................... 28  
   5.3 Additional information ............................... 29  
   5.4 Proposed Transformation Rules applied on examples ........................................... 30  
      5.4.1 On the example of a system composed of two sequential FunctionPrototypes of a system ..................... 30  
      5.4.2 On the example of a system with Simulink hierarchy ........................................... 31  
      5.4.3 On the example of a system with EAST-ADL hierarchy ........................................... 32  
      5.4.4 On the example of a system with event triggering and multiple instantiations of a FunctionType ........................................... 33  

6 Tool Implementation ...................................... 47  
   6.1 First tool development ............................... 47  
   6.2 Final tool development ............................... 47  

7 Brake-by-Wire Case Study .................................. 50  
   7.1 A short introduction to the Brake-by-Wire case study ........................................... 50  
   7.2 An insight on the utilized Brake-by-Wire case study ........................................... 50  

8 Results .................................................. 58  
   8.1 The tool validation with a simple, small-scale system ........................................... 58  
   8.2 The tool validation with a small-scale system with Simulink hierarchy ......................... 58  
   8.3 The tool validation with a system with EAST-ADL hierarchy ........................................... 59  
   8.4 The BBW case study validation ........................ 59  

9 Discussions ............................................. 75
10 Conclusion
  10.1 Limitations ........................................ 76

11 Future Work ........................................ 77

References ........................................... 80
List of Figures

1. An overview of EAST-ADL Abstraction Levels ................................................. 13
2. Representation of EAST-ADL notations .......................................................... 14
3. Simulink model of a system whose behavior is represented by atomic blocks .... 16
4. The light system model in UPPAAL ............................................................... 18
5. Syntax declarations of the light system model in UPPAAL ............................ 18
6. The methodology used in the thesis work ....................................................... 20
7. A template of the network of TA produced by the transformation rules for a simple EAST-ADL model, with behavioral specifications in Simulink ............. 23
8. A template of the network of SystemInterfaceComponent TA produced by the transformation rules for a hierarchical EAST-ADL model, with behavioral specifications in Simulink ......................................................... 26
9. A template of the network of TA produced by the transformation rules for one of the subsystems which comprise an EAST-ADL model, with behavioral specifications in Simulink ............................................................. 27
10. A network of SystemBehavior TA generated by the transformation rules on an EAST-ADL model with hierarchical behavioral specifications in Simulink ............................................................... 29
11. First example system, modelled in EAST-ADL with behavioral specification in Simulink ........................................................................................................ 34
12. The network of TA produced by the transformation rules for the first example ............................................................... 35
13. EAST-ADL and Simulink specifications for the system depicted in the second example ............................................................... 36
14. The network of TA produced by the transformation rules for the second example for the first FunctionPrototype ......................................................... 37
15. The network of TA produced by the transformation rules for the second example for the second FunctionPrototype ............................................... 38
16. EAST-ADL specifications of the third example system ..................................... 39
17. SystemInterfaceAverage TA of the third example ......................................... 39
18. The network of TA produced by the transformation rules for the third example for the first FunctionPrototype ......................................................... 40
19. The network of TA produced by the transformation rules for the third example for the second FunctionPrototype ......................................................... 41
20. The network of TA produced by the transformation rules for the third example for the third FunctionPrototype ......................................................... 42
21. EAST-ADL specifications of the fourth example system .................................. 43
22. The network of TA produced by the transformation rules for the fourth example first FunctionPrototype ......................................................... 44
23. The network of TA produced by the transformation rules for the fourth example second FunctionPrototype ......................................................... 45
24. The network of TA produced by the transformation rules for the fourth example for the third FunctionPrototype ......................................................... 46
25. The graphical user interface of the implemented tool ....................................... 47
26. The architecture of the tool ........................................................................... 49
27. The Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool ............................................................... 52
28. The Timing model of the Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool ............................................................... 53
29. The Behavior graph of the Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool ............................................................... 54
30. Simulink model of the BBW system ................................................................ 56
31. The Simulink behavior of each FunctionType .................................................. 57
32. Pictures that exhibit the verification of ‘not deadlock’ query in the first model .... 62
33. Pictures that exhibit the verification of the second example ............................ 63
34. Pictures that exhibit the verification of the third example ............................... 63
35. The network of TA generated for HW_BrakePedalSensor FunctionPrototype ............................... 64
36. The network of TA generated for LDM_BrakePedal FunctionPrototype ............ 65
37. The network of TA generated for BrakeTorqCalc FunctionPrototype ............... 66
List of Tables

1. Light system properties verified by UPPAAL .............................................. 19
2. List of FunctionTypes and FunctionPrototypes in the BBW model .................. 55
1 Introduction

Model-Driven Development (MDD) \cite{1} has been proposed as an effective way of managing system complexity and providing a basis for high-quality design and implementation of large scale systems. It mostly focuses in constructing software models before code is generated in order to reduce the amount of work done by developers and have a better planning phase of the projects.

Model transformation and traceability between different levels of abstraction make it possible to model complex systems starting from high level system architecture, specified in architectural modeling languages like EAST-ADL, AADL, SysML, proceeding with the implementation, represented through code generated from models of tools like Simulink, Scilab, IEC 61131-3 etc.

High level system architecture \cite{2} refers to the representation of a system depicting the high level structures which it is composed of. These structures portray different features of the system, providing an outline of it, along with information on how the structures interact with and relate to each other. EAST-ADL \cite{3} is an architectural description language which provides design artifacts of system models, by capturing information like functions, timing, triggering. MetaEdit+ is the tool chosen for this thesis work which implements the EAST-ADL language. EAST-ADL language has notations only for structural elements, and it has no notation for the behavioral information. However, the language allows for the model to be extended with a set of additional models that describe the behavior of the EAST-ADL components. These additional models can be created using other tool, for example Simulink.

Simulink \cite{4}, a MATLAB extension, is a graphical programming environment which applies the concepts of Model-Driven development in the modeling, simulation, verification and code generation of dynamic systems. One of its main functionalities is the modeling and simulation of time-dependent system component behavior, which is used by EAST-ADL language to extend behavioral specifications of the modeled system.

Beyond planning, modeling and developing in the engineering field, an important concept that we will focus a lot in this thesis is model checking. Model checking \cite{5} refers to the technique of formally verifying the correctness of the modeled system by exhaustively exploring the model, in order to reveal unexpected possible interactions between components of the systems model, and hence reveal potential flaws in the actual system. The model checker needs a requirement/property, formalized as a TCTL query, for it to exhaustively explore the model to decide if the property is satisfied or not. Systems nowadays are becoming more and more complex, thus model checking is becoming crucial in industry for safety and economical reasons. Each tool, implementation or system is virtually model checked before “coming to life”. It is checked whether the upcoming system is going to fulfill all the predefined requirements. It checks if it is safe to use, and tries all the possible scenarios of using it first in a virtual environment and then in a real life example. UPPAAL \cite{6} is a model checker used to model, simulate and verify real-time systems modeled as timed automata, developed as a joint effort between Uppsala University, Sweden and Aalborg University, Denmark.

In this thesis work, we take the high-level EAST-ADL architectural model together with all the Simulink behavioral model and we transform them into a network of timed automata that can be model checked with the UPPAAL verification tool. The goal of our work is to come up with some valid transformation rules and automate the transformation of the input EAST-ADL files into UPPAAL readable files by implementing this transformation chain in a tool. This is an issue that has not been addressed before. Up to our knowledge, some research work and minor developments have been made in this field regarding EAST-ADL transforming into Timed Automata.

The ongoing pursuit for advanced embedded system technology and safety improvements has made the “bywire” systems gain ground in machine manufacturing. Its origins are found in air craft and military designs, however nowadays it is a trend in autonomous vehicle manufacturing. The flexibility and optimization this technology brings, together with the energy savings and sim-
plicity in installation are some of the main advantages. It is the electric components that this technology brings that control the brake pressure, instead of any hydraulic fluid. Thus, we have chosen a Brake-by-wire case study to support the work and to validate our thesis work.

The remainder of the report is organized as follows. The following subsections provide a background on this thesis work and take a further look into the goal of this thesis work. Section 2 gives an overview on the state-of-art of this topic and identifies gaps in the existing work. Section 3 presents the reader with the basis for getting familiar with the languages and tools used in this thesis work. Section 4 outlines thoroughly the research methodology and development approach used to conduct the thesis work. Section 5 elopes the proposed transformation rules, followed by Section 6 where the tooling implementation effort of automating the transformation rules is described. The Brake-by-Wire case study is explained in Section 7. Section 8 describes the results of the case study, which are later discussed in Section 9. The conclusion of the thesis work and future work are explored in Section 10 and 11, respectively.

1.1 Problem Formulation

The combination of an EAST-ADL model and the set of corresponding Simulink models provides an integrated system model that encapsulates different information about the system. EAST-ADL provides information about the structure of the system and extra-functional constraints (execution time, end-to-end deadlines), whereas Simulink provides information about the behavior of the components. To provide evidence about the correctness of the design of the system, both representations need to be merged and transformed into a formal model that can be subject to formal (exhaustive) verification. Timed Automata is the chosen formal model and UPPAAL serves as the formal verification tool.

The main focus of the paper is to provide a verification framework based on UPPAAL for EAST-ADL models extended with behavioral specifications in Simulink. In order to implement this framework, we need to implement a transformation from the EAST-ADL and Simulink model to a network of timed automata, transformation that takes into account the following:

- Both EAST-ADL and Simulink are hierarchical models, while timed automata is a flat model, thus the transformation rules should provide a flattening algorithm that preserves the semantics of the original model;
- The verification is done on the timed automata model and not on the original model, thus a traceability mechanism should be put in place to ensure that any error found in the formal model can be traced back on the original model;
- To avoid the state space explosion problem, the transformation should also include different modeling choices that could reduce the state space during exhaustive verification.

The proposed transformation will be evaluated on an industrial use case, called the Brake-by-Wire prototype. Different functional and timing requirements are verified with UPPAAL model checker.
2 Related Work

This is not the first time that research has been conducted towards the analysis of industrial systems modeled in EAST-ADL and Simulink specifications, due to: (i) the increasing interest of automotive companies in EAST-ADL and (ii) Simulink being the de facto modeling tool for automotive systems. In order to present the related work, we divide our literature review into two parts that will reflect previous investigations on the formal analysis of EAST-ADL and Simulink models, respectively.

2.1 Formal Analysis of EAST-ADL models

Closely related to our work, Marinescu et al. [7] present different analysis techniques for EAST-ADL architectural models. They propose the transformation of EAST-ADL models into Simulink, which are later model-checked in timed automata semantics. UPPAAL SMC is used as a tool to statistically model-check the models and afterwards, these frameworks are applied on the industrial Brake-by-Wire use case. Another relevant contribution in the formal analysis of EAST-ADL models is made by Kang et al. [8]. In this paper, a model-based approach that links the architectural models with component-aware model checking is proposed, together with a description of the tool that gives support to this called ViTAL. EAST-ADL models are expressed in timed automata, in terms of functional and timing behavior of each function block, together with the interactions between function blocks. The timed automata models can then be formally verified with UPPAAL PORT [9]. Furthermore, it is described how the presented methodology proves the real-time and behavioral requirements of EAST-ADL system model. However, unlike [7] and [8], we focus on EAST-ADL models that have hierarchy, and in which the behavior of each component is modeled in Simulink.

Mallet et al. [10] present an approach where UML MARTE is used to define EAST-ADL timing requirements, whereas Feng et al. [11] propose an approach to transform the EAST-ADL augmented behavioral model in a SPIN model for formal analysis. Qureshi et al. [12] explore a relationship between EAST-ADL and timed automata, by proposing a template mapping scheme for automated transformations between EAST-ADL models and model-checking tool with timed automata notations, as a contribution to the model-driven development of the automotive embedded system.

