LANGUAGE SPECIFIC ANALYSIS OF STATE MACHINE MODELS OF REACTIVE SYSTEMS

by

KAROLINA ZUROWSKA

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy in Computer Science

Queen’s University
Kingston, Ontario, Canada
June 2014

Copyright © Karolina Zurowska, 2014
Abstract

Model Driven Development (MDD) is a paradigm introduced to overcome the complexities of modern software development. In MDD we use models as a primary artifact that is being developed, tested and refined, with code being a result of code generation. Analysis and verification of models is an important aspect of MDD paradigm, because they improve understanding of a developed system and enable discovery of faults early in the development. Even though many analysis methods exist (e.g., model checking, proof systems), they are not directly applicable in the context of industrial MDD tools such as IBM Rational Software Architect Real Time Edition (IBM RSA RTE). One of the main reasons for this inapplicability is the difference between modeling languages used in MDD tools (e.g. UML-RT language in IBM RSA RTE) and languages used in existing tools. These differences require an implementation of a transformation from a modeling language to an input language of a tool. UML-RT as well as other industrial MMD models, cannot be easily translated, if the target languages do not directly support key model features. To address this problem we follow a research direction that deviates from the standard approaches and instead of bringing MDD models to analysis tools, the approach brings analysis “closer” to MDD models. We introduce analysis of UML-RT models dedicated to this modeling language. To this end we use a formal internal representation of UML-RT models that preserves the important features of these models, such as hierarchical structures of components, asynchronous communication and action code. This provides us with formalized models using straightforward transformation. In addition, this approach enables the use of MDD-specific abstractions aiming to reduce the size of the state space necessary. To this end we introduce several MDD-specific types of abstractions for: data (using symbolic execution), structure and behavior. The work also includes model checking algorithms, which use the modular nature of UML-RT models. The proposed approach is implemented in a toolset that enables analysis
directly of UML-RT models. We show the results of experiments with UML-RT models developed in-house and obtained from our industrial partner.
Acknowledgments

I would like to thank my supervisor, Dr Juergen Dingel and my PhD supervisory committee: Dr Pat Martin and Dr Ahmed Hassan. This thesis would not be possible without the patience and support of my parents and Paul Geesaman. I also thank my dear friends Sarah Jane Whitaker for company, Ben Hall for jazz, Kathrin Tyryshkin for baking and Wendy Powley for WISC and skiing. For Fridays stress relief I thank my euchre partners and/or opponents, some of which include Dr Janice Glasgow (the better half of the dream team), Dr Jim Cordy, Dr Ernesto Posse, Eric Rapos, Andrew Stevenson, Adrien Muresan, Mark Fisher and all others. I also thank my current and former lab mates Scott Grant, Doug Martin, Matthew Stephan, Gehan Selim, Tuhin Das, Tawhid Bin Waez Md and Elizabeth Antony. Many other people have helped me along this very long journey in and outside School of Computing. Thank you all!
I, Karolina Zurowska, certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with standard referencing practices. Earlier versions of some parts of the work reported in this thesis have previously appeared as [112, 114, 113, 115, 111].
# Table of Contents

Abstract ......................................................... i

Acknowledgments ................................................. iii

Statement of originality ......................................... iv

Table of Contents ................................................ v

List of Tables ..................................................... viii

List of Figures ................................................... x

Chapter 1:

    Introduction ................................................... 1
    1.1 Problem and thesis statement ............................... 2
    1.2 Proposed approach ......................................... 3
    1.3 Contributions .............................................. 5

Chapter 2:

    Background ..................................................... 7
    2.1 Introduction to UML-RT modeling language ............... 7
    2.2 Symbolic execution ......................................... 16
    2.3 CTL model checking ....................................... 19
    2.4 Summary .................................................... 25
Chapter 3:

Related work .................................................. 27
3.1 Analysis and verification of UML behavioral models ................. 27
3.2 Symbolic execution ........................................ 32
3.3 Model checking ............................................ 34
3.4 Summary ................................................... 36

Chapter 4:

Syntax and semantics of CFFSMs .................................. 38
4.1 Structure of CFFSMs ........................................ 39
4.2 Semantics of CFFSMs ....................................... 51
4.3 Transforming UML-RT models to CFFSMs ......................... 71
4.4 Summary ................................................... 78

Chapter 5:

Types of abstractions for CFFSMs .................................. 80
5.1 Symbolic execution ........................................ 83
5.2 Structural abstraction ...................................... 106
5.3 State aggregation .......................................... 113
5.4 Composing abstractions .................................... 119
5.5 Summary ................................................... 121

Chapter 6:

Verification of CFFSMs ............................................ 122
6.1 Logic for properties of CFFSMs ................................ 122
6.2 On-the-fly model checking of CFFSMs .......................... 126
6.3 Checking with abstractions .................................. 149
6.4 Limited exploration ........................................ 154
6.5 Summary ................................................... 161
### Chapter 7: Implementation

- **Design of rule-based engine** .................................................. 164
- **Implementation of execution engines** .................................. 168
- **Implementation of model checking** ...................................... 172
- **Summary** ............................................................................. 175

### Chapter 8: Evaluation

- **Models used in experiments** ................................................ 177
- **PBX model** ......................................................................... 177
- **Symbolic execution** ............................................................ 182
- **Structural abstractions** ....................................................... 185
- **State aggregation** .............................................................. 194
- **Model checking algorithms** ................................................ 198
- **Evaluation conclusions** ...................................................... 209
- **Summary** ............................................................................. 210

### Chapter 9: Conclusions and future work

**Bibliography** ............................................................................ 217

**Appendix A:**

- **Ecore meta-model of CFFSMs** ............................................. 229

**Appendix B:**

- **TUMLAV** ............................................................................ 232
# List of Tables

3.1 Comparison of translational approaches to verification of UML and UML-RT behavioral models. ....................................................... 31

4.1 Specification of modules in an example. ................................. 47
4.2 Informal description of the mapping of the structural elements from UML-RT models to CFFSMs ......................................................... 72
4.3 Mapping of the state machine elements from UML-RT to CFFSMs. ......................... 74
4.4 Symbolic execution rules for code. ............................................ 75

5.1 Comparison of the default rule between concrete and symbolic execution. ........ 89
5.2 Comparison of the match rule between concrete and symbolic execution. ........ 90
5.3 Comparison of the external rule in symbolic and concrete execution. ............... 92
5.4 Comparison of the timeout rules in symbolic and concrete execution. ............ 95
5.5 Comparison of the drop rules in symbolic and concrete execution. ............... 96
5.6 Comparison of rules in concrete an state aggregation execution. ................. 115
5.7 The specification of the execution for composed abstractions. ..................... 120

7.1 Rules in the concrete execution engine. ..................................... 169
7.2 Rules in symbolic execution engine. ....................................... 171
7.3 Rules in structural abstraction engine. ..................................... 172
7.4 Rules in state aggregation engine. ........................................ 173
7.5 Rules in limited exploration engine. ....................................... 174

8.1 Number of states in the symbolic execution of PBX subsystems .................. 185
8.2 Number of states in execution LTS of PBX subsystems model with different structural abstractions used to extract a behavior of a single module. ............................... 193
8.3 Properties from group 1 referring to intersection1 used in experiments with the StreetController models. ................................................................. 201
8.4 Properties in group 2 referring to intersection1 and intersection2 used in experiments with the StreetController models. ................................. 201
8.5 Properties in group 3 referring to intersection1, intersection2 and intersection3 used in experiments with the StreetController models. .................. 202
8.6 Number of symbolic states in execution LTS of PBX subsystems models when checking different formulas (with their informal explanation). .......... 208
List of Figures

1.1 The overview of the analysis process [111]. ................................................. 4

2.1 Structure diagrams of capsules in an example model ................................. 10

2.2 The UML-RT State Machine for TrafficController .................................. 14

2.3 The UML-RT State Machine for CarLights ................................................ 15

2.4 The UML-RT State Machine for WalkLight ............................................... 15

2.5 An example of symbolic execution tree for programs. ............................... 18

2.6 A transition system and checking of satisfaction of $E[b U c]$ ...................... 23

2.7 A transition system and checking of satisfaction of $EGb$ ........................... 24

4.1 Specification of signals and connectors in the traffic controller model. ........... 46

4.2 State machine of Controller. ..................................................................... 51

4.3 State machine of CarLights. ................................................................. 52

4.4 State machine of WalkLights. ............................................................... 52

4.5 An example of a concrete execution state. ................................................. 56

4.6 An illustration of the output effect. ........................................................... 60

4.7 Example of the application of the default rule. ......................................... 65

4.8 Example of the application of the match rule. ........................................... 67

4.9 Example of the application of the external rule. ......................................... 69

4.10 Example of the application of the timeout rule. ....................................... 70

4.11 Example of the application of the drop rule ............................................. 73

4.12 An example of symbolic execution tree for action code. .......................... 77

5.1 An overview of exploration modes. ......................................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Example of a symbolic execution state.</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>An example of application of the match rule.</td>
<td>93</td>
</tr>
<tr>
<td>5.4</td>
<td>An example of the application of the external rule.</td>
<td>94</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of an application of the timeout rule.</td>
<td>97</td>
</tr>
<tr>
<td>5.6</td>
<td>Example of an application of the drop rule.</td>
<td>98</td>
</tr>
<tr>
<td>5.7</td>
<td>An example of application of the subsumption rule.</td>
<td>99</td>
</tr>
<tr>
<td>5.8</td>
<td>An illustration of the inductive proof.</td>
<td>102</td>
</tr>
<tr>
<td>5.9</td>
<td>An example of a structural abstraction.</td>
<td>110</td>
</tr>
<tr>
<td>5.10</td>
<td>An illustration of the inductive proof in case of structural abstraction</td>
<td>112</td>
</tr>
<tr>
<td>5.11</td>
<td>An example of the state aggregation abstraction.</td>
<td>117</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of an execution state.</td>
<td>125</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of LTS used to check formulas.</td>
<td>127</td>
</tr>
<tr>
<td>6.3</td>
<td>An example of execution states used for atomic propositions labeling.</td>
<td>134</td>
</tr>
<tr>
<td>6.4</td>
<td>An example of labeling with a formula $EX(EXp)$.</td>
<td>137</td>
</tr>
<tr>
<td>6.5</td>
<td>An example of labeling with a formula $AFp$.</td>
<td>140</td>
</tr>
<tr>
<td>6.6</td>
<td>An example of labeling with a formula $E[p U q]$.</td>
<td>143</td>
</tr>
<tr>
<td>6.7</td>
<td>An example of labeling with a formula $AF p$ in the presence of cycles.</td>
<td>144</td>
</tr>
<tr>
<td>6.8</td>
<td>An example of a symbolic state.</td>
<td>151</td>
</tr>
<tr>
<td>6.9</td>
<td>An example of limited exploration.</td>
<td>160</td>
</tr>
<tr>
<td>7.1</td>
<td>Design of Toolset for UML-RT Analysis and Verification (TUMLAV)</td>
<td>162</td>
</tr>
<tr>
<td>7.2</td>
<td>A classes used to describe execution engines.</td>
<td>164</td>
</tr>
<tr>
<td>8.1</td>
<td>Structure of capsules $IntersectionController$ and $StreetController$</td>
<td>178</td>
</tr>
<tr>
<td>8.2</td>
<td>$CallController$ structure</td>
<td>180</td>
</tr>
<tr>
<td>8.3</td>
<td>$OAMSubsystem$ structure</td>
<td>181</td>
</tr>
<tr>
<td>8.4</td>
<td>$ProxyManager$ structure</td>
<td>181</td>
</tr>
<tr>
<td>8.5</td>
<td>$DeviceManager$ structure</td>
<td>182</td>
</tr>
<tr>
<td>8.6</td>
<td>Number of states in execution $LTS$ of $Controller$ model with different executions.</td>
<td>183</td>
</tr>
</tbody>
</table>
8.7 Number of states in the symbolic execution of StreetController model with different number of intersections. ................................................................. 184

8.8 Number of states in execution LTS of Controller model with different structural abstractions. ................................................................. 186

8.9 Number of states in symbolic execution LTS of StreetController model with 1 intersection with different structural abstractions. ...................... 188

8.10 Number of states in symbolic execution LTS of StreetController model with 2 intersection parts with different structural abstractions. .............. 189

8.11 Number of states in execution LTS of StreetController model with 3 intersection parts with different structural abstractions. ...................... 190

8.12 Number of states in execution LTS of TrafficLights model with different structural abstractions used to extract a single module. ....................... 191

8.13 Number of states in execution LTS of TrafficLights model with different structural abstractions used to extract a single module. ....................... 192

8.14 Number of states in execution LTS of Controller model with different state aggregation abstractions. ................................................................. 195

8.15 Number of states in execution LTS of StreetController model with 1 intersection with different state aggregation abstractions. ...................... 196

8.16 Number of states in execution LTS of StreetController model with 2 intersections with different state aggregation abstractions. ....................... 197

8.17 Number of states in symbolic execution LTS of Controller used for checking properties. 200

8.18 Number of states explored during checking properties of StreetController with 1 intersection. ................................................................. 203

8.19 Number of states explored during checking properties of StreetController with 2 intersections. ................................................................. 205

8.20 Number of states explored during checking properties of StreetController with 3 intersections. ................................................................. 206

A.1 Ecore meta-model of structural elements in CFFSMs. ................................. 230

A.2 Ecore meta-model of state machine elements in CFFSMs. ............................ 231
B.1 A screenshot from TUMLAV- a symbolic LTS. .................. 233
B.2 A screenshot from TUMLAV- selection of execution parts to abstract. .......... 234
B.3 A screenshot from TUMLAV- selection of locations to be aggregated. ........ 235
Chapter 1

Introduction

Model Driven Development (MDD) is a paradigm introduced to deal with complexities of modern software development. In MDD we use models, which are continuously refined until code can be automatically generated from them [66]. MDD has been applied in various development areas, but has been the most successful in the development of embedded, reactive systems. The development of such systems is supported with industrial-strength MDD tools such as: IBM Rational Software Architect RTE (IBM RSA RTE), IBM Rational Rhapsody [61], Mousetrap from Motorola [12] and Scade Suite from Esterel Technologies [7]. These tools represent a wide spectrum of modeling paradigms, nevertheless in order to support the necessary executability of models [66], they must specify the structure and the behavior of systems. In this research we focus on those modeling languages, in which structure is provided with hierarchically organized modules and the behavior of these modules is given as UML-like state machines [8] with message-based communication. Such characteristics are incorporated in the modeling languages of IBM RSA RTE, IBM Rational Rhapsody and Mousetrap.

One of the promises of MDD is the possibility to analyze models early in the development process. Ideally, analysis enables the detection of faults in models, i.e., verification. Moreover, analysis should also facilitate better understanding of systems. However, analysis of MDD models is challenging, because they have complex dynamic structures and complex behaviors [96]. Additionally, industrial MDD models are large and scalability of techniques provided for them is crucial.

The majority of the approaches in the literature proposed for analysis and verification of models reuse tools such as model checkers partly because they implement sophisticated optimizations, but also because the existing tools, such as SPIN or NuSMV are mature. However, a translation to
an input language of a model checker is required. Such a translation often requires simplifications including flattening of hierarchies, omitting communication details or encoding object-orientation. The goal of our work is to minimize the required translation and to provide more dedicated, language specific analysis [106], similar to SLAM or JPF projects (early efforts by the JPF team to use Promela/Spin were eventually abandoned). This approach reduces the “semantic gap” between the language of a checker and the language of a model, so we bring analysis to MDD models. The direct benefit of our approach is reduced translational effort, more indirect advantages include support for verification methods that are tailored to MDD models. The dedicated approaches to analysis are less often researched, and ours is the first one designed for UML-RT models.

The approach proposed in this work introduces a formal representation, which supports important characteristics with MDD models such as hierarchical components, strong encapsulation, message-based communication and state machines. Preserving these features enables definition of “language specific” abstractions in the context of the formal notation. We propose three types of abstractions: symbolic execution to deal with data, structural abstraction to deal with complex hierarchical structures and state aggregation to simplify certain state machines. Abstractions reduce the size of state space for better understanding, but we also use them to improve model checking algorithms. The improvements come from lazy composition, i.e. exploring only those parts of a model that influence the satisfaction of a property.

1.1 Problem and thesis statement

In this work we deal with the problem of analysis and verification MDD models, which are important techniques to support MDD. Analysis of models is important, because it improves their understanding and consequently improves understanding of software being developed. Analysis also enables verification, that is, the detection of faults early in the development process.

As an example of a MDD language we use UML-RT from IBM RSA RTE [3], because it supports useful modeling features common to several MDD languages. A UML-RT model contains capsules, which are active classes in the model. A capsule has parts, which are instances of other capsules, so the hierarchical composition of components is supported. Capsules communicate using typed ports and the type of a port (called a protocol) gathers messages sent or received through the port. The
behavior of a UML-RT capsule is given with a UML-RT State Machine. UML-RT State Machines contain action code, written in, e.g., Java that updates attributes or sends messages.

The goal of analysis of a UML-RT model (or a model that is similar to it) is to increase its understanding and to allow for verification of properties of the model. In order to achieve this, the model can be executed in IBM RSA RTE, that is, we can follow which messages are being sent or received, in which states of UML-RT State Machines currently are. The execution of a model is useful, but it is limited to one path, i.e., to one set of input values. This gives some insight, but is insufficient to check properties concerning all possible executions, which is necessary to fully understand and to verify models. The first possibility of exhaustive analysis is to reuse existing model checkers, and to translate the model to, e.g., Promela, the input language of the SPIN model checker [92]. Although this enables exhaustive checking of properties, a translation dealing with a sufficiently large subset of UML-RT is complex and difficult to test, and the analysis results are not directly available back in the original model.

The alternative and less followed approach, which we take, is to build a dedicated analysis tool. Our hypothesis is that language specific approach allows for analysis and verification that is not possible with existing universal tools.

Thesis statement

A language specific approach can enhance the analysis of MDD models of embedded system. In this context a language specific approach means an approach that preserves the important aspects of models. By the analysis we understand methods and techniques that can support better understanding and verification. Finally, MDD models of embedded system in our approach are hierarchically organized modular models with behavior given as state machines and action code, which we exemplify with UML-RT models.

1.2 Proposed approach

The approach we propose for UML-RT models is summarized in Figure 1.1. There are three major parts: representation, abstraction and verification; we describe them below.

The first part of the approach is formal representation and consists of translating UML-RT models into the custom formal representation called Communicating Functional Finite State Machines
(CFFSMs). CFFSMs have similar features as UML-RT. A model consists of a set of modules (a counterpart of a capsule). Similarly to UML-RT a module may contain parts and their types are other modules. The communication between parts is realized by signals. CFFSMs support asynchronous communication, hence modules use queues. The behavior of modules is given with state machines. Guards and effects are assigned to transitions in state machines. Effects on transitions include updates of attributes, sending signals, creating or destroying parts. Effects are obtained by symbolic execution of action code in UML-RT. The semantics of CFFSMs is given with a labeled transition system (LTS). An execution state has information about current execution, such as current values of the attributes, current states of state machines, and contents of all queues.

We support the basic features of UML-RT models. However, to maintain simplicity of the approach, we do not support all of the UML-RT language. In the context of the structure we omit features such as inheritance of capsules, plugin capsules, multiple ports and wiring of ports at the run-time. The most important features of UML-RT State Machines that outside of our approach are history pseudostate and synchronous communications. We also do not support some advanced features of action code, such as objects creation, pointers and so on. Finally, our treatment of timers is rudimentary.

In the second part of the approach we introduce abstractions to reduce the size of the state space. We identify abstractions with their execution rules used to generate the execution LTS. We support the following types of abstractions (which may be combined):
1. Symbolic execution. In symbolic execution we use symbols instead of concrete values [65]. To distinguish between branches of execution we include path constraints in an execution LTS. Symbolic execution is very useful to deal with data and to combine execution states that are different only due to the values of input variables. However, symbolic execution is not effective when models have complex structure.

2. Structural abstraction. In this type of abstraction parts of a model are ignored during the execution. The abstracted parts are treated as if they were removed from a model, but signals delivered by them are assumed to be available at all times. Therefore, we say that structural abstraction is an overapproximation. The set of parts to be abstracted is selected by the user.

3. State aggregation. In this type of abstraction states of a state machine are aggregated and act as one state. Therefore, as in the case of the previous abstraction type, this abstraction is also an overapproximation. The user decides which states to aggregate, who may use existing hierarchies of states in the original UML-RT State Machines as guidance.

The last part of our approach includes verification methods, that is the specification of properties of models and model checking algorithms. The properties of models are expressed with an extension of Computation Tree Logic (CTL) [35]. As atomic propositions we use model characteristics such as being in a particular state of state machine, a queue having signals or satisfying conditions on attributes. The checking of CTL formulas is performed on-the-fly and it is combined with the exploration. To this end, the standard labeling algorithm for CTL properties [35] is extended to use the labels from the previous execution steps. Additionally, we optimize the model checking with lazy composition of parts. Initially, parts not mentioned in the checked formula are abstracted. All other parts are executed as usual. If during the execution one of the executed parts requires a trigger generated by some abstracted part then this abstract part is explored to check if it can deliver a signal. Consequently, only some parts of the model are explored, which may potentially limit the number of states in an execution LTS.

1.3 Contributions

The work we present in the thesis includes the following contributions:
1. Formal representation of a subset of the UML-RT language that includes modules, hierarchical composition, communication, dynamic part creation and state machines.


3. Design of verification algorithms for UML-RT like models: on-the fly model checking and model checking based on lazy composition.

4. Implementation of engines that can execute models with abstractions and the implementation of model checking algorithms.
Chapter 2

Background

In this chapter we introduce the necessary background information. First, we describe the most important features of the UML-RT language. After that, we present the basics of symbolic execution and CTL model checking.

2.1 Introduction to UML-RT modeling language

The UML-RT modeling language is a proper UML profile supported by IBM Rational Software Architect Real Time Edition with the latest version 8.1 [3]. Although UML-RT is currently defined in the context of UML, the language has started as the Real-time Object Oriented Modeling (ROOM) language [96]. ROOM is a combination of object oriented features (such as active and passive classes, roles) and statecharts based on the Harel’s original statecharts [59]. Statecharts used in ROOM are simplified, most notably by removing orthogonal regions. The basic building block of ROOM models is a capsule, which can be viewed as an active class. An important aspect of ROOM is encapsulation, which makes capsules highly independent. More specifically, capsules can communicate only by sending or receiving signals. Both the simplification of state machines and the encapsulation are important features that are also contained in UML-RT as shown below.

A model in UML-RT contains capsules and passive classes. Passive classes are the same as classes in any object-oriented language, such as Java, so they will not be discussed here. We will focus on capsules. A capsule is specified with its members such as attributes, a structure diagram and a state machine. Attributes are variables that can be accessed within a capsule (e.g., in its state machine) and they have a user specified default value. Structural aspects of capsules are captured
in a structure diagram, which includes elements that this capsule must have. Behavioral aspects of a capsule are given with a UML State Machine and its usual elements. Both of these diagrams are described below in more detail.

The introduction presented below is by no means an overview of the full UML-RT language capabilities, which can be found elsewhere [96, 3]. We omit some important aspects of UML-RT, such as inheritance and synchronous communication and introduce elements of UML-RT that are relevant to the concepts presented later in this thesis.

### 2.1.1 Structure of UML-RT capsules

A structure diagram of a capsule is a specification of elements included in a capsule and how they are connected. Included elements are parts, ports and connectors.

A capsule may contain parts, which are instances of other capsules identified by their name. A part has a type, which is another capsule. Each part can be fixed, optional or plugin, which determines the way a part is created. If a part is fixed it is simply created with its owner (or parent) capsule, whereas an optional part must be created explicitly at runtime by its owner. Finally, a plugin part is a part created elsewhere in a model and then imported by a capsule – in this way the same part may be shared between capsules. Parts can also have multiplicities greater than 1. In this way they form arrays of instances identified by the index. Although a capsule $C$ can contain parts no containment cycles are allowed, that is, a capsule containing $C$ cannot also be a part of type $C$.

Ports enable communication between a capsule and its parts or its environment. A port can be external or internal, which indicates whether it communicates with the environment (external port, placed on boundary of the capsule) or with the internal parts of a capsule. Internal ports are also used to declare timers. A port can be behavioral or relay. A behavioral port enables receiving and sending signals, so it influences the behavior of the state machine. Relay ports just connect the outside of a capsule with its internal parts. A port can be base or conjugated. This property determines which signals defined by a protocol can be sent and received. Finally, ports declare the multiplicity, which typically is one. However, if the multiplicity of some port is greater than one, the port becomes an array (similarly to parts).

The type of a port is a protocol. A protocol gathers signals that can be sent or received through a
port. *Signals* can be in or out. In case of a base port, out signals are sent, and in signals are received. This is reversed for conjugated ports. Signals may have typed variables (parameters). The type of a signal can be a primitive type (integer, boolean, float) or it can be an object of a passive class declared for a model. The object can be passed through its reference (if communicating capsules share the same memory) or it can be passed by a value, as a new instance.

A *connector* connects two ports. If both ports are behavioral one of them must be base and one of them must be conjugated. The same is necessary if both ports are relay and both of them are declared for internal parts. However, if one of the connected ports is a relay external port, connected ports are both base or conjugated. Connectors declare explicit wiring of ports. In UML-RT it is possible also to declare connections between ports at runtime, during the execution of a model.

**Example 1.** In Figure 2.1 the structure diagrams of capsules in an example of a UML-RT model are presented. The first capsule *TrafficController* has two attributes *carsD* and *walkD* (not shown in the figure). Figure 2.1a shows its structure. It includes two optional parts (optional parts are parts marked with diagonal lines): *cars* and *walk*. It also includes several behavioral ports, one of them is external (*external*) and the others are internal. Two of them connect with internal parts (*carsManager* and *walkManager*) and two of them are timers (*carsTimer* and *walkTimer*). The structure of *CarLights* and *WalkLights* is more straightforward. *CarLights* has one external conjugated port (*manager*) and one timer (*yellowTimer*), whereas *WalkLights* has just one conjugated external port (*manager*).

In this work we present methods to deal with fixed and optional parts, with regular and relay ports and with wired types of connectors. We do not support multiple parts and ports, plugin capsules and runtime connectors. Also we limit the types of variables associated with signals to primitive types.

### 2.1.2 UML-RT State Machines

In this section we present UML-RT State Machines which describe the behavior of each capsule in a model. A UML-RT State Machine includes states, transitions and action code that drives the execution.

A *state* in a UML-RT State Machine represents some distinguished aspect of the execution of a
CHAPTER 2. BACKGROUND

(a) Controller structure

(b) CarLights structure

(c) WalkLights structure

Figure 2.1: Structure diagrams of capsules in an example model
chapter 2. Background

Capsule. States can be composite (hierarchical) including other states inside them. State inclusion specifies the standard ancestry relationship (children, parents and so on). During the execution, a UML-RT State Machine can only be in a non-composite state, because composite states serve as containers. States may include action code in the form of entry and exit actions executed when entering or leaving a state.

In UML-RT we can distinguish several types of pseudostates, which similarly to composite states do not represent an execution state that a UML-RT State Machine can be in:

- initial pseudostates represent the initial state of a given UML-RT State Machine or of a given state.

- entry and exit pseudostates are placed on borders of states and they connect incoming or outgoing transitions,

- choice pseudostates represent branching of transitions based on evaluation of guard code,

- history states represent the last visited state in the containing composite state.

States are connected with transitions. A transition has a state (or a pseudostate) as its source and its target, as well as an optional trigger and two pieces of action code: guard and effects. In UML-RT State Machines transitions cannot cross states boundaries, so very often transition chains are used. A transition chain is a sequence of transition segments connected with exit, entry or branching pseudostates. Such a chain must have a trigger, it starts in a non-composite state and it targets a non-composite state. Some segments of a transition chain might not be modeled directly with a transition but with a group transition. A group transition is a transition that exits from a parent (or ancestor) of a given state and it is a shortcut for having transitions from each of the included states (hence the name, group transition). Some of the transitions are internal, which means that if fired they do not leave a state and no exit code is executed.