2.2 Formal Analysis of Simulink models

Filipovikj et al. [13] propose the transformation of Simulink blocks into stochastic timed automata, that can be formally analyzed with UPPAAL SMC. The transformation is automated in a tool called SIMPAAL and it is later applied in a Brake-by-Wire industrial use case system. The CheckMate tool [14] is another tool that transforms Simulink models in a class of hybrid automata. However, many limitations condition its scalability to complex industrial cases. Yang et al. [15] present an effective toolkit, evaluated in different industrial applications, which translates Simulink Stateflow models into UPPAAL timed automata. In addition, Kekatos et al. [16] have presented a methodology for the transformation of simulation models like Simulink in a network of timed automata, described in SX format. They make use of two tools to facilitate the process of constructing the verification model and apply a new hybridization model for blocks not translated correctly. Agrawal [17] describes an algorithm which carries out the translation of Matlab’s Simulink and Stateflow models into equivalent models of hybrid automata using a tool which is based on metamodel graph transformation, to allow for these models to be verified by verification tools without providing concrete means to conduct the formal verification. Tools for translating Simulink models to different model checkers have been carried out in various works like the work by Meenakshi [18], which introduces a tool that acts as a translator of discrete Simulink blocks to the open-source model checker NuSMV.
2.3 Reflection on the state-of-the-art of formal analysis of EAST-ADL and Simulink models

Despite the various and abundant work that has been conducted related to our topic, there is scarcely any work on the formal analysis of EAST-ADL architectural models, extended with behavioral specifications in Simulink. Kang, in [19, 20, 21, 22] extends the work started in [8] on translating EAST-ADL models to UPPAAL by including support for Simulink and validating the set of rules and mapping strategies in automotive case studies.

We present in this thesis work a formal framework that goes beyond the current state-of-the-art in the transformation of both EAST-ADL and Simulink models into a formal model to be verified with UPPAAL timed automata. However, we must highlight that the transformation proposed in this thesis builds upon the works of Marinescu et al. [7] and Filipovikj et al. [13] and requires a new way of expressing the transformation rules that takes into account the formal integration of EAST-ADL with Simulink. To our best knowledge, we present the only approach of formally verifying models with both EAST-ADL and Simulink specifications.
3 Preliminaries

In this section, we provide a basis for understanding the languages and tools used in this thesis work, by briefly explaining their basic concepts, which include the notations of EAST-ADL architectural language, Simulink tool and UPPAAL model checker that are relevant to our work.

3.1 EAST-ADL

EAST-ADL is an architecture, domain-based language which provides early design artefacts of integrated system models, by capturing structure information like functions, timing and triggering. It is assisted by EAST-ADL Association, a non-profit, non-governmental organization which undertakes the maintenance and refinement of the language.

In the automotive domain, the late incorporation of complex functions can lead to critical failures that can be left undiscovered in the final product. Due to such potential problems, in the automotive industry the safety standard ISO26262 has been introduced. In this context, EAST-ADL and similar architectural models being developed for the automotive embedded systems, should be integrated in the automotive development practice to ensure the fulfillment of this kind of safety standards and to provide a level of quality of service. Since automotive products contain many safety-critical subsystems, it is desirable to properly verify their correctness, especially at the early stages of development. For this, in the model-driven development context, it is needed to transform the architectural model to an executable formal model that can be exhaustively verified.

The system features described by EAST-ADL (structures, execution times, deadlines, functions, software and hardware components, variability, requirements etc.) are explicated in four abstraction levels: Vehicle, Analysis, Design and Implementation level, where each abstraction level represents specific system analysis during the development process. A more thorough explanation of EAST-ADL abstraction levels is overseen in Section 3.1.1. EAST-ADL complements AUTOSAR standard by conforming to its representation of software architecture and hardware implementation details. It is in the lowest abstraction level, the Implementation level, where AUTOSAR defines the elements to represent the software architecture and its implementation to the hardware.

EAST-ADL serves as a framework for modeling systems, supporting all activities of the automotive embedded system development. Additionally, its modular attributes allow the support of structural modeling.

3.1.1 EAST-ADL Notations

EAST-ADL meta-model depicts engineering information of the system by covering it in four abstraction levels, as depicted in Figure 1.

Components of the higher levels are decomposed into detailed elements in lower levels, by giving modeling context and ensuring relation between adjoining levels. The Vehicle level describes what the system features are, whereas the other three levels describe how these features function at different detail levels.

The Vehicle level, through TechnicalFeatureModel, describes externally visible features of the vehicle. Those features are defined in Analysis level, including FunctionalAnalysisArchitecture, where abstract functional definitions of the vehicle properties are represented, avoiding implementation details. This is decomposed in Design level by the SoftwareDesignArchitecture and HardwareDesignArchitecture, which represent the software architecture allocated to hardware topology and detailed function designs. The Implementation level contains detailed software architecture and ComponentPrototypes of the hardware, represented by AUTOSAR elements. Nevertheless, all abstraction levels depict a complete embedded system with traceability relations between them.
Furthermore, the system model interacts with two entities: Environment Model and Extensions. The Environment Model captures the environment behavior and represents the relevant elements that interact with the system, in the environment context. It is applied across all abstraction levels and its interaction is supported by the model structure. The Environment Model is needed for verification and validation during all the stages of modeling and developing the system. Extensions cover concerns like requirements, timing, safety/ISO26262, variability, behavior, etc., organized according to the abstraction level.

Functional component modeling is used to model system features at different analysis levels. The main components are functions, which are described by different features like name, description, dependencies, constraints, etc. Constraints act as restrictions used to ensure the correct use of the element. Functions are represented by FunctionType at each level, which abstracts functions types used in the system functional structure modeling. The instantiation of a FunctionType results in FunctionPrototype, which is executed based on the “read-execute-write” semantics. Each FunctionType is associated with FunctionBehavior and FunctionFlowPorts. FunctionBehavior represents the behavior of the component. Behavioral representation tools, like Simulink or UPPAAL PORT timed automata [8], are used commonly. FunctionFlowPorts are ports used in component interaction. They can act as input ports, allowing the reception of the data, or as output ports, allowing the transmission of data. In order for functions to interact with one another, they are connected at their respective ports through connectors. FunctionConnectors are created between two FunctionPrototypes; they ensure intercommunication and indicate that the connected elements exchange signals. FunctionTrigger defines the triggering of the component behavior and can be either event-based or time-based. A graphical representation of these notations is depicted in Figure 2. EAST-ADL provides support to requirements (conditions to be met), verification and validation (through VVCases), timing and generic constraints, etc.

3.1.2 Modeling Tool

There are a set of tools that apply the EAST-ADL language: EATOP [26], PAPYRUS [27], MetaEdit+ [28], VSA [29], MagicDraw [30], SystemWeaver [31]. In this thesis work, we have chosen to work with MetaEdit+ as the modeling tool for EAST-ADL.

MetaEdit+ is a multi-tool environment, developed by MetaCase, that allows the design, edit-
ing, manipulation of architectural models of systems, using EAST-ADL language and complying with its standards. The main drawback of MetaEdit+ is that it is a commercial tool rather than open source.

Its graphical user interface is resourceful, allowing the use of available libraries and icons. The versatile nature of code generation allows a quick and clear model definition. MetaEdit+ has an advantageous graphical representation of the system, lacking in many other architectural model tools. It acts as its own tool rather than a plug-in for other developing environments as it provides a wide variety of modeling and integration capabilities, which include: EAST-ADL XML interchange format, tool specific formats (Simulink, UPPAAL), programmable API that makes possible the integration with other programming environments, availability for Eclipse (as a plug-in) and Visual Studio (as an extension), etc. Its environment allows two versions: workbench (where the developers design and use personalized modeling languages) and modeler (where developers use modeling languages). It offers means to support language concepts (defined in the metamodel), constraints, checking rules, notations (to annotate errors) and generic and timing constraints are easily added.

MetaEdit+ allows working on several projects simultaneously, where data sharing between different projects is possible. Moreover, it supports various modeling languages except EAST-ADL, like UML, GOPRR, CPL, etc.

Despite all this, the EAST-ADL language does not include notations to explicitly specify the behavior of the components. Such a behavior can be defined with external tools, such as Simulink.

### 3.2 Simulink

Simulink [4] is a graphical programming environment, developed as a MATLAB extension, which applies model-driven development and model-based design in the modeling, simulation, verification and code generation of dynamic systems. Its main functionalities are the modeling and simulation of time-dependent system component behavior, testing the system under troublesome and complex circumstances by generating tests automatically, inspecting if requirements have been met, certifying to standards (ISO 26262 [24]), etc. It investigates complex systems by designing graphical models using the block diagram notation and then simulating a time-dependent system represented
by the model. This user-friendly graphical interface visually aids in the designing of the model by allowing block components to be configured accordingly. They are inserted using the drag and drop mouse operation whereas the simulation can be started, paused and stopped interactively from the Simulink Editor “Run simulation button” and results are displayed in real time. Since the results can be seen while the simulation is running, the user can go back and forth through the simulation to analyze the behavior of the system components and identify peculiar situations and errors.

Simulink supports the simulation of a broad range of analog, digital, mixed signal and multi-rate systems, together with linear and nonlinear systems, static and dynamic conditions. Its features allow it to explore the behavior of a variety of systems such as electrical, braking, thermodynamic systems. Due to the shift of the automotive industry towards model-based development [32] after some recent technological advances, Simulink has taken the place as the state-of-practice environment and implementation language in the development process of automotive industry. Moreover, Simulink is compliant with AUTOSAR [25] standards, which are predominantly used in the automotive industry in the development of software architecture for electronic control units. Integration with MATLAB allows an ease of share of information between the two programs.

3.2.1 Simulink notations

Let us take a further look into Simulink notions. A Simulink system is a graphical representation of mathematical equations which link the output to the input(s).

Systems are built using blocks. Blocks consist of some functionality and an arbitrary number of ports. Simulink provides the user with a set of predefined block libraries to browse from. They represent the main behaviors that a Simulink system can have such as:

- Continuous - function blocks such as Derivative and Integrator
- Discrete - function blocks such as Unit Delay
- Math Operators - function blocks such as Sum, Gain and Product
- Signal attributes - function blocks such Data Type Conversion
- String - function blocks to manipulate strings

Nevertheless, the blocks can be custom-made through S-functions block written by the user in C, C++, Fortran, Matlab, and Block Models written in Simulink, which enclose their logic from the user. The library blocks that act as a single unit with single operations are called an atomic block. Figure 3 depicts a simple model designed in Simulink, where the behavior is represented by the atomic blocks of Sum and Product. The model measures the average number of three integer inputs given by a user and displays the result in the screen.

However, a block can consist of a set of atomic blocks and/or other subsystems. A subsystem block is used to structure the model hierarchically. It contains a subset of blocks within the model that have some functionality. During the execution of the system model, all blocks within a subsystem block are executed before moving on to the next block. The subsystem block can be either virtual or non-virtual subsystem. Virtual subsystems have their content flattened to the level of the parent system, whereas non-virtual subsystems have their content computed at once as a single unit (atomic execution). Connections between system blocks are done through an arrow and are used to link the block ports to pass data between them.

3.3 Model Checking

Timing behavior is critical in many real-time systems, such as vehicle airbag deploy system, brake-by-wire automotive technology, etc. The behavior of these systems is time constrained from event to response down to the order of microseconds or less. Thus, the specified time constrains (deadlines) must be guaranteed to be met as safety issues arise. This brings the issue of model checking
the correctness of every modeled system with regard to time constrains. This is done by verifying if the timing behavior of the system complies to the timing requirements. Consequently, systems are modeled as parallel finite state machines with timing properties called timed automata [33].

UPPAAL [34] is a tool designed to verify real-time systems modeled as networks of timed automata.

3.3.1 Timed Automata Notations

A timed automaton (TA) [33, 35, 36] is a finite automata augmented with a set of real-valued clocks. With timed automata, we can model timing properties of real-time systems and verify their behavior.

Syntax The system is modeled as a network of timed automata, composed of different automata. Each automaton is composed of locations and edges, allowing the change from one location to another. Every automaton has an initial location, which acts as the primary location where the execution starts from. A location can be of three types: normal, urgent and committed. A location is made urgent when no time must be passed in it by the system. The same applies to committed locations, with the provision that the following transition must include an outgoing edge of the committed location. Certain constraints can be put to the system locations and edges by comparing the clocks values to time constants, in order to control possible behaviors that the automaton shows. Invariants are conditions put on locations, that act as constraints on the system behavior defining when the system can be in a specific location. When the invariant evaluates to false, the system must leave that location. Guards are conditions put on edges that need to be true in order for the change from one location to another to be possible. Updates on the guard variables can be achieved in edges with simple assignments (a=5) or through a C function (a=f(5)). Additionally, synchronization channels are annotated in complementary labeled edges. Synchronization channels refer to channels labeled as ChannelName! or ChannelName?, where the ChannelName! (the inciter) tries to sync with the ChannelName? channel (the reception). Broadcast channels apply the same principle, but for the fact that one ChannelName! channel triggers many ChannelName? channels.

Semantics The semantics of timed automata is a labeled transition system, where a system
state is defined as the actual location of the TA in the system jointly with the values of all variables of the system. From the initial state of the automaton, two types of transitions can be made: action transition and delay transition [37]. An action transition is undertaken when the current clock satisfies the guard on the corresponding edge, enabling the edge and immediately moving on to the following state. In the case of action transitions, the value of the clocks can be affected by resets on the clocks, if present, otherwise no change happens on them since the transitions are instantaneous. A delay transition delays the automaton for some amount of time by increasing the (synchronous) clocks values with the same amount of time, respecting the current location invariant. Delay transitions are not bounded and they can delay the running of the system up to infinite.

The system (the abstract model of a timed system composed of as states and transitions between states) starts with values of the logical clocks initialized with zero. The clocks keep track of the time passed when timed automata is running. They can be started and checked independently of one another and their values can be increased with the same speed. Each state transition can independently set to zero one or more clocks.

Extensions like stopwatches, real-time tasks or cost functions have been largely studied. There have already been developed a few verification tools to input and analyze timed automata, including the model checkers UPPAAL[38] and Kronos[39]. However, they are still academic research tools.

3.3.2 UPPAAL

UPPAAL[6] is a model-checker tool; it is used to model the dynamic behavior of systems by analyzing and simulating the resulting models, where real-time information about the system can be modeled as well. It was jointly developed by Aalborg University in Denmark and Uppsala University in Sweden. UPPAAL uses the timed-automata notations to model systems. It was chosen as the model-checker tool for this thesis work due to its straightforward graphical user interface, simplicity of use, efficiency and the short learning curve.

The UPPAAL window is made of three main parts: system editor, simulator and verifier. The system editor is used to construct models as a network of timed automata that can communicate with one another using the shared clock and variable values. Global and local declarations are made, together with system declarations which instantiate template objects and system processes. The simulator part of the UPPAAL window enables the simulation of possible executions of the system behavior. The verifier analyzes the system by checking its properties written in TCTL [35].

Figure 4 is an example of a system modeled in UPPAAL. It represents the light switch example, explained in the paper [40] by Behrmann et al. The global declarations of the system depicted in this example are shown in Figure 5.

When the user first presses the button, the light is turned on. If he presses twice quickly, the light becomes brighter, but if he presses the button the second time after a longer time interval defined in the declarations and requirements, the light is turned off. In this example, the time interval has been set to three units of time. We have two timed automata in the model, called User and LightSystem. The User TA has one location, denoted by a vertex, called User and a loop transition that is annotated with the “push!” synchronization, which describes the behavior of the user pressing the light switch. Similarly, the LightSystem TA, which represent the different behaviors of the light bulb, has three locations called: Off (which is also the initial location, marked by double circles), Light and Brighter. The timing behavior of this model is controlled by one clock $x$. The logic behind the system model solution stands in the guards of the edges that connect these three locations. The transition from Off location to Light location occurs when the “push?” synchronization annotation of the edge takes place, meaning when a “push!” happens from the User TA. The automaton leaves the Light location to go to either Off or Brighter location, depending on which of the two edges’ guards is true, more specifically, on the clock value. One edge has been annotated with a guard of a clock value smaller or equal to three and a “push?” synchronization, whereas the other has been annotated with a guard of a clock value bigger than
three, a “push?” synchronization and an update on the clock value which resets it. The system transitions from \textbf{Brighter} to \textbf{Off} location through the edge annotated with a “push?” synchronization and an update of the clock value which resets it.

The semantics of this system, where transitions are labeled with numbers representing the delay amount in case of delay transition, whilst action labels are used in case of action transitions, would be as following:

\begin{align}
(U_{\text{ser}}, \text{Off}, x = 0) & \xrightarrow{\text{delay for } 1\text{-time unit}} (U_{\text{ser}}, \text{Off}, x = 1) \quad (1) \\
(U_{\text{ser}}, \text{Off}, x = 1.2) & \xrightarrow{\text{push}} (U_{\text{ser}}, \text{Light}, x = 1.2) \quad (2) \\
(U_{\text{ser}}, \text{Light}, x = 5.2) & \xrightarrow{\text{push}} (U_{\text{ser}}, \text{Off}, x = 0) \quad (3)
\end{align}

As we can see, the transitions can be either delays (of \(n\) units of time), or action transitions (push). Figure 4 is the simulator view of the UPPAAL window. We can notice that the loop transition (annotated with only a “push!” synchronization) of the User TA and the first edge (annotated with only a “push?” synchronization) of the LightSystem TA are in red color as they represent synchronization channels in the simulation of the system model; their actions happen at the same time.

In the verifier view of the UPPAAL window, we write queries in order to verify if the system meets certain properties. State operators are written as “\(<>\)” and “\([\cdot]\)” \cite{34} to indicate that the query refers to one state and all states respectively. Path quantifiers are notated as “\(A\)” and “\(E\)” to denote that the query should be satisfied by all paths and an individual path respectively. The “leads-to” operator, written as “\(-\to\)”, is used to show that when the notation on its left is satisfied, then eventually the notation on its right will be satisfied. The system is tested in different queries, which express true properties of the modeled system, and the verifier outputs text lines, which indicate whether or not the property is satisfied. Table 1 explains the queries, the properties
to be verified and the status of the different examples of property verifying queries.

<table>
<thead>
<tr>
<th>Query</th>
<th>Property to be verified</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[[] x==3</td>
<td>Verify if in all states of one path, the clock is equal to 3</td>
<td>Not satisfied</td>
</tr>
<tr>
<td>A&lt;&gt; x==5</td>
<td>Verify if of in some state of all possible paths, the clock is equal to 5</td>
<td>Not satisfied</td>
</tr>
<tr>
<td>A[] not deadlock</td>
<td>Verify the lack of deadlock for all possible paths and states</td>
<td>Satisfied</td>
</tr>
<tr>
<td>A[] LightSystem.Light &amp;&amp; LightSystem.Brighter</td>
<td>Verify if for all possible paths and states, the system can reach both the Light and Brighter locations</td>
<td>Not satisfied</td>
</tr>
</tbody>
</table>

Table 1: Light system properties verified by UPPAAL
4 Research Method

In this section, we present the research methodology and development approach adopted in this thesis work, as presented in the work by Tayal [41] and Nunamaker et al. [42]. In order to achieve the goal of this thesis, the research methodology is comprised of a series of structured and systematic steps, which are problem-based and research-oriented. These steps are executed sequentially. If any problem arises during any of the steps, backtracking to the previous step is conducted. Figure 6 is a graphical representation of the multi-step research method of this thesis work, described in Section 4.1.

4.1 Research Method steps

In this subsection we represent a thorough description of the research method steps followed to complete the thesis work.

Formulate the preliminary research goals to be investigated. In this initial step, we formulated and proposed the following research goal: “A framework for the formal verification of EAST-ADL architectural models, extended with behavioral specifications in Simulink”.

Literature review. A critical literature review of the current state-of-the-art in the topic in order to identify gaps in the research literature is performed. We investigate previous attempts of using both EAST-ADL and Simulink features in the modeling, simulation and formal verification of systems. The literature review is performed by splitting it in three pillars:

- Formal analysis of EAST-ADL models
- Formal analysis of Simulink models
- Reflection on the approaches of EAST-ADL models with behavioral specifications in Simulink

The review of literature consists of literature related to the formal verification of the respective models, their transformation to timed automata and tooling attempts to create formal models of EAST-ADL and Simulink models respectively. The research of the literature is performed in multidisciplinary citation databases, and publisher and scientific literature database like Scopus, SpringerLink, IEEE Xplore and Google Scholar. The key words used in the search queries to define and slenderize the search results are “formal analysis” or “formal verification” jointly with “EAST-ADL models” and/or “Simulink models”, combined in successive searches with “UPPAAL” and/or
Refine thesis research goals from the previous step. After getting a better insight on the current state-of-the-art on the topic and making sure our research is not repeating previous works, we keep the initial research goal of proposing a framework, consisting of a tool that will take as input both EAST-ADL and Simulink models and generate automatically the network of timed automata to be verified with UPPAAL, based on the proposed transformation rules of the method. We refine the features of the tool like traceability to explicitly point to the original part it is checking, and reducing the number of hierarchy levels, based on previous attempts.

Propose transformation rules Starting from the initial step, a self education process with focus on the basic notations of EAST-ADL architectural language, Simulink tool and UPPAAL model checker, relevant to our work, takes place. These concepts are represented in Section 3. We start with studying the notations of timed automata, used in the UPPAAL model checker. We look through its syntax and semantics and work with different examples, with the support of our supervisor, in order to get a better grasp of the model checking environment. We then move on to studying the Simulink notations and work with transforming different Simulink models (from simple, to more complex) to UPPAAL models, in order to start thinking on the transformation rules. Firstly, the transformation is done manually for each example, then we design the first drafts of the automatic transformation rules. After concluding this, we move on to studying the EAST-ADL architectural language notations. We start working with different models and then investigate how to transform simple EAST-ADL models into timed automata, in order to come up with transformation rules. Subsequently, we investigate EAST-ADL architectural models with Simulink behavioral specifications and their transformation to timed automata. We do a systematic work on iteratively refining and improving the transformation rules and finalize them in a proposal, as explained more thoroughly in Section 5.

Tool implementation The succeeding step is that of implementing the proposed transformation rules in a tool that will automatically generate the formal model. Since two files would be needed to be read (EAST-ADL and Simulink), we decided to develop the tool by splitting it in six parts:

- Reading and parsing of the EAST-ADL file
- Implementation of the program methods that automate the transformation rules regarding the EAST-ADL features
- Reading and parsing of the Simulink file
- Implementation of the program methods that automate the transformation rules regarding the Simulink features
- Creating UPPAAL templates
- Implementation of the methods linking EAST-ADL and Simulink features

The development is conducted in incremental fashion, with iterations including testing the tool with modeling examples. A detailed explanation of the tool implementation takes place in Section 6.

Case study evaluation After finalizing the tool, we performed a case study on an industrial use case with the Brake-by-Wire (BBW) prototype, in order to evaluate our solution. BBW is the prototype of a braking system equipped with an Anti-lock Break System (ABS) function, which serves as an accurate model of a real industrial system. We use the BBW system in order to observe the use of the tool implementation and show the potential of our solution, by revealing the advantages and limitations. The application of our solution on the Brake-by-Wire use case is presented in Section 7.

Result analysis As a final step, we analyze the results of the case study in terms of method applicability and scalability. The results are displayed in Section 8 whereas the discussion on their
analysis can be found in Section 9. Final remarks are presented in Section 10 and Section 11.

4.2 Threats to Validity

This subsection describes the validity threats which may limit the ability of our thesis work to yield reliable results or their scalability to more complex systems.

1. The method followed in order to achieve the goal of this thesis work does not include proof of the correctness of the transformation. We cannot validate that the formal model is in a bisimulation relation to the original model.

2. The validation is done only on one system, so other problems might appear when the transformation is applied on other systems. Further studies would be needed to apply the transformation on other systems.
5 Transformation Rules

After providing the necessary theoretical background for conducting the thesis work, in this section we present the next step of our research study. Hereby, we propose the transformation rules required to create formal models of systems modeled in EAST-ADL architectural language, extended with behavioral specification in Simulink. Further on this section, we discuss the naming convention, how we cope with hierarchy and other important information, concluding with a set of examples to help the reader understand how the transformation rules produce the formal model.