The first transition of a transition chain must have a trigger. It is a signal that a capsule can receive on one of its behavioral ports. One of such signals can be also a timeout event received from an internal timer port.

Transitions can have action code associated with them as guards and effects. Action code might be also present as exit or entry actions in states. In both cases the action code is executed when
executing the transition and it can be any arbitrary C++ or Java code. We list here the most important uses of action code relevant to our purposes:

- `port.signal(...).send()`, where `port` is a port name and `signal` is a signal name: asynchronously sending signal `signal` (possibly with some value) through port `port`. This call will use a connected receiving port and put the signal in the queue of the owner of the receiving port. There are several versions of this method, for instance, to deal with multiple ports,

- accessing the input received with a triggering signal (in C++ such input is accessed via pointer `*rtdata`, in Java method `getMsgData()` is invoked),

- accessing an attribute to update or to get its value (for instance, to send it with a signal),

- `timer.informIn(...)`, where `timer` is a timer: setting a timer with some specified delay. This means that after the specified delay a timeout event is generated, which is used to trigger a transition. UML-RT supports soft real time constraints, that is, the timeout event will be generated, but the time it is actually processed might be later than specified. This delay is due to the fact that the currently processed event must finish its execution before we can start processing the next event, i.e., timeout event. Other versions of this call enable generation of multiple timeout events,

- `frame.incarnate(part,...)`, where `part` is a part: incarnating an inner part `part` of a capsule. This method has a number of parameters including the type of a capsule to create, some input values, and an index (in case of multiple parts). A similar method is `frame.import(...)`, which is used for plugin parts. To destroy a part we use `frame.destroy(part)`,

- `return value` is used in guard action code. The returned boolean expression `value` is used as a guard condition. The condition needs to be satisfied to fire a transition.

UML-RT supports run-to-completion semantics with a step of execution being signal handling. This means that after a signal is picked from a queue, all actions that this signal may trigger are treated as an atomic effect and will all be executed before any other signal is handled. The action code is executed in the order imposed by a transition chain. The order of execution of action
code is the following: action code on a transition, exit actions from a state we are leaving, entry actions from a state we are entering, and action code on a next transition. The run-to-completion semantics is affected by thread assignments. In UML-RT we can select on which thread a capsule will run. Each thread has its own queue and run-to-completion is separate for each thread, so code running on different threads might be executed concurrently. By default all capsules in a model are running on the same thread and they share a queue. If capsules are assigned to different threads, the encapsulation of capsules ensures that effects are not interwoven, except for the order of signals sent to the same queue.

Example 2. We will continue the example introduced in Example 1. Figure 2.2 shows an example of the UML-RT State Machine for the TrafficController capsule. This UML-RT State Machine has a number of states and an initial pseudostate. Most of them are non-composite states, except for the Operating state which is composite. The transitions are labeled with triggers and effects, which contain different actions. For instance, consider a transition between the initial pseudostate and Starting. The effect of firing this transition includes incarnating two internal parts of this capsule, i.e., cars and walk. Another type of effect occurs on the transition chain between Both ready and Stop walk. This chain consists of two segments and it enters a composite state. In the action code the received input value is stored in a variable $k$ and then if it is greater than zero it is used to update the attribute carsD. The value of the attribute is used as a parameter of the send signal. If the state Operating had an entry action, it would be also executed. Other actions are also used, for example, between Start cars and Cars the timer carsTimer is set. Figure 2.3 shows the UML-RT State Machine for the CarLights capsule and Figure 2.4 shows the UML-RT state machine for WalkLights. CarLights cycles through Red, Yellow, Green and YellowRed and WalkLights through Walk and NoWalk.

2.1.3 Supported subset of UML-RT

In this section we summarize features that are support by the proposed approach. In the structural aspects we currently support the following features:

- variables (as attributes and associated with signals) of primitive numeric and boolean types,
Figure 2.2: The UML-RT State Machine for TrafficController. The labels on transitions are of the form trigger/effects.
Figure 2.3: The UML-RT State Machine for CarLights. The labels are as in Figure 2.2.

Figure 2.4: The UML-RT State Machine for WalkLight. The labels are as in Figure 2.2.
- fixed and optional parts,
- internal, external, relay ports with multiplicity of 1,
- wired connectors.

In the context of UML-RT State Machines we support the following features:

- states with and without hierarchies,
- transition chains,
- group transitions,
- pseudostates: entry exit points, basic choice points,
- action code present on transitions and in states,
- action code as guards (without side effects).

We also support the following action code features:

- operations on attributes: assignments, conditional statements, loops,
- creating and destroying parts,
- setting timers,
- sending signals (possibly with data) and receiving values from signals.

Our approach can be extended to support excluded features. Some of them need only implementation support. However, some of UML-RT features require more comprehensive changes, because they influence the execution semantics. These features include support for creation and destruction of objects.

2.2 Symbolic execution

In this section we introduce a traditional approach to the symbolic execution as proposed by King [65]. The main idea of symbolic execution is to use symbols instead of actual values of parameters. In this way a program is executed as usual but whenever a value of a parameter is required,
its symbol is used instead. These symbols represent all possible values of given parameters and it might be necessary to restrict them in order to follow a certain branch of execution. To represent such restrictions we use path constraints, that is, constraints that must be satisfied to follow a certain path of execution. The result of symbolic execution is a symbolic execution tree that gathers execution states encountered when executing code.

For illustration of symbolic execution of code, we use the example of the function presented in Listing 2.1. This function has 2 parameters: \( a \) and \( b \). It declares a local variable \( k \), to which a value of \( a \) is assigned. The local variable \( k \) is then used in a condition of an if statement in line 3. Then a value is assigned to \( b \), which is used as a condition in another if statement. The function returns either the value of \( k \) or the sum of \( a \) and \( b \).

Listing 2.1: An example of a function

```plaintext
1  int fun(int a, int b) {
2      int k = a;
3      if (k > 10) {
4          b = k;
5      }
6      if (b > 0){
7          return k;
8      }
9      return a + b;
10  }
```

In Figure 2.5 we have the symbolic execution tree that represents the result of symbolic execution of the function in Listing 2.1. It consist of states describing the current execution snapshot including values of variables and parameters and path constraints. We start with the state \( S_1 \). This state is the initial state and it contains the current values of parameters \( a \) and \( b \). These values are symbolic and represented with \( a_s \) and \( b_s \). The next state, \( S_2 \), includes a snapshot after the execution of the statement in line 2, that is, initialization of a new variable \( k \) to the value of \( a \), which in this execution is \( a_s \). The next statement is conditional with the condition \( k > 10 \). The current value of \( k \) is \( a_s \) and we need to consider both possibilities: that the condition is satisfied and that it is not. To distinguish these cases we add path constraints and we branch the execution. This results in two resulting states: \( S_3' \) and \( S_3'' \). In the first state we assume that the condition holds and we execute the code in line 4, which assigns the current value of \( k \), i.e., \( a_s \) to \( b \). In the state \( S_3'' \) the value of \( b \) is not changed. We need to execute the remaining code for both of those branches. Let us start with \( S_3' \). In this state we want to execute another conditional statement, with the condition \( b \)
> 0, which for the current values is \( a_s > 0 \). We should again consider two cases: one in which the condition is satisfied and one in which it is not. However, the current path constraints include the condition \((a_s > 10)\) which cannot be satisfied along with \( a_s \leq 0 \). This means that this branch is not reachable and it is omitted. The only resulting state is \( S4' \) which has an extra constraint in PC and the return value, since this is the statement executed inside the if branch. In case of state \( S3'' \), both cases must be considered and they result in the states \( S4'' \) and \( S4''' \) with different path constraints and different return values.

The symbolic execution tree in Figure 2.5 includes all possible execution paths of the function \( \text{fun} \). This means that the tree can be used to generate test cases that cover all reachable code. To get such test cases for each leaf in the tree we have to find values of parameters that satisfy the path constraints. In the above case one of the solutions is \((a_s = 11,\ b_s = 0)\), \((a_s = 10,\ b_s = 1)\) and \((a_s = 10,\ b_s = 0)\). In case of more complex constraints we may use existing SAT solvers to find values of parameters that satisfy all path constraints.
2.3 CTL model checking

In this section we present the syntax and semantics of Computation Tree Logic (CTL) formulas and the original model checking algorithm for CTL formulas as presented for instance in Clarke et al. [32, 35].

Let us first introduce a model, which is a transition system.

**Definition 1 (Transition System)**

Transition system $TS$ is a tuple $(\Sigma, \Delta, \Sigma_0)$, where $\Sigma$ is a set of states, $\Delta$ is a transition relation $\Delta \subseteq (\Sigma \times \Sigma)$ and $\Sigma_0 \subseteq \Sigma$ is a set of initial states. We define a path execution $\pi$ to be an infinite sequence $\sigma_0, \sigma_1, \ldots$, where $\forall i > 0 : (\sigma_i, \sigma_{i+1}) \in \Delta$. $\pi[i]$ denotes the $i$th element of the sequence.

We define a function $P$ that assigns atomic propositions to states. Atomic propositions are properties of states we wish to check.

**Definition 2**

Let $TS = (\Sigma, \Delta, \Sigma_0)$ and let $AP$ be a set of atomic propositions. We have $P : \Sigma \rightarrow 2^{AP}$. So function $P$ assigns a subset of atomic propositions to each state in $TS$.

We define Computation Tree Logic (CTL) formulas starting with their syntax.

**Definition 3 (Computation Tree Logic)**

CTL formulas are defined using the following rules:

- if $p$ is an atomic proposition, then $p$ is a formula,

- if $f$ and $g$ are formulas, then $\neg f, f \lor g, f \land g$ are formulas,

- if $f$ and $g$ are formulas, then $AXf, EXf, AFf, EFf, AGf, EGf, E[f U g]$ and $A[f U g]$ are also formulas.

The semantics of CTL formulas is standard for logical connectives $\neg, \lor, \land$. For the temporal connectives, the first part $A$ or $E$ is a path quantifier and it indicates whether all or some paths satisfy the formula. The operator next $X$ indicates that the formula holds for the next state, the operator finally $Ff$ means that the formula $f$ finally holds and $Gf$ that the formula $f$ holds along
all the path (globally). The operator $f U g$ has two parameters and it holds iff $f$ finally holds and up until that state $g$ holds.

**Definition 4** (CTL semantics)

Let $TS = (\Sigma, \Delta, \Sigma_0)$. For a state $\sigma \in \Sigma$, an atomic proposition $p$ and formulas $f$ and $g$ we define the satisfaction as:

- $\sigma \models p$ iff $p \in P(\sigma)$,
- $\sigma \models \neg f$ iff $\sigma \not\models f$,
- $\sigma \models f \lor g$ iff $\sigma \models f$ or $\sigma \models g$,
- $\sigma \models f \land g$ iff $\sigma \models f$ and $\sigma \models g$,
- $\sigma \models AX f$ iff for all states $\sigma'$ such that $(\sigma, \sigma') \in \Delta : \sigma' \models f$,
- $\sigma \models EX f$ iff for some state $\sigma'$ such that $(\sigma, \sigma') \in \Delta : \sigma' \models f$,
- $\sigma \models AF f$ iff for all paths $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\exists i > 0 : \pi[i] \models f$,
- $\sigma \models EF f$ iff for some path $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\exists i > 0 : \pi[i] \models f$,
- $\sigma \models AG f$ iff for all paths $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\forall i > 0 : \pi[i] \models f$,
- $\sigma \models EG f$ iff for some path $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\forall i > 0 : \pi[i] \models f$,
- $\sigma \models A[f U g]$ iff for all paths $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\exists i > 0 : \pi[i] \models g \land \forall j < i : \pi[j] \models f$,
- $\sigma \models E[f U g]$ iff for some paths $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\exists i > 0 : \pi[i] \models g \land \forall j < i : \pi[j] \models f$.

Some of the introduced formulas are redundant, since they can be represented with others.

**Lemma 1.** Every CTL formula can be represented with the following operators only: $\neg$, $\lor$, $EX$, $EU$ and $EG$. We show this by providing the following equalities:

- $AX f = \neg EX \neg f$,
- $AF f = \neg EG \neg f$,
- $EF f = E[true U f]$,
- $\text{AG} f = \neg \text{EF} f$,

- $A[f \ U \ g] = \neg E[\neg g \ U (\neg f \land \neg g)] \land \neg \text{EG} \neg g$

Thanks to the above lemma, we reduce model checking to checking only formulas that contain $\neg, \lor, \text{EX}, \text{EU}, \text{EG}$.

Checking the satisfaction of a CTL formula $f$ in some transition system $TS$ should determine which states in $TS$ satisfy this formula. The algorithm is based on labeling states with formulas $f_{sub}$, which are subformulas of the checked one. The function $\text{label} : \Sigma \rightarrow 2^{f_{sub}}$ maps states in $TS$ to one of the subsets of $f_{sub}$. Initially we have $\text{label} = P$. The labeling is performed bottom-up, so we start with the smallest subformulas and build our way up. We need to consider all six different CTL formulas.

Labeling with $\neg f$ and $f \lor g$ is obvious. We label a state $\sigma$ with $\neg f$ if $f \notin \text{label}(\sigma)$. We label a state with $f \lor g$ if $f \in \text{label}(\sigma)$ or $g \in \text{label}(\sigma)$.

Labeling with $\text{EX} f$ is shown in Algorithm 1. It simply finds all the states that satisfy $f$ and then marks their predecessors with $\text{EX} f$.

Algorithm 1 An outline of an algorithm $\text{checkEX}(f)$.

\begin{verbatim}
T ← {σ | f ∈ label(σ)}
2: for all σ ∈ T do
3: S ← {σ′ | (σ′, σ) ∈ Δ}
4: for all σ′ ∈ S do
5:   label(σ′) ← label(σ′) ∪ \{EX f\}
\end{verbatim}

Labelling with $E[f \ U \ g]$ is more complex and it is shown in Algorithm 2. We use the following expansion equality $E[f \ U \ g] = g \lor (f \land \text{EX}(E[f \ U \ g]))$. First, all the states that satisfy $g$ are gathered in the $T$ set (line 1). We label all of them with $E[f \ U \ g]$. Next in line 7, we find all the predecessors of some state $\sigma$ in $T$ and gather them in the set $S$. If such a state is labeled with $f$ and not yet labeled with $E[f \ U \ g]$ (condition in line 9), we label it (in line 10) and we add it to $T$ set for further consideration (in line 11).