5.1 Proposed Transformation Rules

Having conducted a thorough investigation of the transformation rules and logic that EAST-ADL (through the MetaEdit+ tool in our work) and Simulink use to model systems, we propose a set of transformation rules that transforms the EAST-ADL and Simulink model to a network of timed automata. We will use a simple example to illustrate the transformation, an example of a model without any kind of hierarchy and where the period and execution time are defined. The transformation shall present how each element - which is relevant to our thesis work - of the EAST-ADL and Simulink model is mapped to an element or a set of elements in the timed automata. The resulting network of TA represent both the interface information encoded in the EAST-ADL architectural model and the behavior model represented by the associated Simulink model. The proposed transformation of the EAST-ADL and Simulink model to a network of TA is as follows.

In order to preserve the informal semantics of the EAST-ADL architectural language, the proposed transformation produces a network of two synchronized TA for each EAST-ADL FunctionPrototype: one TA for the interface, with the elements provided by the EAST-ADL architectural model, and one TA for the behavior (represented by Simulink), as presented in Figure 7. They are named **SystemInterfaceComponent** TA and **SystemBehaviorComponent** TA, respectively. Component stands for the name of the FunctionPrototype, as specified in the EAST-ADL model. This naming convention is chosen in order to reflect traceability and will be applied to all the elements of the formal model which include the word “Component” in the respective annotation.

![SystemInterfaceComponent TA and SystemBehaviorComponent TA](image)

Figure 7: A template of the network of TA produced by the transformation rules for a simple EAST-ADL model, with behavioral specifications in Simulink

The **SystemInterfaceComponent** TA, which is a template for all FunctionPrototypes without hierarchy in EAST-ADL, as depicted in Figure 7a, serves as a representation of the interface of the current FunctionPrototype and is annotated according to it. It has two locations: the initial location, named **Idle**, which stands for the inactive state of the system, and the second location, named **Compute**, which stands for the active state of the system and triggers the **SystemBehaviorComponent** TA.
The two locations are connected by two edges: i) an edge pointing from the Idle location towards the Compute location, which indicates that the system has left the initial location and thus, has started executing, and ii) an edge pointing in the opposite direction, which indicates that the system has left the Compute location, hence the execution is coming to an end. Constraints are applied on both locations (invariants) and on both edges (guards, updates or assignments, and synchronization channels) in order to make sure that the system behaves appropriately. These constraints deal with features of the system like the time properties (period and/or execution time), communication between the TA and connections. We shall explain these elements before coming back to the constraints.

The timing properties of the FunctionPrototype, specified in the EAST-ADL architectural model, are mapped in the formal model through a clock named \( \text{systemClockComponent} \), which is declared globally. This clock will keep track of the execution time and/or period of the current component, if they have been defined. The period and execution time are transformed to integer values.

The synchronization between the \( \text{SystemInterfaceComponent} \) TA and \( \text{SystemBehaviorComponent} \) TA is conducted through the synchronization channels \( \text{startComponent}! \) and \( \text{stopComponent}? \), where the former triggers the behavior of the system, whereas the latter signals the end of the behavior cycle of the whole system and sends the system back to the inactive state.

Each input and output port of the associated FunctionType, specified in the EAST-ADL architectural model, is transformed into a global variable. Their original naming convention of the respective port is preserved in the formal model in order to ensure traceability.

We can now look at the big picture of the \( \text{SystemInterfaceComponent} \) TA and explain how the constraints are applied on all the locations and edges, starting with the first edge which points from the Idle location towards the Compute location. An update is applied on this edge through a simple assignment, \( \text{systemClockComponent} := 0 \), in order to initialize the clock of the current component to zero, thus for it to start measuring the time as the execution has started. Moreover, a synchronization channel is annotated in this connection, \( \text{startComponent}! \), which signals the start of the behavior of the system, whereas the latter signals the end of the behavior cycle of the whole system and sends the system back to the inactive state.

On the Compute location, we apply an invariant, \( \text{systemClockComponent} \leq \text{execTime} \), which allows the system to leave this location only after the execution time has finished. This is because we have now triggered the \( \text{SystemBehaviorInterface} \) TA and the execution time is the time this TA needs to execute the behavior of the component.

On the second edge, which points from the Compute location towards the Idle location, we apply a guard, an assignment and a synchronization as well. The synchronization channel, \( \text{stopComponent}? \) listens to when the execution of the \( \text{SystemBehaviorComponent} \) TA has finished for the triggering to happen in order for the system to move on to this edge. An update happens on this edge and it is a simple assignment which passes the result of the \( \text{SystemBehaviorComponent} \) TA to the output edge of the component, using a temporary variable. The guard on this edge, \( \text{systemClockComponent} = \text{execTime} \), assures that only as much time as the execution time has passed.

After leaving the second edge, the system now goes back to the initial location, where an invariant, \( \text{systemClockComponent} \leq \text{periodTime} \), defines that the system can leave this location only after a full period has been completed.

We will now explain the elements of the second TA. The \( \text{SystemBehaviorComponent} \) TA, as depicted in Figure 7b, represents the behavior of the current FunctionPrototype (specified in Simulink). It has two locations: the initial location, named Off, which represents the state that
the system does not perform anything, and the second location, named On, which represents the state of the system where it starts performing actions, thus, it ‘starts computing’.

These two locations are connected through two edges: i) an edge that points from the Off location towards the On location, which indicates that the execution of the component behavior has been triggered to start, and ii) an edge which points from the On location towards the Off location, which indicates that the execution of the behavior is already running and will soon finish. Constraints are applied on both edges of this TA. On the first edge, a synchronization channel, “startComponent?” is listening to when the triggering of the execution of this TA will happen from the SystemInterfaceComponent TA. If triggered, the system will move on to the On location and from there to the second edge. This edge is composed of an update and a synchronization channel as described below.

As the behavior of the component is specified in Simulink, we transform the Simulink primitive blocks as C functions, where we update a temporary variable, “temp”, by assigning to it the result of the function on the second edge of the SystemBehaviorComponent TA. These functions are declared in the global declarations of the network of TA, together with the number of arguments and the type of the arguments (if any). They are global in order for them to be reused. In order to provide traceability and preserve the Simulink semantics, we name these functions by adjoining the ‘simulinkFunctionName’ to the string ‘Function(arguments)’, where arguments consists of the number and type of arguments (if any). If the component behavior has more than one Simulink block, then each connection between two Simulink blocks is transformed into sequential updates of the respective block functions on this edge, reusing the temporary variable. In that case, the update on the second edge would be “temp=simulinkNameFunction1(), temp=simulinkNameFunction2(temp)”.

After the update has been completed, the synchronization channel, “stopComponent!” signals to the system the end of the execution of the component behavior, which goes back to the initial Off location and leaves the SystemBehaviorComponent TA, going back to SystemInterfaceComponent TA.

In this example we deal with only one type of triggering: time-based triggering. However, there are two possible types of triggering:

1. Time-based triggering, which executes the FunctionPrototype at a specific time, where the timing specifications are made with periods;

2. Event-based triggering, which executes the FunctionPrototype in response to the occurrence of a certain event associated with it, where the timing specifications are made with a set of dedicated boolean variables, that need constant update and reset.

This information is gained from the associated FunctionType in the EAST-ADL architectural model. Moreover, the timing properties include the execution time, which can be either defined or not. In this example, we consider the scenario where the triggering is time-based, with the execution time specified. However, there are four combinations to be taken into consideration:

1. Time-based triggering with specified execution time (as the example we have just presented),

2. Time-based triggering without a specified execution time, where only the period-related constrains remain and the Compute location is annotated as committed.

3. Event-based triggering with specified execution time, where only the execution-time-related constraints remain and we use a boolean variable for each component, named “variableComponent”, by updating and resetting it continuously (making it true or false) in order to trigger the event.

4. Event-based triggering without a specified execution time, where the only time-related constraints are the ones related to the boolean variable.
5.2 Hierarchy

We now discuss how we the proposed transformation rules cope with hierarchy in EAST-ADL, Simulink or both. The presence of hierarchy affects the timed automata, by generating different models depending on the presence or lack of hierarchy. This subsection is divided in four parts which cover the possible combinations: EAST-ADL without hierarchy, EAST-ADL with hierarchy, Simulink without hierarchy, and Simulink with hierarchy.

5.2.1 EAST-ADL without hierarchy

In the case where the system is not hierarchical in EAST-ADL, the SystemInterfaceComponent TA would look like what we have already represented and discussed in Figure 7a. This TA is annotated according to the information that is extracted from the EAST-ADL model. Idle and Compute are the two locations, which represent the state where the system is not active and the state where the system starts executing, respectively. The first edge (pointing to the Compute location) triggers the execution of the SystemBehaviorComponent TA, whereas in the second edge (which points to the Idle location), the Outflow port(s) of the component is (are) assigned the result value of the SystemBehaviorComponent TA, upon its completion. The synchronization between these two TA is done by the synchronization channels startComponent and stopComponent. The information on triggering and execution time is mapped into the elements of this TA, acting as guards, invariants or updates throughout the execution of this TA.

5.2.2 EAST-ADL with hierarchy

We will now expand our transformation rules to the case of hierarchical EAST-ADL models, with behavioral specifications in Simulink. Let us consider as an example the model of a FunctionPrototype with period and execution time defined. The FunctionPrototype is composed of two subsystems in EAST-ADL, which have behavioral specifications in Simulink. The SystemInterfaceComponent TA of the FunctionPrototype will be affected by this hierarchy. It will now be a template as the one represented in Figure 8. This timed automaton will now act as a “coordinator” of the TA representing the interfaces of the EAST-ADL subsystems. For each EAST-ADL subsystems, a network of two TA will be generated, as explained above: SystemInterfaceSubsystem TA and SystemBehaviorSubsystem TA, where Subsystem stands for the name of the FunctionPrototype of the respective subsystem. The execution order of the components is left to right, top to bottom as for the visual inspection of the EAST-ADL components.

![Diagram](image)

Figure 8: A template of the network of SystemInterfaceComponent TA produced by the transformation rules for a hierarchical EAST-ADL model, with behavioral specifications in Simulink

One location will be added to the SystemInterfaceComponent TA in Figure 8 for each EAST-ADL subsystem it is comprised of, for a total of four locations in our example. They will be
connected by four sequential edges. The initial location remains the \textit{Idle} location, and it again represents the inactive state of the system. On the first edge, pointing from \textit{Idle} to the second location, the same constraints are applied: an update which initializes the \texttt{systemClockComponent} to 0 so it can start measuring time, a synchronization channel which triggers the \texttt{SystemInterfaceSubsystem} TA of the first subsystem, and a guard which ensures that the formal model verifies this TA again only after a full period has been completed. The synchronization channel which syncs the \texttt{SystemInterfaceComponent} TA and \texttt{SystemInterfaceSubsystem} TA has a special annotation, where the keyword “Interface” has been added, in order to distinguish this synchronization channel from the one that is used to between the \texttt{SystemInterface} and \texttt{SystemBehavior} TA of the same component. The presence of hierarchy has affected this edge by adding another update, which reflects the mapping of the connection between the model and its subsystem. An update is added which assigns the Inflow port of the system to the Inflow port of the first subsystem.

On the second edge, which connects the second location with the third, two constraints are applied. A synchronization channel which listens if the execution of the \texttt{SystemInterfaceSubsystem1} TA has finished, and an update which assigns the result of the first subsystem to its Outflow port, through a temporary variable declared globally. The third edge, which connects the third location with the fourth location, has both an update and synchronization channel. The update reflects the connection between two EAST-ADL subsystems, which is mapped to the formal model as an assignment of the Output value of the first subsystem to the Input of the second subsystem.