Example 3. Figure 2.6a presents a transition system and the initial labeling, that is, the assignment of atomic propositions. The formula to be checked is $E[b \ U \ c]$. The initial contents of $T$ (after line 1) is shown in Figure 2.6b marked with filled circles. We select the only element in $T$ and we gather its predecessors in $S$. This set contains a single state and it is labeled with $b$, so we label it with
Algorithm 2

An outline of an algorithm checkEU\((f, g)\) for \(TS = (\Sigma, \Delta, \Sigma_0)\).

\[
\begin{align*}
T &\leftarrow \{\sigma \mid g \in \text{label}(\sigma)\} \\
2: & \text{for all } \sigma \in T \text{ do} \\
3: & \quad \text{label}(\sigma) \leftarrow \text{label}(\sigma) \cup \{E[f U g]\} \\
4: & \text{while } T \neq \emptyset \text{ do} \\
5: & \quad \sigma \leftarrow \text{pick an element from } T \\
6: & \quad T \leftarrow T \setminus \{\sigma\} \\
7: & \quad S \leftarrow \{\sigma' \mid (\sigma', \sigma) \in \Delta\} \\
8: & \text{for all } \sigma' \in S \text{ do} \\
9: & \quad \text{if } E[f U g] \notin \text{label}(\sigma') \text{ and } f \in \text{label}(\sigma') \text{ then} \\
10: & \quad \text{label}(\sigma') \leftarrow \text{label}(\sigma') \cup \{E[f U g]\} \\
11: & \quad T \leftarrow T \cup \{\sigma'\} \\
\end{align*}
\]

\(E[b U c]\). This state is added to \(T\). In Figure 2.6c we can see states that are currently labeled with \(E[b U c]\). We repeat this until there are no more states in \(T\). In Figure 2.6d all filled states are the states that satisfy the formula.

Finally, labeling with \(EGf\) is given in Algorithm 3. First, in line 1, we gather states that are labeled with \(f\) in \(T\). Then we have to restrict this set by removing states that do not satisfy \(EGf\). We do so by gathering all states that are not labeled with \(EGf\) (initially \(\Sigma \setminus T\)) and iterating over it. We pick a state \(\sigma\) (line 6) and we go over its predecessors. If a predecessor \(\sigma'\) is labeled with \(EGf\) we make sure that this label is correct, by going over its successors in \(Post\). If we encounter at least one successor state that is labeled with \(EGf\) (line 15), we can assume that the label of \(\sigma'\) is fine. But if none of the post states is labeled with \(EGf\) we knew that \(\sigma'\) cannot be labeled with \(EGf\). Therefore, we remove it from labels in line 17 and we move \(\sigma'\) from \(T\) to \(N\). We follow this until there are no more states to check in \(N\).

Algorithm 3

An outline of an algorithm checkEG\((f)\) for \(TS = (\Sigma, \Delta, \Sigma_0)\).

\[
\begin{align*}
T &\leftarrow \{\sigma \mid f \in \text{label}(\sigma)\} \\
2: & \text{for all } \sigma \in T \text{ do} \\
3: & \quad \text{label}(\sigma) \leftarrow \text{label}(\sigma) \cup \{EGf\} \\
4: & \text{while } N \neq \emptyset \text{ do} \\
5: & \quad \sigma \leftarrow \text{pick an element from } N \\
6: & \quad N \leftarrow N \setminus \{\sigma\} \\
7: & \quad S \leftarrow \{\sigma' \mid (\sigma', \sigma) \in \Delta\} \\
8: & \text{for all } \sigma' \in S \text{ do} \\
9: & \quad \text{if } (\sigma', \sigma) \in \Delta \text{ then} \\
10: & \quad \text{hasPostEG} \leftarrow \text{false} \\
11: & \text{for all } \sigma'' \in Post \text{ do} \\
12: & \quad \text{if } EGf \in \text{label}(\sigma'') \text{ then} \\
13: & \quad \text{hasPostEG} \leftarrow \text{true} \\
14: & \text{if } \neg \text{hasPostEG} \text{ then} \\
15: & \quad \text{label}(\sigma') \leftarrow \text{label}(\sigma') \setminus \{EGf\} \\
16: & \quad T \leftarrow T \setminus \{\sigma'\} \\
17: & \quad N \leftarrow N \cup \{\sigma'\} \\
\end{align*}
\]

Example 4. Figure 2.7a shows a transition system with the initial labeling of states. We wish to
(a) A transition system with some initial labeling.

(b) Initial $T_0$

c) $T_1$

d) $T_2$

Figure 2.6: A transition system and checking of satisfaction of $E[b \ U \ c]$. Filled states indicate ones that are labeled with the formula.
(a) A transition system with some initial labeling.

(b) Initial $T_0$

(c) $T_1$

Figure 2.7: A transition system and checking of satisfaction of $EGb$. Filled states indicate states labeled with the formula.
check the formula $EGb$. First, we label all states that have $b$ in their labels with $EGb$. The result is shown in Figure 2.7b. The set $N$ contains only one state and we start by considering its predecessors. We have three such states: the state itself, the state with atomic propositions \{a, b\} and the state with \{b, c\}. The state itself is not in $T$ and it is simply removed from $N$. The state with \{b, c\} has two successors and one of them is labeled with $EGb$ so nothing changes. The state labeled with \{a, b\} has only one successor and this successor is not labeled with $EGb$. We can therefore execute the statements in lines 17–19. This state is removed from $T$ and added to $N$. The result of this operation is shown in Figure 2.7c. Next, the state labeled \{a, b\} is checked as it is in $N$. It has only one predecessor, which in turn has at least one successor state labeled with $EGb$. No more changes are applied. All the states labeled with $EGb$ in Figure 2.7c are the states that satisfy the formula.

In the model checking algorithm presented in our approach in Chapter 6 we check for $AFf$ formula. Here we present the original algorithm to check for it. In line 1 we gather states that satisfy $f$ in $T$ and we label them all with $AFf$. Next we iterate through the set $T$. We check all predecessors (line 7). If all successors of a given state are labeled with $AFf$, then we can also label it with $AFf$ (line 16). We add such a state to $T$, so we can check the state later.

**Algorithm 4** An outline of an algorithm `checkAF(f)` for $TS = (\Sigma, \Delta, \Sigma_0)$.

\begin{verbatim}
T ← {σ | f ∈ label(σ)}
2: for all σ ∈ T do
   label(σ) ← label(σ) ∪ \{AFf\}
4: while T ≠ ∅ do
   σ ← pick an element from T
6: T ← T \ \{σ\}
8: S ← {σ′ | (σ′, σ) ∈ Δ}
10: for all σ′ ∈ S do
   if AFf /∈ label(σ′) then
      Post = {σ'' | (σ', σ'') ∈ Δ}
      hasNegativePost ← false
12: for all σ'' ∈ Post do
      if AFf /∈ label(σ'') then
         hasNegativePost ← true
14: if ¬hasNegativePost then
      label(σ′) ← label(σ′) ∪ \{AFf\}
16: T ← T ∪ \{σ′\}
\end{verbatim}

2.4 Summary

In this chapter we present the background information. First we introduce the UML-RT modeling language by showing its most important features. We introduce two specification elements: structure diagrams and state machines. We showed the basic elements of each of them and we provide some
basc information about execution of UML-RT models.

In the chapter we also show the basics of two analysis methods: symbolic execution and CTL model checking. We introduced symbolic execution of code with an example. We provided the syntax and semantics of CTL formulas. Additionally we give basic algorithms used to check satisfiability of CTL formulas.
Chapter 3

Related work

In this chapter we look at existing work relevant to our approach. First, we focus on methods developed for verification of UML behavioral models. Next, we present methods, which use techniques mentioned in this thesis. Therefore, we review the relevant work in symbolic execution and how it is used in the context of model verification. We also look at some variants of model checking algorithms.

3.1 Analysis and verification of UML behavioral models

We review methods developed to analyze and verify UML-based behavioral models. The methods presented in this section are divided in two groups. The first group includes translations of models to existing verification tools. We call methods in this group translational. The second group we call language specific and we include there methods developed specifically for a modeling language. These groups are presented below in more detail.

3.1.1 Translational methods

In this section we discuss the translational approaches to verification and analysis of UML-based behavioral models, that is, state machines and interaction diagrams. In these translation approaches some other language or tool is used to perform verification. In Table 3.1 we gather the translational approaches in chronological order and we group based on the modeling language used and the target verification tool.
CHAPTER 3. RELATED WORK

The early works on UML State Machine verification (e.g., [70, 74, 94, 63]) proposed their translation to Promela, which is the input language of the SPIN model checker. Latella et al. [70] emphasize hierarchical states and Extended Hierarchical Automata are used as an intermediate representation of hierarchical state machines. Lilius et al. [74] flatten state machines, and implementation details are presented. Schäfer et al. [94] propose the verification of state machines with the use of UML collaboration diagrams as properties, which are then translated to PROMELA together with state machines. This approach supports the use of collections of state machines as well as classes. Jussila et al. [63] take the subset of UML State Machines models that enables protocol verification is taken into account. This subset includes classes, deployments and supports queues.

Besides SPIN, other model checkers are also used. For instance, a translation to Abstract State Machines exists [38, 100], which enables the use of the dedicated ASM model checker build on SMV. Coloured Petri Nets have also been proposed, e.g., by Lian et al. [73] and Choppy et al. [30]. This enables the use of CPN tools for simulation, but also to check some of the temporal properties [30]. In all of these works, a subset of UML State Machines is supported, covering the basic features, such as events and choice points. The support of UML 2.0 State Machines is considered Lian et al. [73]. Another approach uses UML-B, which is an extension of the formal language Event-B [101]. It has support for collections of state machines as well as classes and a large case study is introduced. More straightforward translation of a single state machine to CSP# is given by Zhang et al. [109]. Finally, there exists a translation of simple state machines to a high level programming language, such as Java [83]. This enables the use of Java Path Finder, which is a model checker for Java code.

An important aspect of systems is time, so efforts to support verification of timing requirements have also been made. The first approaches (e.g. by Bianco et al. [42]) map UML to timing statecharts, which can be then used by the Kronos model checker. A later approach [68] is based on the translation of collections of UML State Machines with timing constraints to UPPAAL, and has been implemented in a tool called HUGO/RT. The tool requires UML Collaborations to specify properties, which are also translated to UPPAAL. Automata based approach has also been used by Schinz et al. [95] and by Ober et al.[85]. This method is comprehensive (implemented with the commercial UML Rhapsody), as it includes not only collections of state machines and classes, but also dynamic features, such as dynamic object creation. The models are analyzed and verified using
the existing IF toolset. The most recent work has been proposed in the MADES project[16]. The approach includes a translation of class diagrams and state machines to TRIO, which is a first-order temporal logic, and which has a model checker associated with it.

UML provides other ways to represent behavior, for example, with UML Collaborations and Interactions diagrams. These diagrams were used to represent properties of models, such as collaborations [94, 67, 68]. A translation of UML Activity Diagrams to SMV is shown by Eshuis et al. [45]. Beato et al. [17] models can be specified using both UML State Machines and Activity diagrams, which are then translated to the input language of SMV. More recently, Lima et al. [76] proposed a translation of UML 2.0 Sequence diagrams to Promela in which temporal properties based on sent and received messages are verified.

In the context of UML there have been efforts to define executable UML [84], which are appropriate for UML 2.0. In these works the emphasis is put on actions and how they can be executed. The support for verification of actions and executable UML has also been proposed. For instance, Graw et al. [55] report on a transformation to cTLA (compositional Temporal Logic of Actions) and the existing TLA proof system is used to verify properties. In a more recent work by Hansen et al. [57] the translation from xUML to an algebraic language mCRL2 is presented. In this work the safety properties are expressed as state machines and model checking is possible with the existing model checker LTSmin [24].

Some of the above works resulted in the development of tools:

- vUML [75] is one of the first translators to PROMELA (an input language of SPIN), which supported basic UML State Machines,

- HUGO and HUGO/RT [78] is a model translator for collections of UML State Machines and UML Collaborations to SPIN, UPPAAL and KIV. It supports UML in version 1.4.,

- OMEGA was a European project [5] to develop tools and methodologies for real-time and embedded systems. It used extended timed automata as the formal language and it supports UML.

- TABU [17] is a translator of both UML State Machines and activity diagrams to the SMV model checker.
Methods applicable directly to the UML-RT language are less common. An early translation to SPIN was reported by Saaltink [92]. This translation includes a translation of basic features of UML-RT models. Leue et al. [72] propose a translation to the AsmL language used in SpecExlorer. This translation focuses on dynamic features of models, such as optional capsules and multiple containment. The most recent and the most comprehensive translation is given by Posse et al. [87]. The target language is Kiltera an extension of classical process algebras. Unfortunately, no analysis tool is reported for the target language.

As can be seen from this brief review, most of the existing approaches are very limited in the support for more advanced features of UML. More specifically, even approaches that deal with collections of state machines do not provide support for structural features such as hierarchy and dynamic changes. The existing model checking tools, such as SPIN and SMV, are also not very well suited for action code.

3.1.2 Language specific analysis methods

In this chapter we will focus on methods that were developed specifically for UML behavioral models without the support of existing model checkers. Such methods are closer to the work proposed in this thesis. However, compared to the translational approach, there have been fewer attempts on building language specific UML checkers.

One of the efforts to develop analysis methods designed for UML started with UML Class Diagrams and OCL constraints and then were extended to Sequence diagrams. One of the tools developed in this area is USE (UML-based Specification Environment) [91, 110, 54]. The main functionality of this tool is the validation, that is, checking validity of OCL constraints present in the model.