The fourth location and the fourth edge are the same as in the case of no EAST-ADL hierarchy. On the last (fourth) location, an invariant is applied which makes sure for the system to go to the last edge only when as much time as the execution time of the whole system has passed. On the last location, which connects the fourth location to the initial location, a guard, updates and a synchronization channel are applied. The synchronization channel makes sure that the \texttt{SystemInterfaceSubsystem2} TA has finished executing. The guard makes sure that only as much time as the execution time of this component has passed. As for the updates, they are assignments which represent the connection between the result of the behavior of the second subsystem to its Outflow port in the first update, and the connection between the subsystem and the system in the second one, represent by an assignment which passes the value of the Outflow port of the second subsystem to the Outflow port of the system. The initial location has an invariant that makes sure that the system leaves this TA only when a full period has passed.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{network_TA.png}
\caption{(a) The \texttt{SystemInterfaceSubsystem} TA of one of the subsystems (b) The \texttt{SystemBehaviorSubsystem} TA of one of the subsystems}
\end{figure}

Figure 9: A template of the network of TA produced by the transformation rules for one of the subsystems which comprise an EAST-ADL model, with behavioral specifications in Simulink

The \texttt{SystemInterfaceSubsystem}, as depicted in Figure 9a, is comprised of four locations in total - two more than the template used for the system without hierarchy in EAST-ADL. The four edges that connect the locations, are annotated only with synchronization channels. The pair of \texttt{startInterfaceSubsystem1} and \texttt{stopInterfaceSubsystem2} synchronization channel are used to communicate with the \texttt{SystemInterfaceComponent} TA, whereas the \texttt{startSubsystem1} and \texttt{stopSubsystem1} are used
to communicate with the SystemBehaviorSubsystem1 TA, in order to trigger the behavior of the current component and signalize that the behavioral execution cycle has been concluded.

The SystemBehaviorSubsystem TA, depicted in Figure 9b, is similar to the TA representing the behavior of the system with no EAST-ADL hierarchy, shown in Figure 7b. The same pair of TA as presented in Figure 9 would be applied for the second subsystem of which the system is comprised of. When the EAST-ADL model has hierarchy, the number of TA generated is twice the number of FunctionPrototypes present in the system, minus one - as in this case, there would be no SystemBehaviorComponent as the behavior of the system would be represented by the behavior of its subsystems, specifically SystemBehaviorSubsystem TA.

5.2.3 Simulink without hierarchy

In the case where the system has behavioral specification in Simulink without any hierarchy, the SystemInterfaceComponent TA is a template which we have already introduced in Figure 7b. It consists of two locations: Off and On, which represent the states where the system does not perform anything and where it starts executing, respectively. This TA includes two edges connecting the locations in both directions and two synchronization channels - startComponent and stopComponent where Component stands for the name of the respective FunctionPrototype -, which serve to synchronize with the respective SystemInterfaceComponent TA. Simulink function blocks are represented with C functions on the last edge, where a local variable is used to save their outputs.

5.2.4 Simulink with hierarchy

Now we will expand the transformation rules explained above in an EAST-ADL component, with behavioral specifications in Simulink, where the latter is comprised of Simulink subsystems, thus manifesting Simulink model’s hierarchy. This would not affect the SystemInterfaceComponent TA none soever, thus we will not discuss this TA. The SystemBehaviorComponent TA would change from what we have represented in Figure 7b to what is depicted in Figure 10a. The SystemBehaviorComponent TA of the current FunctionPrototype now acts as a “coordinator” of the network of TA representing the Simulink subsystems, as it is used to coordinate the triggering of each one of them and signalize the end of the execution cycle. The number of locations in this TA increased due to this. For each subsystem in Simulink, two locations are added. They represent the active and inactive state of the behavior of the subsystem. They are added after the On location of the SystemBehaviorComponent TA of the current FunctionPrototype. Each Simulink subsystem is represented in the formal model by a TA, as represented in Figure 10b and Figure 10c. This TA is identical to the SystemBehaviorComponent TA of Figure 7b, thus the same naming convention is used, aside from the fact that “Simulink” here represents the name of the Simulink subsystem.

Synchronization channels are added in the edges to trigger the behavior of the respective TA of each Simulink subsystem, where the same naming convention is kept: startSimulink is used to trigger the start of the behavior, whereas stopSimulink is used to signalize the end of a behavioral cycle. ‘Simulink’ is annotated according to the name of the Simulink subsystems. When the execution order of the Simulink subsystems is sequential, the couples of locations of each Simulink subsystem are added one after another in the SystemBehaviorComponent TA of the current FunctionPrototype. However, if the Simulink subsystems were to be executed in parallel, a broadcast channel would be used to trigger the behavior of both subsystems and only two locations would be added. Moreover, in this case, in the outgoing edge of the second added location, boolean variables would be used to signal the ending of the execution of the behavioral cycle of both subsystems.

The timing specifications remain in the SystemInterfaceComponent TA. The execution order of the components is left to right, top to bottom as for the visual inspection of the Simulink blocks. This is achieved by reading the information in the Simulink file that contain the order of execution of the BlockType(s).
As mentioned above, the input and output ports of the EAST-ADL architectural model are declared globally with the same naming convention. The same applies to the inports and outports of the Simulink model, which are matched with the input and output ports of the EAST-ADL architectural model. The connections between the Simulink subsystems (input=output) are read in the Simulink file and transformed into assignments in the edge which signals the end of behavioral execution of the previous subsystem in the connection.

5.3 Additional information

We discuss now other important information related to the transformation rules.

- The semantics of the transformation rules are “Read-Execute-Write”, where “read” stands for the process of parsing the respective EAST-ADL and Simulink files, then the transformation rules are “executed” (automated according to the information read from the files), and in the end, the formal model is ‘written’.

- Number of channels = Number of subsystems + 2.

- Number of variables = 2(input and output) + 2x(number of blocks).

- Number of clocks = number of EAST-ADL components.
5.4 Proposed Transformation Rules applied on examples

Having provided the high-level transformation rules, we now explain how we have applied the transformation rules to produce the formal model on examples which present different scenarios of features like the timing properties, triggering, hierarchy and multiple FunctionType instantiations.

5.4.1 On the example of a system composed of two sequential FunctionPrototypes of a system

We will now show how the proposed transformation rules apply in a system with two FunctionPrototypes, with behavioral specifications in Simulink. The EAST-ADL components are executed sequentially and they are instantiations of two different FunctionTypes. Figure 11 shows us the EAST-ADL model, together with the Simulink specifications. Both FunctionPrototypes are time-triggered, and the execution time has been specified as well. The Simulink model for each FunctionType has been demonstrated in Figure 11. This system has no hierarchy neither in EAST-ADL, nor Simulink.

The first FunctionPrototype, named Average, as depicted in Figure 11, is the instantiation of the Avrg FunctionType. Average is a simple EAST-ADL model, with an execution time of 10 and a period of 100. It has two inputs, one output and the behavioral specification in Simulink is represented by two sequential primitive blocks: sum and gain. The second FunctionPrototype is named Comparison and it is a simple EAST-ADL component, being an instantiation of the Comp FunctionType. It has an execution time of 10 and a period of 150. It is connected to the first FunctionPrototype through a connection between the output of the first component and the only input of this component. OutputE is its only output, whereas the Simulink behavior is represented by a primitive block: compare to constant.

Figure 12 represents the formal model of this example, which is composed of a network of four TA: SystemInterfaceAvrg TA, SystemBehaviorAvrg TA, SystemInterfaceAvrg TA, and SystemBehaviorAvrg TA.

The SystemInterfaceAverage TA, as depicted in Figure 12a, is an instantiation of the SystemInterfaceComponent template, and it is depicted in Figure 12a. It serves as a representation of the interface of the current component and is annotated according to the information read from the EAST-ADL model. It has two locations: Idle and Compute, which stand for the inactive and active state of the system, respectively. The first edge of this TA initializes the clock and triggers the SystemBehaviorAverage TA through the synchronization channel startAvrg!. It then stays in the Compute location for 10 time units, for the execution of SystemBehaviorAverage to be finished. Upon that, in the second edge, the value of the result of the C function is passed on to the output of the component.

The SystemBehaviorAverage TA, as depicted in Figure 12b, represents the behavior of the current component, specified in Simulink. It has two locations, named Off and On. After being triggered by the synchronization channel startAvrg, it assigns the value of the functions to a temporary variable on the second edge. The C functions represent the simple primitive blocks of Simulink that specify the behavior of the component. The order of function calling is according to the order of execution of the Simulink blocks (information extracted from the Simulink file). In this example, the behavioral specifications are comprised of two Simulink blocks, sum and gain, which are represented by the C functions of sumFunction() and gainFunction(), with their appropriate inputs. Since more than one function or update can be executed at one edge, these functions are executed sequentially on the second edge of the TA, similar to the execution order of the blocks in Simulink. Having finished the execution of C functions, the synchronization channel stopAvrg signalized the end of execution of this TA.

The inputs and outputs of the EAST-ADL architectural model are declared as global variables in the formal model - InputA, InputB, and OutputC-, where the same naming convention as in the EAST-ADL model is preserved to ensure traceability. The clock of this component is declared...
globally as well, as `systemClockAverage`. As this component has both the execution time and period specified, their values (integer) are used to check the value of the `systemClockAverage` throughout the execution of the formal model.

The `SystemInterfaceComparison` TA and `SystemBehaviorComparison` TA are modelled the same way as for the Average FunctionPrototype, as portrayed in Figure 12c and Figure 12d. An extra element that is mapped in this example is the connection between the two FunctionPrototypes. This is represented by an assignment. In our example, we have this represented in the first edge of the `SystemInterfaceComparison` TA; right after the clock is initialized, the value of the output of the first FunctionPrototype is assigned to the input of the second (current) FunctionPrototype.

5.4.2 On the example of a system with Simulink hierarchy

The next example represents the same EAST-ADL components as presented previously, with behavioral specifications in Simulink. Figure 13 depicts the EAST-ADL and Simulink information of both components. The same EAST-ADL information is applied in this system as well, however we can notice that a small change has been applied to the system behavior. The behavior of the first FunctionPrototype now displays hierarchy in Simulink. `Avrg1` and `Avrg2` are two Simulink subsystems, which are comprised of primitive function blocks: `Avrg1` is comprised of `Sum` and `Gain`, whereas `Avrg2` is comprised of `Product` and `Squareroot`. The Simulink specification of the second FunctionPrototype is composed of `Max` function. Figure 14 depicts the network of TA generated for the formal model of this example.

Each Simulink subsystem is represented in the formal model by a TA. These TA are identical to the `SystemBehaviorComponent` TA of the EAST-ADL architectural model (explained in the previous subsection), thus the same naming convention is used, aside from the fact that Component here represents the name of the Simulink subsystem.

The `SystemInterfaceAverage` TA, as depicted in Figure 14a, represents the information of the first EAST-ADL component. It beholds no difference from the previous example since the EAST-ADL architectural model does not have any hierarchy. However, the hierarchy in Simulink affects the `SystemBehaviorAverage` TA. As two subsystems are present in the Simulink specifications, this TA now acts as a “coordinator” of all the behavioral components, as it is used to coordinate the triggering of each one of them and signalize the end of the execution cycle. The number of locations in this TA has increased due to this. For each subsystem in Simulink, two locations are added, as in Figure 14b. Synchronization channels are added in the edges to trigger the behavior of the respective TA of each Simulink subsystem, where the same naming convention is kept. The behavior of each Simulink component is depicted in Figure 14c and Figure 14d, respectively.

When the execution order of the Simulink subsystems is sequential, the couples of locations of each Simulink subsystem are added one after another in the `SystemBehaviorAvrg` TA of the current FunctionPrototype. If the Simulink subsystems were to be executed in parallel, a broadcast channel would be used to trigger the behavior of both subsystems and only two locations would be added. However, in this case, we are considering the systems to be executed sequentially for simpler logic flow.