In case of UML State Machines the first verification tool was JACK [50]. JACK allows to check temporal properties in ACTL (the action version of CTL) for Extended Hierarchical Automata (also used by Latella et al. [70] when translating to SPIN), which are very similar to hierarchical UML State Machines. The same formalization has been used in the on-the-fly model checking algorithm of µACTL [51]. The limited exploration is applied when checking the existential properties. The approach has been implemented with some further improvements in the UML Model Checker (UMC)
Table 3.1: Comparison of translational approaches to verification of UML and UML-RT behavioral models.
tool [80, 102]. In its last version the tool supports collections of communicating state machines. Their executions are represented with Labeled Transition Systems with minimization and abstraction possibilities. For instance, abstractions might be based on the equality of transition labels. Properties in the UMC tool are expressed with event based branching time temporal logic.

Another dedicated approach for collections of State Machines is the FUJABA project. Giese et al. [49] explore compositional aspects of models based on the parallel composition and synchronous communication between UML State Machines. This approach is implemented in the FUJABA tool suite [25] and uses compositional verification inspired by the popular assume-guarantee paradigm [9]. In this approach the RAVEN model checker is used.

More recently Liu et al. [77] reported on a self-contained tool USMMC (UML State Machines Model Checker) for communicating UML State Machines. The tool is capable of model checking several properties using the capabilities and the theory of the existing PAT model checker, which supports LTL properties, deadlock detection and reachability checking.

### 3.2 Symbolic execution

Symbolic execution was introduced in the 70s as a method to analyze correctness of code. In the first works concerning symbolic execution [65, 58] only the basic features of procedural programming languages are considered, such as assignments, conditional statements and loops. Although symbolic execution started as a verification method, the method has proved itself as a very efficient technique for test case generation. This is possible because symbolic execution results cover all reachable branches of code. The basic rules for symbolic execution have been extended to deal with high level programming languages and with more advanced constructs such as complex data types or function calls.

In case of references and array types Khurshid et al. [64] introduced a technique called Generalized Symbolic Execution. In this technique if a reference is encountered we perform branching and we either use already existing objects or null or a newly created object. Another approach to deal with references is to model the heap explicitly [108]. In this way it is possible to model several references to the same memory object as a single object.

More recently symbolic execution with string support is being considered. Shannon et al. [99] use
automata to represent strings (nodes are characters) and abstract the basic operations on strings, so that they operate on these automata. Another approach for string manipulation is taken in [93], this approach is based on bit-vector operations on the entire strings and introduces a constraint solver specific for strings. Redelinghuys et al. [90] compares the approaches with implementation in Java Path Finder and it seems both approaches have similar capabilities.

One of the directions in which symbolic execution of programs has been extended, is the composition of symbolic execution results in order to efficiently deal with external function calls. The underlying idea is to treat such external modules (functions, methods) as independent entities, analyze them separately and use only the results whenever the function or method is called. These abstract results of analysis are usually in the form of logical expressions of pre and post conditions [58, 52, 10, 104].

Another way to overcome the complexities of the high level programing languages is to combine symbolic and concrete execution of code. In so called concolic testing [53, 97] concrete values are used to direct the symbolic execution to different branches of code. The main idea behind it is to generate random inputs and execute code with it. During the execution the path constraints that need to be satisfied to follow the path are gathered (as in symbolic execution). The next test case is obtained by negating one of the gathered constraints, thus directing the execution to some other branch. This approach has been extended to other systems, such as testing mobile apps [11].

Over the years several symbolic execution tools were developed [28, 29]:

- Symbolic JPF [86] – JPF is a model checker for Java code developed at NASA. JPF is a dedicated virtual machine for Java byte code and its symbolic extension provides support for symbolic execution. Symbolic JPF implements the Generalized Symbolic Execution approach. The tool requires code instrumentation, to introduce symbolic branching options.

- CUTE and jCUTE [98] – are tools for concolic testing C and Java programs. The tool is an extension of DART to support multithreading,

- Symstra [108] – is a framework for the generation of objects to unit test classes,

- Klee [26] and EXE [27] – are symbolic execution engines for C code by the same authors. Both can use mixed symbolic and concrete execution. EXE focuses on system level code and enables
efficient generation of inputs. It deals with memory level modeling and is efficient in finding inputs that lead to a crash in a system.

Although symbolic execution of programs is a successful approach [88, 29], symbolic execution of state based models is less popular. Rapin et al. [89] introduce a method to symbolically execute an Input-Output Symbolic Transition System and then to check LTL properties on symbolic execution tree. The method is formal, but it is restricted to very basic models with inputs driving transitions between states. The predecessors of UML State Machines, i.e., Statecharts are considered by Thums et. al. [103] and Statecharts with timing (Modecharts) by Lee et al. [71]. In the first work symbolic execution is used to generate LTL formulas representing statecharts, in the second to generate test cases. Balser et al. [15] have extended the correctness proofs to an early version of UML State Machines. Balasubramanian et al. [14, 13] address the problem of heterogeneity of state based models. The proposed tool, Polyglot, uses JPF and symbolic execution as a common framework to analyze such models.

3.3 Model checking

In this section we introduce model checking techniques similar to the ones used in the thesis, that is, on-the-fly, symbolic and abstraction based approaches. The model checking problem is to decide whether $M \models \phi$, where $M$ is a model of a system and $\phi$ is a property being checked.

On-the-fly approaches are very popular in the context of LTL (Linear Temporal Logic) model checking, that is, if formula $\phi$ is in LTL. The semantics of LTL is defined over sets of execution paths in $M$, which can be interpreted as words and languages over automata. The first automaton $A_M$ is for a system and the second one $A_{\neg \phi}$ for the negation of the property to check, which is usually a Büchi automaton [48, 41, 46]. The model checking problem is reduced to checking whether the product of these two $A_M \parallel A_{\neg \phi}$ is non-empty [35]. The early on-the-fly algorithms enabled dynamic construction of the product $A_M \parallel A_{\neg \phi}$ [39]. Later works reduced the complexity by making the generation of $A_{\neg \phi}$ on demand [56]. Most of the methods use the nested depth first search. Geldenhuys et al. [47] present an algorithm that avoids nesting of the depth first search and is more efficient in detection of cycles [47].

The on-the-fly versions of CTL model checking based on labeling [32] are less common. The most
A popular approach is localized CTL model checking [23, 105]. In this approach the checking is directed to the parts of the model that are necessary to check. This is achieved with appropriate backtracking procedures, that is, procedures that can update the labels in the states that are incoming for a given state. Another approach is to reuse automata based theories for CTL [21].

One of the most successful optimization of the CTL model checking is symbolic model checking [81]. This is based on the observation that determining the satisfaction of CTL formulas can be treated as operations on sets of states. For instance, if we check for satisfaction of $\text{AF}p$ we can start with a set of states that satisfy $p$, and then add to this set another set of states that satisfy $\text{AX}p$, and then perform the same operation on this new set. If we cannot add any new states to a resulting set we will have a set of states satisfying $\text{AF}p$. This operation is simply finding the least fixed point of some operator on sets of states. In order to make operations on sets easier than enumeration, efficient representations of sets are necessary. Such a representation is a boolean formula that characterizes states in a model and transitions between them. The boolean formulas can be represented with Ordered Binary Decision Diagrams (OBDDs) and the application of CTL formulas is defined as straightforward transformations on OBDDs. Therefore, the algorithm for symbolic model checking with OBDDs, requires that $\mathcal{M}$ is represented as $\text{OBDD}_M$. Then the formula $\phi$ is traversed according to its subformulas, and transformations are performed $\text{OBDD}_M' = \text{check}(\text{OBDD}_M)$. The transformation $\text{check}$ depends on the formula, and may be a fixed point operation, so may require several iterations. Once the last subformula, i.e., formula $\phi$ itself, is checked the resulting $\text{OBDD}_M$ will represent states satisfying the formula. Some of the more recent works support symbolic model checking based only on satisfaction of the boolean formulas and they use modern SAT-solvers [82].

On-the-fly version of symbolic model checking [18, 20] has been proposed for formulas $\text{AG}p$. The algorithm checks after each step of transition generation whether the negation of $p$ has been detected. As soon as this happens, the algorithm can stop and return a state that has invalidated the formula. In this way it is possible to avoid generation of the entire transition relation.

One of the techniques to reduce the complexity of verification is based on the use of abstractions [79]. In [34] it is shown that for certain CTL properties (only universal path quantifiers) it is possible to check them on the abstracted model $\mathcal{M}_a$ as long as there is a simulation relation between $\mathcal{M}$ and $\mathcal{M}_a$. This approach is similar to abstract interpretation [40], in which the semantics
of certain operations is replaced and simpler state spaces are generated. The abstract interpretation, since it is usually simpler, enables more effective program verification. In this way a state space of an original program “collapses” to the smaller and abstract one which is easier to verify. There were also other numerical abstraction methods proposed for model checking [43, 62].

There are several popular model checkers:

- SPIN [19] is one of the most mature tools, which is based on the automata approach. Its input language is called Promela,

- SMV [36] and nuSMV [31] are based on symbolic model checking and OBDDs,

- JPF [107] is a model checker for Java programs,

- BLAST [22] is a model checker for C programs used for verification of software drivers,

- UPPAAL [69] is a model checker for timed systems.

3.4 Summary

In this chapter we outline the most important works in the areas relevant to our research. First, we show approaches to verification of UML-based behavioral models. Next, we review the current state of methods based on symbolic execution. Finally, we introduce some of the techniques used with model checking.

Verification of UML-based models is important in the context of MDD and there have been a lot of efforts in this research area. These efforts can be divided into translational and language specific approaches. The first ones reuse the existing methods and model checkers (such as SPIN, SMV, UPPAAL). The most important challenge in these approaches is the necessity to implement translations. Usually, the translations cannot handle some of the features of a modeling language such as UML-RT. The second approach is less common, since it requires building dedicated tools. In our research we follow this second approach and we provide language specific analysis and verification. Our approach is different than the already proposed ones, because we focus on the existing UML-RT language. We focus on features that were not dealt with in the previous works such as hierarchical composition of components and more advanced features of the action language.
Symbolic execution has been an active research direction since the 70s. The original symbolic execution has been extended in what it can support. Current versions of symbolic execution engines can deal with advanced features such as function calls and reference types. Current methods can also use symbolic execution to support, not only basic correctness of programs but also test case generation, based on branch coverage criteria. Symbolic execution is a method extensively used to analyze code, it is less commonly used to deal with behavioral state-based with models. In our research we focus on symbolic execution of models, but unlike existing works we symbolically execute collection of state machines. We also provide a technique for using symbolic execution to represent action code in UML-RT models.

Finally, model checking is a very popular method to check models exhaustively. We present some of the optimizations possible in model checking such as: on-the-fly algorithms, symbolic model checking and abstractions. This is a very rich research area with many works. Our work is different from them, mostly because we define model checking of UML-RT models. Moreover, as will be shown later, our model checking algorithm is based on step-by-step exploration.
Chapter 4

Syntax and semantics of Communicating Functional Finite State Machines

In this chapter we will introduce a formal representation of UML-RT models. The representation is called Communicating Functional Finite State Machines (CFFSMs). CFFSMs and the notion of functions have been inspired by XMachines [60]. We designed CFFSMs with the following principles:

1. The structure of CFFSMs should be as similar to that of UML-RT models as possible, that is, the following structural features should be supported: capsules, attributes, parts, connectors, events with operations and timers. CFFSMs will also support the following elements of UML-RT State Machines: states (locations), transition chains, guards, effects, triggers and timeouts.

2. CFFSMs will support asynchronous communication.

3. Action code should be represented in a simplified way using conditional cases called functions.

UML-RT has no formal semantics, and the semantics we consider in this work is based on the informal description of the language in the documentation of the IBM RSA RTE tool. In case of ambiguities we execute models in the tool. The semantics of CFFSMs follows the semantics of the UML-RT language as extracted in the above way. However, because CFFSMs do not include all the features of UML-RT, we had to make some simplifications. The most notable simplifications are:

- more restricted action language: The action language in UML-RT can be C++ or Java. In the current work we support only some of the features of those languages. In this way our version of action language is close also to ALF [1] – the language designed specifically for action code.
- simplified communication methods: UML-RT supports synchronous and asynchronous communication. In this work we decided to support only the asynchronous mode. This mode is more common in UML-RT models and we can also encode synchronous communication if necessary.

- removing the timing constraints: Although CFFSMs support timers, the analysis and verification methods do not deal with timing. After a timer is set, its timeout event is available right away without any consideration of real timing constraints.

**Notation:**

In order to describe the syntax and semantics of CFFSMs we will use the notation inspired by object oriented programming languages, with the dot notation to access elements (e.g., \texttt{part.type}, \texttt{ω.l}). In order to describe elements of the syntax will use their full names and we will use plural names to denote sets (e.g., \texttt{parts}). In order to describe semantics we will use Greek letters and one letter abbreviations. Capital letters will be used to denote sets (e.g., \texttt{Σ}, \texttt{Q}) and small ones to describe elements of these sets (e.g., \texttt{σ}, \texttt{q}). Finally, the helper functions or operations on elements will use words with parameters in parentheses (e.g., \texttt{getState(s, σ)}).

### 4.1 Structure of CFFSMs

The components of a CFFSMs model are possibly nested, asynchronously communicating modules with state machines used to represent their behavior. Modules use variables and we start this section with the description of variables and their domains. Next, we introduce the overall structure of models and the structure of state machines.

#### 4.1.1 Domains and variables

We use \textit{domains}, which are first-order structures with constants, variables, relations and operations defined over universes of elements. In these structures \textit{terms} and \textit{formulas} along with their interpretations are defined in the standard way.

**Definition 5** (Domain)

A domain \(D\) is a structure \((U, S, I, V)\) with:
- \( \mathcal{U} \) is a universe, i.e., a set of elements in the domain (e.g., integers). An element of a universe is also referred as a value.