It first triggers the `SystemBehaviorAvrg1` TA (where `Avrg1` is the name of the Simulink subsystem it represents, thus, it is annotated accordingly) through the synchronization channel `startAvrg1!`. The `SystemBehaviorAvrg1` TA is in itself composed of primitive Simulink functions, thus, it only has two locations and is similar to what we have already explained in the previous example. The primitive Simulink blocks of “Sum” and “Gain” are represented through the C functions of `sumFunction()` and `gainFunction()`, which explicitly indicate the Simulink block they represent. Their output is assigned sequentially to a temporary variable, in the order the Simulink blocks are executed. The `SystemBehaviorAvrg` TA, after checking that the first component has finished executing, it triggers the `SystemBehaviorAvrg2` TA, which is the second subsystem. This Simulink subsystem is composed of two Simulink primitive blocks, thus sequential assignments of
the C functions representing the Simulink blocks are conducted on the edge. The same naming
convention applies in this case, despite the Simulink block being primitive or a subsystem. After
the execution of the TA representing the behavior of the model has finished, the system goes back
to the SystemInterfaceAverage TA, where the final value of the functions representing the behavior
is assigned to the respective outputs of this FunctionPrototype.

The timing specifications remain in the SystemInterfaceAverage TA. The execution order of
the components is left to right, top to bottom as for the visual inspection of the Simulink blocks,
according to their order of execution in Simulink.

Figure 15a and Figure 15b are a representation of the two TA generated for the second Func-
tionPrototype, which are identical to what we discussed previously.

As mentioned above, the input and output port(s) of the EAST-ADL architectural model are
declared globally with the same naming convention. The same applies to the Inports and Outports
of the Simulink model, which are matched with the input and output ports of the EAST-ADL
architectural model. The connections between the Simulink subsystems are read in the Simulink
file and transformed into assignments in the edge which signalizes the triggering of the second
subsystem in the connection.

5.4.3 On the example of a system with EAST-ADL hierarchy

We will now expand our transformation rules to the case of hierarchical EAST-ADL models, with
behavioral specifications in Simulink. In Figure 16 we have presented an example of an EAST-ADL
architectural system, which is comprised of two FunctionPrototypes: Average and Comparison.
However, the first FunctionPrototype displays hierarchy in EAST-ADL, as it is comprised of two
parallel FunctionPrototypes: Mean and Geom. As the behavior of the system is not affected by
the hierarchy in EAST-ADL, the TA which represent the behavior, will not be within the scope
of this section. The timing specifications have been made for the Average and Comparison Func-
tionPrototypes, where the triggering is time-based and the execution time is specified.

Each one of the EAST-ADL components, is represented in the formal model with a network of
two TA, as they represent a FunctionPrototype. However, in this example, even though we have
four FunctionPrototypes, we shall have only seven TA for the system in total, as the behavior of
the Average FunctionPrototype is in fact represented by the behavior of its subsystems. The same
naming convention is maintained: SystemInterfaceComponent TA and SystemBehaviorComponent
TA, where Component is the name of the current FunctionPrototype.

Synchronization channels with the same notation as mentioned above are used: startComponent
and stopComponent, but for the fact that Component stands for the EAST-ADL archi-
tectural model component.

The connection between different architectural components is managed by declaring as global
variables all the inputs and outputs of the architectural models, managing the flow of information
by transforming the connections into edge assignments (ex. InputD=OutputC).

SystemInterfaceAverage TA is affected by the EAST-ADL hierarchy and will change its default
template. It will act now as a “coordinator” of the TA representing the interface of the subsystems
it is comprised of, adding one location for each subsystem, for a total of four locations in our exam-
ple, which are connected by four edges. Figure 17 is the graphical representation of this TA. The
initial location is Idle, as in the occasion when there is no hierarchy in EAST-ADL. The first edge
initializes the clock of this FunctionPrototype and triggers the first subsystem. Figure 18 depicts
the network of two TA which is generated for the Geom FunctionPrototype. SystemInterfaceGeom
TA is similar to the previous TA, as it manages the synchronization between the “parent” com-
ponent and its own behavior, by triggering SystemBehaviorGeom TA. The Simulink behavioral
specifications of are simple Simulink models, thus this TA is similar to the one when no Simulink
hierarchy is present.

After the execution of the two TA belonging to the first subsystem has finished, the System\_Interface\_Average TA triggers the second subsystem. The network of TA belonging to Mean is displayed in Figure 19. Both these TA are comprised of the same elements as the previous subsystem. After this execution has finished as well, the System\_Interface\_Average TA goes back to the initial location. Figure 20 is the graphical representation of the Function\_Prototype which is executed after Average, Comparison. Comparison represents a simple EAST-ADL model with no hierarchy neither in EAST-ADL nor in Simulink, thus the same transformation as we have discussed before will be applied to this network of TA.

5.4.4 On the example of a system with event triggering and multiple instantiations of a FunctionType

In this final example we shall explain how two specific cases are handled by our proposed transformation rules: event-based triggering and multiple instantiations of the same FunctionType. Figure 21 shows the EAST-ADL specifications of the system. It is comprised of two FunctionTypes and of three FunctionPrototypes. Average FunctionPrototype is an instantiation of Avrg FunctionType, whereas Comparison1 and Comparison2 are both instantiations of the same FunctionType, Comp. Moreover, we can see that the Comp FunctionType is event-based triggered, with the execution time specified, whereas Avrg is time-based triggered, with the execution time specified as well.

Our transformation rules transform each FunctionPrototype in a network of two TA: System\_Interface\_Component TA and System\_Behavior\_Component TA, where Component is the name of the FunctionPrototype. We extract information from the FunctionTypes and assign it to the respective instantiations. Figure 22 depicts the network of TA generated for the Average FunctionPrototype, Figure 23 for the Comparison1 FunctionPrototype, and Figure 24 for the Comparison2 FunctionPrototype. All these pairs of TA are similar to what we have explained for a simple EAST-ADL component with behavioral specifications in Simulink, without any type of hierarchy.

What is worth noticing in this example is how event-based triggering is handled. Both instantiations of Comp are triggered by an event on the InputD port. Boolean variables are declared globally and are assigned for each event. They are named “variableComponent” where Component is the name of the FunctionPrototype which is event-triggered. On Figure 22, on the second edge of the System\_Interface\_Average TA, after the behavior execution has finished, we update the “variableComparison1” variable to true, as to signalize the triggering of this specific component. On Figure 23, on the first edge of the System\_Interface\_Comparison1 we can notice that this execution will start only if “variableComparison1” is true, that is, if it is triggered by the previous component. After this TA finished its execution, we set the variable back to false. We now update the “variableComparison2” to true in order to trigger by an event the next component. On Figure 24, on the first edge of System\_Interface\_Comparison2 TA, the same guard is applied. It makes sure that the execution of this TA will start only if it is triggered by the previous component. On the last edge, after the execution has finished, we update the variable to false.
Figure 11: First example system, modelled in EAST-ADL with behavioral specification in Simulink
Figure 12: The network of TA produced by the transformation rules for the first example.
Figure 13: EAST-ADL and Simulink specifications for the system depicted in the second example
(a) The SystemInterfaceAverage TA in the second example

(b) The SystemBehaviorAverage TA in the second example

(c) The SystemBehaviorAvg1 TA in the second example

(d) The SystemBehaviorAvg2 TA in the second example

Figure 14: The network of TA produced by the transformation rules for the second example for the first FunctionPrototype
(a) The *SystemInterfaceComparison* TA in the second example

(b) The *SystemBehaviorComparison* TA in the second example

Figure 15: The network of TA produced by the transformation rules for the second example for the second *FunctionPrototype*
Figure 16: EAST-ADL specifications of the third example system

Figure 17: SystemInterfaceAverage TA of the third example
Figure 18: The network of TA produced by the transformation rules for the third example for the first FunctionPrototype.
Figure 19: The network of TA produced by the transformation rules for the third example for the second FunctionPrototype.
(a) The SystemInterfaceComparison TA in the third example

(b) The SystemBehaviorComparison TA in the third example

Figure 20: The network of TA produced by the transformation rules for the third example for the third FunctionPrototype
Figure 21: EAST-ADL specifications of the fourth example system
Figure 22: The network of TA produced by the transformation rules for the fourth example first FunctionPrototype
(a) SystemInterfaceComparison1 TA of the fourth example

(b) SystemBehaviorComparison1 TA of the fourth example

Figure 23: The network of TA produced by the transformation rules for the fourth example second FunctionPrototype
(a) The *SystemInterfaceComparison2* TA in the fourth example

(b) The *SystemBehaviorComparison2* TA in the fourth example

Figure 24: The network of TA produced by the transformation rules for the fourth example for the third *FunctionPrototype*
6 Tool Implementation

In this section, we describe more thoroughly the implementation phase of the tool, which automates the transformation rules. The main goal of our tool is to create an XML file readable by UPPAAL the model checker. This phase is split in three parts, according to the development phases of the tool.

6.1 First tool development

The tool is used to convert EAST-ADL files into XML, UPPAAL readable files is developed in JAVA using the Eclipse IDE. The first development phase started a slightly different way from the final one. The difference was in the implementation flow. Even though we stuck to the same logic (Read-Execute-Write), in the first phase we didn’t use Java libraries which made the developing very difficult due to long and not very precise calculations. Reading the files without a library support would be reading each text line of each file, calculate spaces, special characters, parse important information and build the basic structure of our files. This development strategy would lead to multiple errors and unpredictable bugs, huge number of code lines and a large amount of in-code conditions which would make the new feature almost impossible to add.

6.2 Final tool development

In the final development stage of our tool, we use a “Read-Execute-Write” logic.

The tool offers a simple user interface, as shown in Figure 25, in which the user can browse and choose the EAST-ADL file and then press “Generate XML” button in order to generate an UPPAAL readable file. On its click, the algorithm starts.

Firstly, the number of FunctionPrototypes is counted and then each one of them is initialized as an instance (component) of a class called ComponentClass and added to a list made of ComponentClass-type elements. The ComponentClass is comprised by attributes like: FunctionPrototypeName, pathLocation, functionName, triggerPolicy, isElementary, inFlowPorts, outFlowPorts, inFlowPortId, outFlowPortID, bindings, execTime and periodTime. All these attributes have the getter and setter properties which means that during the algorithm they can be read and written regarding the inputs and the algorithm flow.

Secondly, after the initializing phase, the EAST-ADL file chosen to be converted into an XML file starts to be read. In this phase, information is taken from the EAST-ADL file and immediately written in our initialized ComponentClass-es. We read the properties of the FunctionTypes, like functionName, check and set the TriggeringPolicy (EVENT BASED or TIME BASED), InFlowPorts, OutFlowPorts, InFlowPortID and OutFlowPortID. We continue reading the timing properties of the FunctionTypes. The EAST-ADL files offer two types of timing properties, Execution time and Period time. Thus, different methods are applied for reading these two properties. Based on the respective FunctionTypeName and the path of the Simulink file (found within the EAST-ADL file), we are able to pair each Simulink file to its respective FunctionType. All the FunctionType attributes are saved in the FunctionType class, which is instantiated for every FunctionType that the system contains.

Figure 25: The graphical user interface of the implemented tool
Thirdly, we apply all of the FunctionType attributes to the respective ComponentClass (respective instantiations of the FunctionTypes) and move on to filling the information for the FunctionPrototypes. We check if it is elementary. This is quite a crucial feature to our tool because based on it we decide how to generate the UPPAAL model. When all significant information has been read from the file, we use a method to understand the connection between FunctionPrototypes. It is called bindings, and it creates an ArrayList<String> for each Component in order to pair the Input of one Component with the Output of another, marking the connection between them.

On the next step, the Simulink file, which represents the behavior of the system, is read in the ParseSimulink method. To do such, we use a Java library called Conqat. It facilitates parsing the Simulink (.mdl) files, whose structure is similar to JSON files. Block class is used to save information read from the Simulink file.