- \( \mathcal{S} \) is a signature \((\mathcal{S}_o, \mathcal{S}_r, \text{arity})\), i.e., a set of operation symbols \(\mathcal{S}_o\) and relation symbols \(\mathcal{S}_r\) along with their arity \(\text{arity} \) (e.g., \(\text{arity}(\text{`+'}) = 2\), \(\text{arity}(\text{`<'}) = 2\), \(\text{arity}(\text{`4'}) = 0\)),

- \( \mathcal{V} \) is a set of variables (operations with arity 1) with \(\mathcal{V} \subseteq \mathcal{S}_o\),

- \( \mathcal{C} \) is a set of constants (operations with arity 0) with \(\mathcal{C} \subseteq \mathcal{S}_o\),

- \( \mathcal{I} \) is an interpretation of the signature \(\mathcal{S}\), which assigns a function (a relation) to each operation (relation) symbol in \(\mathcal{S}_o \) (e.g., operation symbol \(\text{`+'}\) operation is interpreted as a sum of its parameters, relation symbol \(\text{`<'}\) relation is interpreted as less-than) We denote elements of a given domain with the superscript, e.g., for a domain \(\mathcal{D}\) its constants are denoted with \(\mathcal{C}_\mathcal{D}\).

For some domain \(\mathcal{D}\) terms \((\mathcal{T}_\mathcal{D})\) and formulas \((\mathcal{F}_\mathcal{D})\) along with their interpretations are defined in the standard way. A term is built from operations \(\mathcal{S}_o\) and variables, whereas a formula is built from relations \(\mathcal{S}_r\) and variables and terms. The interpretation of a term yields an element from the universe and interpretation of a formula yields a true or false value. The set of variables in a term \(t\) (a formula \(f\)) is given by \(\text{vars}(t) \) \((\text{vars}(f))\). For a given set of variables \(V \subseteq \mathcal{V}\) we denote a set of all terms (formulas) by \(\mathcal{F}[V] \) \((\mathcal{T}[V])\) that contain only variables from \(V\). In the sequel we use \(v : \mathcal{D}\) to denote a given variable \(v\) and its domain, as well as \(\text{dom}(v)\) to indicate the domain of the variable \(v\).

We define formulas over multiple domains as compositions of regular formulas using standard logical connectives.

**Definition 6** (Multi-domain formulas)

For given domains \(\mathcal{D}_1, \ldots, \mathcal{D}_n\) we define formulas \(\mathcal{F}\) with \(f \in \mathcal{F}\) and \(f = f_1 \circ f_2 \circ \ldots \circ f_m\), where \(f_1, \ldots, f_m\) are formulas over multiple domain or over one of the domains and \(\circ \in \{\neg, \lor, \land, \Rightarrow\}\). The interpretation of a formula \(f\) is based on the standard interpretation of logical connectives.

In the sequel we use the notion of valuation of variables, that is, a function that assigns to each variable a term from its domain. A special case of a valuation is a concrete valuation that assigns a constant to each variable in it.
Definition 7 (Valuations of variables)
Let $V$ be a set of variables and let $dom(v)$ be the domain of each variable $v \in V$. A valuation of variables in $V$ is a function $val : V \rightarrow \mathcal{T}$, where $\mathcal{T} = \bigcup_{v \in V} \mathcal{T}_{dom(v)}$ is a set of all possible terms. We require that for each $v \in V : val(v) \in \mathcal{T}_{dom(v)}$, so each variable is mapped to a term from its respective domain. A set of all possible valuations of $V$ is denoted as $\mathcal{V}$, so $val \in \mathcal{V}$.

Definition 8 (Concrete valuation of variables)
Let $V$ be a set of variables and let $dom(v)$ be the domain of a variable $v \in V$. A concrete valuation is a valuation $val \in \mathcal{V}$ such that $\forall v \in V : val(v) \in \mathcal{C}_{dom(v)}$, where $\mathcal{C}_{dom(v)}$ is a set of constants in the domain $dom(v)$. The set of all concrete valuations for the set of variables $V$ is denoted with $\mathcal{V}_c$.

Finally, we define a replacement operation on terms or formulas that replaces variables with their respective terms.

Definition 9 (Replacing variables in formulas and terms)
Let $V$ be a set of variables with a valuation $val \in \mathcal{V}$ and let $f \in F[V]$ be a formula (or $t \in T[V]$). We define $f \parallel val$ (or $t \parallel val$) to be a formula (or a term) in which all variables in $v \in V$ are replaced by their values $val(v)$.

4.1.2 Structure of a model
In CFFSMSs a model consists of modules, queues and a queues map used to assign queues to modules. A designated, top-level part in the model serves as the root of the hierarchy for all other modules contained in the model.

Definition 10 (Model)
A model is a tuple $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$, where:

- modules is a set of modules (see Definition 11),

- queues is a set of queues,

- queuesMap is a function that assigns a queue to each module, i.e., $\text{queuesMap} : \text{modules} \rightarrow \text{queues}$,

- topPart is a distinguished top level part,
- `topModule` is the type of the top level part, i.e., `topModule ∈ modules`.

*Modules* are primary building blocks of a model. Each module contains a set `attributes` with their default values, `signals` used in communication and `timers`. The communication topology between a module and its internal parts is given with a set of `connectors`. The behavior of a module is given with its *state machine*.

**Definition 11 (Module)**

A *module* is a tuple `module = (SM, parts, attributes, signals, connectors, timers, defaultValues)`, where:

- `SM` is a state machine that specifies the behavior of the module (see Definition 17),
- `parts` is a set of parts in the module (see Definition 12),
- `attributes` is a set of module variables,
- `signals` is a set of signals (see Definition 14),
- `connectors` is the connection relation (see Definition 15),
- `timers` is a set of timers declared for this module,
- `defaultValues` is the default, concrete valuation of variables in `attributes`.

As mentioned before to access elements of some module `module` we use the dot notation: `module.SM`, `module.parts`, `module.attributes` and so on.

We define a closure operation on elements of a module, to gather a union of elements declared in the module, in parts of the module, and recursively in parts of its parts. For instance, for some module `module`, the closure on parts `module.parts*` includes all parts in `module` and all parts in modules of its parts. That is, `module.parts*` is the smallest set satisfying:

\[
module.parts* = module.parts \cup \bigcup_{\text{part} \in \text{module.parts}} \text{part.type}.parts*
\]

where `part.type` is a module (see later definition of a part in Definition 12).

The hierarchical structure of models is possible with the use of `parts` that are instances of other modules. Each part has a reference to its type, which is a module. Additionally a part has a
boolean parameter that indicates, whether it is fixed, that is, the part is automatically created with its parent.

**Definition 12** (Part)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \). In this model a part is defined as \( \text{part} = (\text{type}, \text{isFixed}) \), where \( \text{type} \in \text{modules} \) is the type of a part and \( \text{isFixed} \in \{\text{true}, \text{false}\} \).

In order to navigate in the hierarchy of modules we define *part paths*, which are sequences of parts that follow the inclusion relation of modules and their parts.

**Definition 13** (Part path)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \). We define a part path to be a sequence of parts: \( \text{partPath} = (\text{part}_1, \ldots, \text{part}_n) \). We require that the sequence follows the inclusion relation, that is, \( \forall (i \leq n) : \text{part}_i \in \text{part}_{i-1}.\text{type.parts} \). We define the following operations on part paths:

- access the \( n \)th part in the path: \( \text{partPath}[n] \) (paths indexes start at 1),
- get the length of the path: \( \text{length} (\text{partPath}) \)
- check whether a part path is empty \( \text{isEmpty}(\text{partPath}) \). It returns \text{true} only for an empty part path, which is denoted with \( \langle \rangle \),
- return the last part in the path (i.e., the most deeply nested part):
  \( \text{last}(\text{partPath}) = \text{partPath}[\text{length}(\text{partPath})] \) if \( \text{partPath} \) is not empty,
- remove the last part: \( \text{levelUp}(\text{partPath}) \) if \( \text{partPath} \) is not empty,
- get the type of the path, that is the type (module) of the last part in the path \( \text{type}(\text{partPath}) = \text{last}(\text{partPath}).\text{type} \) if \( \text{partPath} \) is not empty,
- concatenating two paths \( \text{partPath}_1 \) and \( \text{partPath}_2 \): \( \text{partPath}_1 \odot \text{partPath}_2 \), we can concatenate parts only if the resulting path will respect the inclusion of parts, so the module of the last part in \( \text{partPath}_1 \) contains a part that starts \( \text{partPath}_2 \), that is, \( \text{partPath}_2[1] \in \text{type}(\text{partPath}_1).\text{parts} \),
- create a set consisting of an empty path and all possible part paths, which have some module as the root, that is:
partPaths(module) = \{\langle \rangle \} \cup \{ partPath \mid partPath[1] \in module.parts \},

- checking if a path is a prefix of some other path isPrefix(partPath_1, partPath_2) which returns true if partPath_2 is the same for the first length(partPath_1) as partPath_1.

Example 5. We define an example model of traffic lights: model\text{traffic} = (module\text{traffic}, queues\text{traffic}, queuesMap\text{traffic}, topPart\text{traffic}, topModule\text{traffic}), where:

- modules represent car lights, walk (pedestrian) lights and a controller that coordinates them:
  module\text{traffic} = \{ CarLights, WalkLights, Controller \},

- queues are separate for each module: queues\text{traffic} = \{ CarLightsQueue, WalkLightsQueue, ControllerQueue \},

- queues are assigned to modules: queuesMap = 
  \{ CarLights \mapsto CarLightsQueue, WalkLights \mapsto WalkLightsQueue, Controller \mapsto ControllerQueue \},

- topPart\text{traffic} = (Controller, true) (in the sequel we use simply top to denote this top part),

- topModule\text{traffic} = Controller.

The communication in CFFSMSs is based on signals, which can be received or sent by a part and which may include variables. These variables are used during an execution of a model to communicate data between modules. A signal contains also two boolean parameters to indicate whether it is an output signal and whether it is external.

Definition 14 (Signal)
A signal is a tuple signal = (variables, isOut, isExternal), where variables is a set of variables, isOut \in \{ true, false \} indicates whether the signal is sent by this module and isExternal \in \{ true, false \} indicates whether the signal is received from the environment (or sent there). For a set of signals signals we partition them into signalsOut and signalsIn such that signals = signalsOut \cup signalsIn with signalsOut = \{ signal \mid signal.isOut = true \} and signalsIn = \{ signal \mid signal.isOut = false \}.

In order to specify how modules communicate, each module contains a set of connectors. They represent the connection between signals in a sending and a receiving part and how variables in the
connected signals are mapped. Because a module can declare a connector for parts that are deeper in the hierarchy, a sender of some signal might need to access the connection relation of its parent or its ancestors to find the connected input signal.

**Definition 15 (Connector)**

Let \( \text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues}) \). A connector is a tuple: 
\[
\text{connector} = (\text{signalIn}, \text{receiver}, \text{signalOut}, \text{sender}, \text{mapping}),
\]
where:

- \( \text{signalIn} \in \text{module} . \text{signalsIn} \) is the received signal,
- \( \text{receiver} \in \text{partPaths}(\text{module}) \) is the part path to the receiving module, with an empty path \( \langle \rangle \) indicating that the given module \( \text{module} \) is the receiver,
- \( \text{signalOut} \in \text{module} . \text{signalsOut} \) is the sent signal,
- \( \text{sender} \in \text{partPaths}(\text{module}) \) is the part path to the sending module (as in a case of the receiver an empty path indicates that \( \text{module} \) is the sender),
- \( \text{mapping} : \text{signalOut} . \text{variables} \rightarrow \text{signalIn} . \text{variables} \) is a function that maps variables in the input signal to the variables in the output signal.

At this point note that although parts take part in the communication, we assign queues to modules, so parts of the same module share a queue. This allows defining the queues when models are specified, since only modules are known at that time – parts might be created at runtime. However, during the execution of a model we access the queues using the actual recipients, that is, parts (see Definition 27).

An important property of CFFSMs and of UML-RT is encapsulation, which means that modules have disjoint parts, variables, signals (and their variables) and timers.

**Definition 16 (Encapsulation)**
For \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \) we require the following:

\[
\forall \text{module}_1, \text{module}_2 \in \text{model} . \text{modules} : \text{module}_1 \neq \text{module}_2 \Rightarrow \\
(\text{module}_1 . \text{attributes} \cap \text{module}_2 . \text{attributes} = \emptyset) \land \\
(\text{module}_1 . \text{signals} \cap \text{module}_2 . \text{signals} = \emptyset) \land \\
(\text{module}_1 . \text{parts} \cap \text{module}_2 . \text{parts} = \emptyset) \land \\
(\text{module}_1 . \text{timers} \cap \text{module}_2 . \text{timers} = \emptyset))
\]

**Example 6.** Table 4.1 specifies the three modules introduced in Example 5. In the figure we see that the top module Controller contains two non-fixed parts walk and cars of types WalkLights and CarLights. Figure 4.1 shows the connectors defined in Controller (the other modules do not contain connectors). In the figure arrows indicate the send-receive direction and signals are identified by their names, which are the same for input and output. For instance, cars can send redSet, greenSet and carsStarted to the controller. Also, carsFirst and walkFirst are external signals sent from the model’s environment. We omitted signal variables and the mapping of variables in output signals to variables in input signals.

![Diagram of traffic controller model](image)
### Table 4.1: Specification of modules in Example 6.

<table>
<thead>
<tr>
<th>Module</th>
<th>Controller</th>
<th>CarLights</th>
<th>WalkLights</th>
</tr>
</thead>
<tbody>
<tr>
<td>parts</td>
<td>walk = (WalkLights, false)</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td>cars = (CarLights, false)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attributes</td>
<td>carsD: int</td>
<td>yellowD: int</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td>walkD: int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defaultValues</td>
<td>carsD = 120</td>
<td>yellowD = 20</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td>walkD = 60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signalsIn</td>
<td>carsStarted()</td>
<td>toRed(delayY:int)</td>
<td>toWalk()</td>
</tr>
<tr>
<td></td>
<td>redSet()</td>
<td>toGreen(delayY:int)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>greenSet()</td>
<td></td>
<td>stopWalk()</td>
</tr>
<tr>
<td></td>
<td>walkStarted()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>walkSet()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>walkStopped()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signalsIn</td>
<td>carsFirst</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(delayC:int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>walkFirst</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(delayW:int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signalsOut</td>
<td>toRed(delayY:int)</td>
<td>carsStarted()</td>
<td>walkStarted()</td>
</tr>
<tr>
<td></td>
<td>toGreen(delayY:int)</td>
<td>redSet()</td>
<td>walkSet()</td>
</tr>
<tr>
<td></td>
<td>toWalk()</td>
<td>greenSet()</td>
<td>walkStopped()</td>
</tr>
<tr>
<td></td>
<td>stopWalk()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>timers</td>
<td>carsTimer</td>
<td>yellowTimer</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td>walkTimer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.1.3 Structure of a state machine

The behavior of each module is specified with a *state machine*. A state machine has locations (also called states if no confusion with the notion of execution state can arise) and guarded transitions between them. The transitions also contain functions, which define effects to be executed. There are several types of effects and they may be changing the values of attributes, sending signals to other parts, instantiating modules of non-fixed parts and setting timers.