After finishing reading the files, we apply methods which execute the proposed transformation rules and immediately after that, we apply those rules by writing the XML file which generates an UPPAAL model. An XML library is used in order to create XML tags easily. Every UPPAAL file is made of these elements: declarations, templates (locations, transitions), system declarations and queries. We start by writing the UPPAAL global declaration. It includes channels, input and output variables, clock, C functions used to perform actions in the behavior templates and booleans used in event-triggered components. After writing the declarations, based on the information in each ComponentClass, we decide which rules to apply depending on the triggering policy and the hierarchy (or lack of) in both EAST-ADL and Simulink. These attributes create different combinations between each other and each one of them follows a different flow in the tool defined by the methods ComplexSimulink, ComplexEastAdl, SimpleSimulink, SimpleEast and in-code conditions. Figure 26 shows the actual logic implementation of the final solution, going into technical details of Java classes and methods.
Figure 26: The architecture of the tool
7 Brake-by-Wire Case Study

In this section, we present you with the case study which is used to validate the proposed transformation rules implemented in the tool, the Brake-by-Wire (BBW) case study. We describe both the EAST-ADL model and the Simulink behavior of the BBW use case.

7.1 A short introduction to the Brake-by-Wire case study

Brake-by-Wire [43] is a newly-emerged and promising industrial system prototype applied in the automotive industry. Its technology implements electronic control units which connect electric motors on each wheel of the vehicle to the brake pedal. This way, when the driver applies pressure on the brake pedal, it is interpreted into electronic instructions which are then transmitted in a real-time network by electronic components to all the wheels' brakes. The system is equipped with sensors to evaluate the driver's actions and external situation, in order to apply the correct braking level on the wheels.

The “by-wire” technology was originally used in military and air-craft manufacturing, previously referred to as “Fly-by-wire”, moving on to vehicle manufacturing nowadays, leading the way to autonomous machines due to its technical merits [44]. Its advantages include installation ease and performance benefits where the brake force of the electrical components is higher than that of conventional systems [45]. Short reaction times, noise reduction effect and reduced space consumption are fair to mention as reasons why traditional mechanical or hydraulic systems have been opted out of in favor of the electronic components ever since.

We have selected this model for the validation of our transformation rules due to its increased popularity and application in the automotive industry, and the complexity of its model which makes it a suitable use case. We have applied minor changes to the model in order to make the system more complex and increase the coverage of our proposed transformation rules by this model, modifications which will be explained throughout this section.

7.2 An insight on the utilized Brake-by-Wire case study

The Brake-by-Wire (BBW) braking system is equipped with different assistance functions, which are represented in our model by FunctionTypes, instantiated in FunctionPrototypes. They include:

- **Brake Pedal Sensor** which calculates the pressure applied on the brake pedal by the driver,
- **Brake Torque Calculator** which takes input from the brake pedal and calculates the torque the driver requests,
- **Sensors and Encoders** which measure the angular velocity of the vehicle in rotations per minute,
- **Brake Controller** which handles the different brake requests from the brake pedal and other sensors,
- **Anti-lock Braking System (ABS)** which prevents wheel locking due to braking. It is applied on all four wheels and takes input from the brake controller to calculate the torque to be applied on them,
- **Actuators** which apply the calculated torque on the brake pedals,
- **Logical Data Models** which electronically transmit the information from hardware to the electronic components.

Figure 27 shows the Brake-by-Wire system prototype at Analysis Level modeled in MetaEdit+. There are a total of 9 (nine) FunctionTypes in this model, as can be seen in Figure 29. However, the Brake-by-Wire model has 24 (twentyfour) FunctionPrototypes, which means that the BBW use case features both scenarios of FunctionType instantiations:

1. a FunctionType which is instantiated in one FunctionPrototypes,
2. a FunctionType which is instantiated in many FunctionPrototypes.

Both these scenarios are considered by proposed transformation rules and the tool implementation. The FunctionTypes and their respective FunctionPrototypes of the BBW model are listed in Table 2.

Figure 28 shows us the timing model of the BBW case study, where the timing properties of all the FunctionTypes are defined. We have made slight modifications to the BBW use case in order for it to include both types of triggering (time-based triggering and event-based triggering) in these three scenarios:

1. time-triggered FunctionType(s), where both the period time and execution time are specified,
2. time-triggered FunctionType(s), where only the period time is specified,
3. event-triggered FunctionType(s), where the execution time is specified.

The PeriodicConstraints in Figure 28 define the period value and on which port the time-triggering is to be applied, whereas the ExecutionConstraints define the ports where the triggering will start and stop, and the execution time. All these scenarios are considered by our proposed transformation rules and have been implemented in the tool. We find the information of the FunctionTrigger for each FunctionType in the behavior graph, displayed in Figure 29, where the trigger policy is specified (either TIME or EVENT). We have included in Table 2 the triggering type for each FunctionType and whether the period and execution time has been specified.

The InflowPort(s), OutflowPorts, the FunctionBehavior -which states the Simulink file with the behavioral specifications-, and the FunctionTrigger, are all specified in Figure 29. Figure 30 shows the behavioral specification of the BBW use case, modeled in Simulink, whereas Figure 31 shows the behavior of each FunctionType.
Figure 27: The Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool
Figure 28: The Timing model of the Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool
Figure 29: The Behavior graph of the Brake-by-Wire use case modelled in EAST-ADL through the MetaEdit+ tool
<table>
<thead>
<tr>
<th>FunctionType name</th>
<th>FunctionPrototype instantiation(s) name</th>
<th>Triggering</th>
<th>Period</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW_BrakePedalSensor</td>
<td>HW_BrakePedalSensor</td>
<td>TIME</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>LDM_BrakePedal</td>
<td>LDM_BrakePedal</td>
<td>EVENT</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>BrakeTorqCalc</td>
<td>BrakeTorqCalc</td>
<td>EVENT</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>HW_Encoder</td>
<td></td>
<td>TIME</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>1. HW_RREncoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. HW_RFEncoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. HW_FREncoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. HW_FLEncoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDM_Sensor</td>
<td></td>
<td>TIME</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>1. LDM_RRSensor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. LDM_RLSensor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. LDM_FRSensor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. LDM_FLSensor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GlobalBrakeController</td>
<td>GlobalBrakeController</td>
<td>TIME</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. ABSatRRWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. ABSatRLWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. ABSatFRWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. ABSatFLWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABSatWheel</td>
<td></td>
<td>TIME</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>1. ABSatRRWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. ABSatRLWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. ABSatFRWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. ABSatFLWheel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDM_Brake</td>
<td></td>
<td>TIME</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>1. LDM_RRBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. LDM_RLBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. LDM_FRBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. LDM_FLBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW_Brake</td>
<td></td>
<td>TIME</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>1. HW_RRBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. HW_RLBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. HW_FRBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. HW_FLBrake</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: List of FunctionTypes and FunctionPrototypes in the BBW model
Figure 30: Simulink model of the BBW system
Figure 31: The Simulink behavior of each FunctionType
8 Results

In this section we present the results of the formal models our tool generates. We start by applying the examples we have presented in the previous sections of our tool and verify if the generated formal models are correct. After that, we present you the validation of our tool through the Brake-by-Wire case study.

8.1 The tool validation with a simple, small-scale system

After finishing the implementation of our tool, which applies the proposed transformation rules presented in Section 5, we start its validation. We start by applying the small-scale examples presented in Section 5.4. We evaluate if the generated formal models by the tool comply with the transformation rules and if all the mapped elements that we have previously covered are present. After that, we verify if in any of the examples, the deadlock state happens. Finally, we verify the timing properties of the systems.

Firstly, we apply the system, described in Section 5.4.1, displayed in Figure 11. This system consists of two FunctionPrototypes, which are instantiations of two FunctionPrototypes. They are both time-triggered, with the execution time specified and their Simulink behavior consists of simple primitive Simulink blocks. No hierarchy is displayed neither in EAST-ADL, nor Simulink. After uploading the EAST-ADL file that represents this system to our tool, the generated formal model consists of the network of TA identical to what we have already presented in Figure 12. We can visually perceive that all the mapped elements are present in the formal model, thus, the transformation rules have been applied correctly in this system.

On the next step, we verify if the generated formal model ever exhibits the “deadlock” state, by applying the query “A[] not deadlock”. Figure 32a displays the applied query, and the result, which is “Property is satisfied”. The query is successful as the system never exhibits the deadlock state.

Then, we verify the Timing properties of the system. In order to do such we implement a monitor. A monitor [46] is implemented in a complete different TA that is added to the generated network of TA and it observes the timing properties of the system if they comply with the specifications through an end to end deadline. We do this step manually, as the automatic creation of a monitor is not within the scope of this work. Figure 32b shows the MonitorEX1 TA which implements a simple monitor in order to evaluate the timing properties of the first example. It consists of 5 locations and four edges, where there are two edges for each component - or in other words, one edge per channel. At this step, we have changed the synchronization channels to broadcast channels, in order for them to be synchronized with this TA as well. We have declared a clock for this TA, named EX1, which is initialized in the first edge. All the edges have been annotated with the broadcast channels, according to the order of execution, and in the last location, a guard is applied. “TA1’==0”, which is translated to a guard that makes sure that the derivative of the clock of the monitor will increase with a rate of zero. In other words, it means that it will stop measuring time. The way we have implemented the monitor creates a deadlock in the system as it never leaves its last condition. As this is not a condition we can verify any longer, we move on to verifying the timing properties of the monitor by applying the query “A[] MonitorEX1.TA1 <= 100”. This query controls if in all states and all paths, the clock of the monitor is never bigger that 100 (the period of the first FunctionPrototype). Figure 32c shows that this timing property is successfully satisfied.

8.2 The tool validation with a small-scale system with Simulink hierarchy

After successfully validating the tool on a simple EAST-ADL system, we now move on to applying an EAST-ADL system, with two FunctionPrototypes, which have behavioral specifications in Simulink with hierarchy. This example is the one that was presented in Section 5.4.2. The system
is presented in Figure 13. The Simulink hierarchy is present in the first FunctionPrototype and it consists of two parallel subsystems, which are comprised of primitive Simulink blocks. We upload the EAST-ADL file to our tool and the results are identical to the network of TA presented in Figure 14 and Figure 15. This way, we can visually confirm that all the elements that our transformation rules take into consideration are present in the formal model.

We now check the “A[] not deadlock” query for this formal model in the UPPAAL verifier. As in the previous example, this property is successfully satisfied, which shows that the system never gets stuck in a deadlock state.

Finally, we check the timing properties of this formal model by implementing a monitor in the same way as for the previous example. The template which implements the monitor is now named MonitorEX2, whereas the clock of this template is named EX2. Figure 33a shows the monitor which is implemented for this example. This TA now has nine locations and eight edges, two edges per component - or in other words one edge per channel (here we consider as component the EAST-ADL FunctionPrototypes and the Simulink subsystems). The same constraints are applied in the first edge and in the last location. Figure 33b shows the same query as in the first example, “A[] MonitorEX2.EX2 <= 100”, applied to the UPPAAL verifier and it is successfully fulfilled.

Further validation will be done regarding the implementation rules of Simulink hierarchy on the BBW use case.

### 8.3 The tool validation with a system with EAST-ADL hierarchy

We now move on to validating if the tool is able to implement correctly the transformation rules regarding the EAST-ADL hierarchy. In order to do such, we upload to our tool the EAST-ADL file that has been previously presented in Section 5.4.3. The EAST-ADL specification are displayed in Figure 16 and we can see that the first FunctionPrototype is composed of two subsystems. The generated formal model contains the identical network of TA that is shown in Figure 17, 18, 19 and 20. This way, we can visually confirm that all the respective elements have been mapped properly by the transformation rules and they are present in the formal model.

As a next step, we verify if in all states, all paths, the deadlock state is achieved. We do that by applying the query “A[] not deadlock”, which is successfully satisfied by UPPAAL.

We then check the timing properties of this system, as shown in Figure 34. The result of this query is “Property may not be satisfied”.

This is the only validation that we will use for the implementation of the EAST-ADL hierarchy transformation rules as our BBW case study does not have any EAST-ADL hierarchy.

### 8.4 The BBW case study validation

We will now present the results of applying the Brake-by-Wire case study to our tool, which implements the proposed transformation rules. We upload the BBW EAST-ADL file to the tool and the results are as following.