**Definition 17** (State machine)

Let $\text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$. A state machine is $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$, where:

- **locations** is a set of locations, where $\text{locationInit} \in \text{locations}$ is an initial state (see Definition 18),
- **transitions** is a set of transitions (see Definition 19),
- **guards** is a set of guards (see Definition 20),
- **functions** is a set of functions (see Definition 21).

A *location* in a state machine contains information about transitions that are outgoing from a given location.

**Definition 18** (Location)

Let $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$ be a state machine of some module. A location in this state machine is defined as $\text{location} = (\text{outgoing})$, where $\text{outgoing} \subseteq \text{transitions}$ are transitions that have this location as their source state.

*Transitions* in a state machine are guarded and they include *functions* to specify effects of firing them. In both guards and functions we can have formulas and terms, and their variables are required to be in the set of attributes and, possibly, input variables of an input signal, i.e., a *trigger*.

**Definition 19** (Transition)

Let $\text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$ and $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$. A transition is defined as $\text{transition} = (\text{source}, \text{trigger}, \text{guard}, \text{function}, \text{target})$, where:
- source ∈ locations is the source location of the transition,
- trigger ∈ signalsIn ∪ timers ∪ \{init()\} is the triggering event,
- guard ∈ guards is the guard condition (see Definition 20),
- function ∈ functions is the function that specifies effects (see Definition 21),
- target ∈ locations is the target location.

Transitions contain guards, that is, conditions defined as formulas over all domains of the model. A variable of this formula is either an attribute or an input variable in the trigger of the transition the guard occurs in.

**Definition 20 (Guard)**

Let \( \text{trans} = (\text{source}, \text{trigger}, \text{guard}, \text{function}, \text{target}) \). The guard is a formula \( \text{guard} \in \mathcal{F} [\text{attributes} \cup \text{trigger.variables}] \), where attributes are the attributes of the module that contains the state machine with trans.

Transitions are labeled with functions, which are sets of cases. A case represents a conditional action and is designed to support branching in action code associated with transitions and/or states in UML-RT State Machines. We require that cases are total, that is, they cover all possible values of input variables.

**Definition 21 (Function)**

Let \( \text{trans} = (\text{source}, \text{trigger}, \text{guard}, \text{function}, \text{target}) \) be a transition. A function is defined as a set of cases \( \text{function} = (\text{cases}) \). Each case is a pair \( \text{case} = (\text{condition}, \text{effect}) \), where \( \text{condition} \in \mathcal{F} [\text{attributes} \cup \text{trigger.variables}] \) (with attributes being attributes of a module). An effect is a set of several different actions (see Definition 22). We require that conditions are total, that is, if \( \text{function} = (\text{condition}_1, \text{effect}_1), \ldots, (\text{condition}_n, \text{case}_n) \) we have \( \text{condition}_1 \lor \ldots \lor \text{condition}_n \) is valid for all valuations of variables used in conditions.

**Definition 22 (Effect)**

Let \( \text{trans} = (\text{source}, \text{trigger}, \text{guard}, \text{function}, \text{target}) \) be a transition in a state machine \( \text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit}) \) in module \( (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues}) \). An effect is \( \text{effect} = \text{update} \cup \text{out} \cup \text{setTimers} \cup \text{new} \cup \text{destroy} \), where:
- update = update(newValues) defines an update of attributes, which is represented with a valuation val ∈ attributes. The variables used in the valuation (in the terms) are attributes or input variables of a trigger, i.e., ∀ att ∈ attributes : vars(newValues(att)) ⊆ attributes ∪ trigger.variables,

- out = out(outputs) defines a possibly empty sequence of output signals with their values. This sequence is [output₁,...,outputₙ], where outputᵢ = (outSignalᵢ,outValuesᵢ) and outSignalᵢ ∈ signalsOut is the output signal and outValuesᵢ ∈ outputᵢ:variables is the valuation of variables. As in the update, the terms used to specify values of output variables contain only attributes and variables of a trigger, that is, for any (outSignal,outValues) of an output sequence we have ∀ v ∈ outSignal:variables : vars(outValues(v)) ⊆ attributes ∪ trigger.variables,

- setTimers = setTimers(timersEffect) declares timers to be set, which are in the module, that is, timersEffect ⊆ timers, where timers is a set of timers declared for the given module,

- new = new(partsEffect) declares parts of a module that are to be instantiated, we have partsEffect ⊆ parts,

- destroy = destroy(partsEffect) declares parts of a module that are to be destroyed, we have partsEffect ⊆ parts.

Example 7. Figures 4.2, 4.3 and 4.4 show the state machines for modules introduced in Example 6. The module Controller instantiates its parts and then waits for them to complete startup (indicated with triggers: walkStarted and carsStarted). After initialization, the controller responds to one of the external signals (walkFirst or carsFirst) to start either pedestrians or cars. Note that functions associated with these transitions have two cases depending on the value of input variable delayW (or delayC). If this value is greater than zero it is used to update the value of attributes walkD or carsD. After the initialization the controller cycles through different configurations of cars and pedestrian lights. In the functions used in the state machine we have different actions: setting timers (denoted as setTimers), updating attributes (denoted as update), output of signals (denoted as out) and initializing new parts (denoted as new). State machines for car lights and pedestrian lights are given in Figures 4.3 and 4.4, respectively, and they cycle through lights colors in response to signals from the controller.
4.2 Semantics of CFFSMs

In the previous section we introduced the syntax of CFFSMs, and in this section we describe their semantics. As indicated at the beginning of this chapter, the CFFSMs semantics follows the semantics of UML-RT models as much as possible.

4.2.1 Representation of execution

The semantics of a model is defined with a labeled transition system (LTS). In this system states represent current execution states of the model and execution transitions represent events that can change these states.
CHAPTER 4. SYNTAX AND SEMANTICS OF CFFSMS

Figure 4.3: State machine of CarLights. Labels as in Figure 4.2.

Figure 4.4: State machine of WalkLights. Labels as in Figure 4.2.
Definition 23 (Labeled Transition System)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \). The semantics of \( \text{model} \) is a labeled transition system \( \text{LTS} = (\Sigma, \Delta, \sigma_0, \Phi) \), where:

- \( \Sigma \) is a set of execution states, each of which reflects a state of the model in a particular moment of execution (see Definition 25),
- \( \Delta \) is a transition relation (see Definition 30),
- \( \sigma_0 \in \Sigma \) is the initial state (see Definition 31),
- \( \Phi \) is a set of all execution parts in the model.

Before we describe execution states we introduce the notion of \textit{execution parts} used to identify parts during the execution. Since a module can be instantiated by different parts, we need to distinguish between these different instantiations. To achieve that we use part paths. More specifically, we use part paths that have the top part of a model as their first part and we call them execution parts. In this way all instantiated parts have some identifier in the form of their execution part, that is, the part path leading to them.

Definition 24 (Execution part)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \). A path \( \text{topPartPath} \) has \( \text{topModule} \) as its root or which is empty, i.e, \( \text{topPartPath} \in \text{partPaths}(\text{topModule}) \). We define an execution part \( \phi \) to be a concatenation of the top level part \( \text{topPart} \) and some \( \text{topPartPath} \). We have \( \phi = (\text{topPart}) \circ \text{topPartPath} \). All operations defined for part paths are also applicable to execution parts.

An \textit{execution state of a model} consists of three elements: the first one is the current contents of queues, the second one is the current state of all initialized parts in a model and the third one is the set of currently explored execution parts. This last element is obviously a subset of all possible execution parts \( \Phi \) in a model.

Definition 25 (Execution state)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \) and the semantics of this model is \( \text{LTS} = (\Sigma, \Delta, \sigma_0, \Phi) \). An execution state of \( \text{model} \) is a tuple \( \sigma = (m, s_{\text{top}}, \Phi) \), where:
- $m$ is a map that maps a queue from the model to a current execution queue, i.e., $m : \text{queues} \rightarrow Q$ (see Definition 27),

- $s_{\text{top}}$ is the execution state of the top level part and its inner parts and recursively their parts (see Definition 28),

- $\Phi \subseteq \Phi$ is a set of currently explored execution parts, which is a subset of all execution parts $\Phi$ possible in a model.

Execution queues in an execution state of a model contain execution signals, which are signals in a model along with the values assigned to their variables, gathered in their valuations. Depending on the type of execution (concrete or symbolic), these values can be constants or terms.

**Definition 26 (Execution signal)**

Let $\text{signal} = (\text{variables}, \text{isOut}, \text{isExternal})$ be a signal. An execution signal is a pair $\lambda = (i, v)$, where $i = \text{signal}$ and $v \in \text{variables}$ is a valuation of variables of this signal.

With this definition we can define the contents of an execution state of a model. We start with the execution queues. These queues are given as a sequence of pairs consisting of the receiving execution part and an execution input signal itself. Note that these queues are different from queues declared in a model (see Definition 10), which are merely placeholders for queues. During the execution, in each execution state, we assign to each queue in a model an execution queue and its actual contents.

**Definition 27 (Execution queue)**

An execution queue $q$ is a sequence of pairs $(\phi, \lambda)$ consisting of an execution part identifying the receiver of a signal and an execution input signal. An execution queue $q$ has operations:

- $\lambda = \text{.dequeue}(q, \phi)$ returns the first execution signal in the queue sent to the execution part $\phi$,

- $\text{enqueue}(q, (\phi, \lambda))$ adds an element to the end of the queue and returns the updated queue.

An execution state of a part in the model contains its execution part, the current state details (i.e., location, valuation and a set for timers as in Definition 29) for this part, and the map of its parts and their respective execution states.
Definition 28 (Execution state of a part)

Let $\text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$ with its state machine $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$ and let execution part $\phi$ be of type $\text{module}$. The execution state $s$ of this part is given as a tuple $(\phi, \omega, s_{\text{parts}})$, where:

- $\phi$ is an execution part,
- $\omega$ contains state details (see Definition 29),
- $s_{\text{parts}}$ maps the inner parts of $\text{module}$ to their execution states, that is, $s_{\text{parts}} : \text{parts} \rightarrow S$, where $S$ are execution states of parts of the module.

We define the operation $\text{getState}(s_{\text{top}}, \phi)$, which takes two parameters: an execution state of a top level part $s_{\text{top}}$ and an execution part $\phi$. The operation returns the execution state of the part reached by following the path in $\phi$. We will extend the operation to take the entire execution state as a parameter, i.e., $\text{getState}(\sigma, \phi)$ to be $\text{getState}(\sigma.s_{\text{top}}, \phi)$. We often denote an execution state for $\phi$ with $s_{\phi}$.

State details contain the following information about the execution of a part: the location that the part is in, the current valuation of its attributes and the set of currently set timers.

Definition 29 (State details)

Let $\text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$ with a state machine $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$. Let the type of the execution part $\phi$ be $\text{module}$. The state details of the execution of this part are defined as $\omega = (l, v, t)$ or $\omega = \varepsilon$, where:

- $l \in \text{locations}$ is a current location,
- $v \in \text{attributes}$ is a valuation of attributes in a module,
- $t \subseteq \text{timers}$ is a set of currently set timers,
- $\varepsilon$ denotes the empty state.

Example 8. An example of an execution state for the model introduced in Example 5 is given in Figure 4.5. We use an abbreviated mapping for queues, that is, we map each module directly to
the respective execution queue, because in this example each module has its own queue. Such an abbreviation is not possible if modules share queues. In our case instead of \texttt{queuesMap(Controller)}, which is \texttt{ControllerQueue}, we use \texttt{Controller}. According to the figure we have:

- queues \( m \) :
  
  \[
  \{ \text{Controller} \mapsto \{(\text{top}).\text{carsStarted()}\}, \\
  \text{CarLights} \mapsto [], \text{WalkLights} \mapsto [] \}
  \]

- states \( s_{\text{top}} \) :
  
  \[
  ((\text{top}), \text{Walk ready}, \{\text{carsD} = 120, \text{walkD} = 60\}, \emptyset, \\
  \text{cars} \mapsto ((\text{top}, \text{cars}), \text{Blinking}, \{\text{yellowD} = 20\}, \emptyset, \emptyset), \\
  \text{walk} \mapsto ((\text{top}, \text{walk}), \text{Blinking}, \emptyset, \emptyset, \emptyset)).
  \]

- using access operations we have:
  
  - \( \text{getState}(s_{\text{top}}, (\text{top}, \text{walk})) = ((\text{top}, \text{walk}), \text{Blinking}, \emptyset, \emptyset, \emptyset) \),
  
  - \( \text{getState}(s_{\text{top}}, (\text{top}, \text{walk})).\omega.l = \text{Blinking} \).

Figure 4.5: An example of an execution state. The upper part contains the contents of queues in the form \texttt{module: contents}. The lower part contains the states of parts. In each state of a given part we omit the explicit indication of its execution part and we show only the details, that is \((\phi, \omega, s_{\text{parts}})\) is shown as \(\omega = (l, v, t)\). The map of states is shown with indentation.