Figure 35 shows the network of TA for the HW_BrakePedalSensor FunctionPrototype. This is the instantiation of a FunctionType which is time-triggered and has been annotated accordingly. In the first TA the SystemInterface of this component is shown. The initial location is Idle. The first edge initialized the clock and triggers the behavior of this component through the synchronization channel. The second location, Compute, is committed because the execution time has not been specified so immediately after the behavior cycle finishes executing, we want to leave the location. The second edge assigns the value of the temporary variable to the output of the component. Moreover, since the component that is to be executed after this is event-triggered, we update the respective boolean variable to true. The SystemBehavior TA is annotated accordingly to its
behavior. It has two locations and two edges. The update of the temporary variable happens in the second edge, where the respective Simulink function is executed by taking as input the Inflow port of this component.

Figure 36 shows the network of TA for the LDM_BrakePedal FunctionPrototype. This is the instantiation of a FunctionType which is event-triggered and the execution time has been specified, thus we can see that the SystemInterface TA is annotated with constraints accordingly. On the first edge of this TA we the assignment which represents the connection between the two FunctionPrototypes, thus it represents the passing on the value from the output of the previous FunctionPrototype to the output of the current component. On this edge we see that the guard checks if the variable is true, thus it checks if the event that triggers this component has been executed. On the second edge, we see that the respective boolean variable is set to false, allowing for the component to be triggered again in a second run. Another variable is set to true and that belongs to the next component, which is as well event-triggered.

Figure 37 shows the network of TA for the BrakeTorqCalc FunctionPrototype. This FunctionPrototype has the same features as the LDM_BrakePedal FunctionPrototype, thus is annotated accordingly.

Figure 38 shows the network of TA for the HW_RREncoder FunctionPrototype, which is one of four instantiations of the HW_Encoder FunctionType, which is a time-triggered component with no execution time specified. All four instantiations of this FunctionType generate identical TA.

Figure 39 shows the network of TA for the LDM_RRSensor FunctionPrototype, which is one of four instantiations of the LDM_Sensor FunctionType, which is a time-triggered component with the execution time specified and thus, it has been annotated accordingly. All four instantiations of this FunctionType generate identical TA.

Figure 40 shows the network of TA for the GlobalBrakeController FunctionPrototype, which is a time-triggered component with the execution time specified and thus, it has been annotated accordingly. This component has five inputs and outputs, thus more annotations are stated here. In the TA that represents the interface, we see in the first edge the assignment of the outputs of the previous components to the inputs of our current. The TA that represents the behavior executes the respective C functions sequentially, using temporary variables for each output. Then, after the behavior cycle has finished executing, on the second edge of the interface TA, we assign the values of the temporary variables to the respective outputs.

Figure 41 and 42 shows the network of TA for the ABSatRRWheel FunctionPrototype. This is one of the four instantiations of the ABSatWheel FunctionType, which is a time-triggered component with the execution time specified. Here we can also see that the behavioral specifications show hierarchy in Simulink, thus a total of four TA is generated for this FunctionPrototype. The same transformation rules have been applied to all four instantiations, thus identical TA have been generated for each one of them.

Figure 43 shows the network of TA for the LDM_RRBrake FunctionPrototype, which is one of four instantiations of the LDM_Brake FunctionType, which is a time-triggered component with the execution time specified and thus, it has been annotated accordingly. All four instantiations of this FunctionType generate identical TA.

Figure 44 shows the network of TA for the HW_RRBrake FunctionPrototype, which is one of four instantiations of the HW_Brake FunctionType, which is a time-triggered component with no execution time specified and thus, it has been annotated accordingly. All four instantiations of this FunctionType generate identical TA.

After presenting the network of TA generated by our tool for the BBW case study, we visually confirm that all the elements included by the transformation rules have been mapped successfully.
to the generated formal model. We try to verify if the deadlock state is ever encountered in this system by applying the same query as in the previous examples. Figure 45 shows what happens when this query is checked: a pop-up shows the progress of the verification. However, this process goes on for a long period of time, and only after leaving it overnight to process, the results are what is displayed in the second table “Memory exhausted”.

Another feature of UPPAAL is the model tracing. It is found under the Simulation tab and it shows the states and paths that the model follows when it starts running. In the top of the section, the templates names are shown, together with arrows pointing at the ongoing flow of the running model. The line is interrupted only when a location of a current template is reached. In this case, the locations which are part of the actions are displayed in rectangle shapes. Between the locations, arrows are pointed. They represent how the synchronization channels change the location of the running system and trigger other TA. This procedure continues until a deadlock is found or indefinitely if the system is well-defined and no-deadlock occurs. We have run the formal model of the BBW case study in the simulation. Figure 46 shows the trace of the first Function Prototype of the BBW model. We can see the different locations that the system goes after the triggering of the synchronization channels happens.
(a) The System shows that deadlock is never reached for the first example

(b) The Monitor verifier for the first example

(c) The Monitor query is satisfied for the first example

Figure 32: Pictures that exhibit the verification of 'not deadlock' query in the first model
(a) The Monitor verifier for the second example

Figure 33: Pictures that exhibit the verification of the second example

(b) The Monitor query is satisfied for the second example

Figure 34: Pictures that exhibit the verification of the third example
(a) SystemInterfaceHW_BrakePedalSensor of BBW

(b) SystemBehaviorHW_BrakePedalSensor of BBW

Figure 35: The network of TA generated for HW_BrakePedalSensor FunctionPrototype
Figure 36: The network of TA generated for LDM_BrakePedal FunctionPrototype

(a) SystemInterfaceLDM_BrakePedal of BBW

(b) SystemBehaviorLDM_BrakePedal of BBW
(a) SystemInterfaceBrakeTorqCalc of BBW

(b) SystemBehaviorBrakeTorqCalc of BBW

Figure 37: The network of TA generated for BrakeTorqCalc FunctionPrototype
Figure 38: The network of TA generated for HW_RREncoder FunctionPrototype
(a) SystemInterfaceLDM_RRSSensor of BBW

(b) SystemBehaviorLDM_RRSSensor of BBW

Figure 39: The network of TA generated for LDM_RRSSensor FunctionPrototype
Figure 40: The network of TA generated for GlobalBrakeController FunctionPrototype
Figure 41: The network of TA generated for ABSatRRWheel FunctionPrototype

(a) SystemInterfaceABSatRRWheel of BBW

(b) SystemBehaviorABSatRRWheel of BBW
(a) SystemBehaviorABSatRRWheelSubSystem1 of BBW

Figure 42: The network of TA generated for the Simulink subsystems of ABSatRRWheel FunctionPrototype

(b) SystemBehaviorABSatRRWheelSubSystem2 of BBW
(a) SystemInterfaceLDM_RRBrake of BBW

(b) SystemBehaviorLDM_RRBrake of BBW

Figure 43: The network of TA generated for LDM_RRBrake FunctionPrototype
Figure 44: The network of TA generated for HW_RRBrake FunctionPrototype
(a) The BBW model being verified for the 'not deadlock' query

(b) The BBW memory problem while verifying

Figure 45: Pictures that exhibit the verification of the first example

Figure 46: The simulation of the BBW case study
9 Discussions

We shall now have a look into the results of the validation of our tool and discuss the outcomes.

In all the models used to validate the tool, we could visually check and confirm the presence of all the mapped elements in the generated formal model. This shows that the transformation rules were applied and implemented successfully for every element of both EAST-ADL and Simulink that was within the scope of this thesis work.

“Deadlock” state or the lack of deadlock in all the states and paths was verified for all the examples and it was successfully satisfied in all cases: from small-scale examples, to models with hierarchy in EAST-ADL or Simulink. However, when verifying this property for the Brake-by-Wire use case, we encountered the “Memory exhausted” message from UPPAAL. Let us understand what has happened. UPPAAL creates a tree of all the states and paths of a model, in order to verify it. This tree grows exponentially according to the complexity of the system. In the Brake-by-Wire case study, we generated a formal model that included a network of fifty-six (56) TA. This means that they would generate thousands of states and paths which caused the state space explosion. This was something that was considered from the start of this thesis work. The verification of this specific property needs to be applied in a computer that can handle such situations, with bigger memory space.

We looked into how we could verify the generated formal models. As we could verify the functional and timing properties, we chose the latter as the current tool development lacked the logic implementation of the Simulink functions, which we addressed as a limitation and suggested as Future Work. The timing properties were validated through the implementation of a “monitor”. For a simple system without any hierarchy and for one with Simulink hierarchy, the verified timing properties were successfully verified. However, when verifying it in a system that had hierarchy in EAST-ADL, the result was inconclusive. Further investigation needs to be done on this matter.

However, regarding the proposed transformation rules were quite broad and took into consideration features like: number of instantiations of the FunctionType, triggering, timing constraints, behavior, hierarchy, connections between the ports. Traceability was included in all the proposed transformation rules in order to make it possible to backtrack any issues into the original model.
10 Conclusion

In this thesis work, we propose a framework which allows the formal verification of EAST-ADL architectural models, with behavioral specifications in Simulink. At the beginning, we analyzed the current state-of-art in this topic and investigated previous approaches on this issue, which, to the best of our knowledge, was not tackled before. Afterwards, we proposed a set of transformation rules that take into consideration the EAST-ADL architecture model and the behavioral specifications of the system in Simulink, and generate a network of timed automata. We mapped both the EAST-ADL and Simulink elements that were relevant to our thesis work into elements of the formal model. We then implemented a tool that would automate these transformation rules and generate formal models that can be verified in UPPAAL for each system. We then validated our tool using the Brake-by-wire case study.

As a conclusion for this thesis work, we can say that we came up with some relevant transformation rules that make possible the transformation of EAST-ADL and Simulink to a formal model, with timed automata notations, so the system can be model checked before developed in a real life system. Subsequently, our implemented tool made possible the automatic generation of these formal models. The EAST-ADL hierarchy validation was inconclusive regarding the timing properties. The Brake-by-Wire case study encountered the state space explosion occurrence and thus, the verification needs to be carried on to a more advanced machine. However, all the elements that the transformation rules considered and mapped, were present in the generated formal models.

10.1 Limitations

During our thesis work we faced a few limitations, which mainly affect the tool development and implementation phase.

An initial limitation regards the C functions generated in the UPPAAL formal model. They are comprised of the function name, the return type and the parameters (number and type). However, the logic implementation of the functions is not included in the tool. This limitation affects the functional properties of the formal model as we cannot verify them at the current version of the tool.

Another limitation of the tool is the limited levels of nesting hierarchy. The tool is missing the feature of including systems which have infinite nesting levels in either Simulink or EAST-ADL and it is implementing only one level of hierarchy in each. This limits the systems it can operate normally with. Due to the tool algorithm complexity, sometimes when very complex files are uploaded, the solution can’t be the expected one. It may not be fully correct and the verification might not go as expected.

The last thing that can be marked as a limitation is the limited testing samples. Currently, our tool has been validated by the specific examples that have been explained in the previous sections. Due to this, the tool correctness is limited to the correctness of the provided examples and use cases only.
11 Future Work

According to the results of this thesis work, improvements and further investigation into this issue can be taken into consideration for and conducted as future work.

As the implemented tool is not applicable for all the Simulink blocks, we would suggest to expand this by including more complicated Simulink blocks (taking into consideration that Simulink is a huge application which has an enormous number of functions, blocks, applications and systems).

In our tool implementation, the body of the C functions which represent the Simulink blocks is not stated. Currently, only the name of the Simulink function, the number and type of parameters, and the return type are declared. In further improvements of our work, the actual logic for each Simulink function could be included and their declaration can be automated.

Furthermore, what now that makes the tool limp is the limited nested levels. Theoretically, the tool should work with an unlimited number of nested level. However, due to technical development details, we have chosen to implement hierarchy with one level only. This is an important feature which could be improved in the future, as this would make the automation and the transformation rules more inclusive, leading the way to the tools application into more complex systems.

Additionally, it would be preferred that such improvements were made to the transformation rules and the tool that it might be applicable to generating formal models of complex EAST-ADL and Simulink systems, with more specification details in both EAST-ADL and Simulink.
References