\(\textit{Execution transitions}\) represent a change between states of execution. These transitions are distinct from the transitions in state machines which include guards, triggers and functions. Here we use \textit{execution events} to label transitions. Execution events include the trigger, type of event and any sent signals. We distinguish between six different types of events, which match the execution rules presented in Section 4.2.3.
Definition 30 (Execution transition)

Let $LTS = (\Sigma, \Delta, \sigma_0, \Phi)$ be the $LTS$ of some model in which $\text{signalsIn}$ are input signals, $\text{signalsOut}$ output signals and $\text{timers}$ are timers. An execution transition $\delta$ is a tuple $(\sigma_{source}, \nu, \sigma_{target}) \in \Delta$, where:

- $\sigma_{source} \in \Sigma$ is the source execution state,
- $\nu$ is an execution event, which is a tuple $(\phi, t, g, o)$, where:
  - $\phi$ is the execution part that caused the transition (i.e., the part for which we used the rule as defined in Section 4.2.3),
  - $t$ is the type of the event with $t \in \{\text{match}, \text{external}, \text{init}, \text{timeout}, \text{drop}\}$,
  - $g$ is the trigger with $g \in \Lambda \cup \text{timers} \cup \{\text{init()}\}$, where $\Lambda = \{(i, v) \mid i \in \text{signalsIn}\}$ and $\text{init()}$ is an initial signal (as defined in the default rule, see page 64),
  - $o$ is an output sequence $[o_1, \ldots, o_n]$ with $o_k \in \{(i, v) \mid i \in \text{signalsOut}\}$,
- $\sigma_{target} \in \Sigma$ is the target execution state of this transition.

The transitions $\Delta$ are generated by rules introduced in Section 4.2.3.

Finally, we show how the initial state $\sigma_0$ of an execution $LTS$ is generated. The queues are initialized to empty ones and all fixed parts are set to the initial states and initial values of attributes. We use a helper operation $\text{init}$ that initializes the top level.

Definition 31 (Initial execution state)

Let $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ with a top module $\text{topModule} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$ and $\text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$. The execution is defined as $LTS = (\Sigma, \Delta, \sigma_0, \Phi)$. The initial state of its execution is a state $\sigma_0 = (m, s_{top}, \Phi)$ with:

- $m$ is a map of queues to execution queues, which is initialized as $\forall q \in \text{queues} : m(q) = []$
- $s_{top} = \text{init}(\langle \text{topPart} \rangle)$ is the result of initialization of top part (see Definition 32),
- $\Phi = \text{initialized}(s_{top})$ is the result of gathering all execution parts which are initialized (see Definition 32).
The initialization operation of an execution part returns the execution state $s$ of this part. In this state we initialize its details ($\omega_0$) to the initial location and initial valuation. We recursively initialize all fixed parts in the module (i.e., the type of the execution part).

**Definition 32** (Initialize part)

Let module $= (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues})$ and $\phi$ be an execution part to be initialized, such that type($\phi$) = module. A state machine for this module is SM $= (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit})$. We define operation $s = \text{init}(\phi)$, which returns an initial execution state for a part:

$$s = (\phi, \omega_0, \{\text{part} \mapsto s_{\text{part}} \mid \text{part} \in \text{type}(\phi).\text{parts} \land \text{part}.\text{isFixed} \land s_{\text{part}} = \text{init}(\pi \odot \langle \text{part} \rangle)\})$$

where:

$$\omega_0 = (\text{locationInit}, \text{defaultValues}, \emptyset)$$

We also define the operation initialized that gathers all initialized parts. The operation goes through parts and check whether state execution details are non-empty. The operation is defined as the smallest subset of execution parts that satisfies:

$$\text{initialized}(s) = \begin{cases} 
\{\phi\} \cup \bigcup_{\text{part} \in \text{type}(\phi).\text{parts}} \text{initialized}(s.s_{\text{parts}}(\text{part})) & \text{if } s.\omega \neq \varepsilon \\
\bigcup_{\text{part} \in \text{type}(\phi).\text{parts}} \text{initialized}(s.s_{\text{parts}}(\text{part})) & \text{if } s.\omega = \varepsilon
\end{cases}$$

### 4.2.2 Evaluation of functions

State machines in CFFSMSs contain functions assigned to the transitions. In this section we define how the functions are evaluated during the execution. They are evaluated separately for each case in the context of a current execution state, for some part and for some valuations of variables. In the specification below we use the regular values of variables, and note that a concrete valuation is just a special case of a regular valuation.

**Definition 33** (Case evaluation)

We define the following operation $(\sigma', \text{output}) = \text{eval}(\text{effect}, \sigma, \phi, \text{val}_c)$. The parameters of this
operations are:

- **effect** ⊆ \{update, out, setTimers, new, destroy\} indicates the effects to be evaluated (attached to the case that is considered),

- **σ** = (m, s\textsubscript{top}, \Phi) is the current execution state of the model,

- **φ** is an execution part,

- **val**\textsubscript{c} is the valuation of attributes type(\phi).attributes as stored in the state details σ.ω.\textsubscript{val} as stored in the state details σ.ω.\textsubscript{val} and variables of the triggering event (if any).

The result of evaluation is an updated execution state of a model σ′ = (m′, s\textsubscript{top}′, \Phi′) and a sequence of execution output signals output. We use s\textsubscript{φ} = getState(s\textsubscript{top}, \phi) and s′\textsubscript{φ} = getState(s\textsubscript{top}′, \phi) to refer to the original and updated execution state, respectively. The σ′ and output are:

- evaluation of an update effect results in an updated valuation of attributes for a part \phi. Let this valuation be denoted by \text{v}′\textsubscript{φ} = s′\textsubscript{φ}.ω.v. These new values of attributes are calculated by updating the attributes as specified in update.newValues using the current valuation val\textsubscript{c}. The new valuation \text{v}′\textsubscript{φ} is defined as as follows:

\[
∀ \text{att} ∈ type(\phi).\text{attributes} : \text{v}′\textsubscript{φ}(\text{att}) = \text{update.newValues(\text{att}) || } val\textsubscript{c} \]

- evaluation of a timers effect results in adding the timers in setTimers.timersEffect to the timers currently set. So we have:

\[
s′\textsubscript{φ}.ω.t = s\textsubscript{φ}.ω.t \cup \text{setTimers.timersEffect} 
\]

- evaluation of a new effect results in adding a part to the parts map s\textsubscript{φ}.s\textsubscript{parts} a map that assigns the new part from new.partsEffect to its initial execution state and adding their execution parts to the currently explored parts \Phi′.

\[
s′\textsubscript{φ}.s\textsubscript{parts} = s\textsubscript{φ}.s\textsubscript{parts} \cup \{p \mapsto \text{init}(\phi \circ \langle p \rangle) \mid p ∈ \text{new.partsEffect \land φ} \circ \langle p \rangle ∈ \Phi\}
\]

and

\[
\Phi′ = \Phi \cup \text{initialized}(\phi \circ \langle p \rangle)
\]
- evaluation of a *destroy* effect results in removing execution states for parts that are being destroyed and removing the execution parts from the set of explored parts $\Phi'$, that is:

$$s^{'\prime}\phi = s\phi \setminus \{p \mapsto s_p \mid p \in \text{destroy.partsEffect}\}$$

and

$$\Phi' = \Phi \setminus \{\phi \circ (p)\}$$

Figure 4.6: An illustration of the output effect.

- evaluation of an *output* effect has 2 results as informally shown in Figure 4.6. The first result is generating a sequence of output signals, which serves as an output sequence $\nu.o$ on a label of an execution transition (see Definition 30). The second result is adding signals to queues. Let $\text{out.outputs}$ be a sequence $[\text{output}_1, \text{output}_2, \ldots, \text{output}_n]$, where $\text{output}_i = (\text{outSignal}_i, \text{outValues}_i)$. The effect of an output action is considered for each element $\text{output}_i$, that is:

$$\forall \text{output}_i \in \text{outputs} : \text{send}(\text{getInput}(\text{output}_i), \text{getReceiver}(\text{output}_i))$$

$$\land \text{add}(\text{output}, \text{getExec}(\text{output}_i))$$

(4.1)
### Algorithm 5

An outline of function `targetConn(output, φ_context, suffix)`.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Require</strong>: an output signal <code>output</code></td>
<td></td>
</tr>
<tr>
<td><strong>Require</strong>: a current execution path <code>φ_context</code></td>
<td></td>
</tr>
<tr>
<td><strong>Require</strong>: a suffix path part <code>suffix</code></td>
<td></td>
</tr>
<tr>
<td><strong>Ensure</strong>: connector with required output signal (output) and sender (suffix)</td>
<td></td>
</tr>
<tr>
<td><strong>Ensure</strong>: <code>φ_context</code> execution part which type contains connector</td>
<td></td>
</tr>
<tr>
<td><code>if</code> <code>isEmpty(φ_context)</code> <code>then</code></td>
<td><code>return (null, null)</code></td>
</tr>
<tr>
<td><code>for all</code> <code>connector ∈ type(φ_context).connectors</code> <code>do</code></td>
<td></td>
</tr>
<tr>
<td><code>if</code> <code>connector.signalOut = output.outSignal ∧ connector.sender = suffix</code> <code>then</code></td>
<td><code>return (φ_context, connector)</code></td>
</tr>
<tr>
<td><code>lastPart ← last(φ_context)</code></td>
<td></td>
</tr>
<tr>
<td><code>φ_context ← levelUp(φ_context)</code></td>
<td></td>
</tr>
<tr>
<td><code>suffix ← ⟨lastPart⟩⊙ suffix</code></td>
<td></td>
</tr>
<tr>
<td><code>9: targetConn(output, φ_context, suffix)</code></td>
<td></td>
</tr>
</tbody>
</table>

The operation `add(output, λ)` adds an output execution signal to the end of `output`. In this operation the output execution signal `λ` is the result of the operation `getExec`. The function takes `output_i = (outSignal_i, outValues_i)` as its parameter and returns an output execution signal `λ_{output} = (i, v)`. Signals `outSignal` and `i` are the same. The valuation `v` is obtained by using the valuation `outValues` and replacing variables with their current values from the valuation `val_c`. We have

\[
getExec(output_i) = (outSignal_i, v), \text{ such that }
\forall \text{ var \in outSignal.variables : } v(\text{var}) = outValues(\text{var}) \parallel val_c
\]

Now we define the `send` operation used in Equation 4.1. This operation simply adds an input execution signal `λ_{input}` and the receiving execution part `φ_{input}` (both obtained with the operations defined later) to a queue using the enqueue operation (see Definition 27). In order to find the receiving queue we use `queuesMap(module)` (module is `type(φ_{input})`) and then the execution queue `m'(queue)` is a queues map during the execution.

\[
send(λ_{input}, φ_{input}) = m'(queuesMap(type(φ_{input}))).enqueue(φ_{input}, λ_{input})
\]

In order to define operations `getInput` (getting the input signal) and `getReceiver` (getting the receiving execution part) used as parameters in `send` in Equation 4.1, we use a helper function `targetConn`. This operation searches for a target connector, that is a connector with an output signal equal to `output` and a sender equal to the current execution part `φ`. The operation `targetConn(output, φ_context, suffix)` is outlined in Algorithm 5. This function checks connectors of the current module (identified by `type(φ_context)`). The connector is returned if its `signalOut`
matches the output signal in the connector and suffix matches sender in the connector. If the connector is not found in the current module, the parent of the module is checked. We remove the last part from context and we add this part to the beginning of suffix. In this way context will refer to a parent module and suffix will keep the path from the current module to the execution part we started from. So the concatenation of context and context is the execution part for which the initial call to targetConn is made. The function returns a pair (context, connector). The initial call we make is targetRes = targetConn(output, φ, ⟨⟩). With the above we can define the functions:

\[ \text{getInput}(\text{output}, i) = (\text{targetRes}.\text{connector}.\text{signalIn}, \text{makeVal(outValues, targetRes.connector.mapping)}) \]

and

\[ \text{getReceiver}(\text{output}, i) = \text{targetRes}.\text{context} \odot \text{targetRes.connector.receiver} \]

The above operation uses the function makeVal that creates inputValues. This valuation is the valuation of the input signal variables. To obtain it we use the valuation outValues, (part of the output, i in the effect) and mapping, which maps output signal variables to input signal variables. The actual values of variables are obtained by replacing the variables in the output signal valuation with the current values of input variables and attributes val_c.

\[ \text{inputValues} = \text{makeVal(outValues, mapping)} \]

such that:

\[ \forall \text{var} \in \text{outSignal.variables} : \]

\[ \text{inputValues(var)} = \text{outValues, (mapping(var))} \parallel \text{val_c} \parallel \]

### 4.2.3 Execution rules

We are now ready to introduce the execution rules. We use the rules to provide the semantics as mentioned in the specification of the transitions \( \Delta \) of \( LTS = (\Sigma, \Delta, \sigma_0, \Phi) \) (see Definition 23). Each rule has a precondition and a result, that is, a transition \( \sigma \xrightarrow{} \sigma' \). In the following rules we assume that \( \sigma' \) is the same as \( \sigma \) for all elements that are not explicitly changed in the rule. We do not prioritize rules explicitly and if a precondition is satisfied it is enough to fire a rule. This means
Algorithm 6 An outline of function \textit{isInitialTop}(s_{\text{top}}, \phi).

\begin{algorithm}
\caption{Algorithm 6 An outline of function \textit{isInitialTop}(s_{\text{top}}, \phi).}
\begin{algorithmic}[1]
\Require a current execution state $s_{\text{top}}$
\Require an execution path $\phi$
\If {\textit{isInitial}(s_{\text{top}}, \phi)}
\State $\phi' \leftarrow \text{levelUp}(\phi)$
\EndIf
\While {\textit{notEmpty}(\phi)}
\If {\textit{isInitial}(s_{\text{top}}, \phi')}
\State return false
\EndIf
\State $\phi' \leftarrow \text{levelUp}(\phi')$
\EndWhile
\State return true
\State return false
\end{algorithmic}
\end{algorithm}

that the same rule can be applied many times in the context of an execution state. Additionally, an execution part $\phi$ that will be explored is in the set of explored parts that is stored in each execution state as $\sigma.\bar{\Phi}$.

First, we define the \textit{default rule}. This rule is applied to an execution part that is in its initial state and is used to perform a default transition in a part and all its inner parts that are initialized.

The rule is defined as follows.

\textbf{Rule 1} (Default rule). Let $\sigma = (m, s_{\text{top}}, \bar{\Phi})$ be an execution state of some model with some execution part $\phi$. The rule for generating a transition $\sigma \xrightarrow{\phi, \text{inst, inst}(\cdot), o} \sigma'$ is:

\[
\exists \phi \in \bar{\Phi} : \text{isInitial}(s_{\text{top}}, \phi) \land \\
\text{isInitialTop}(s_{\text{top}}, \phi) \land \\
(\sigma', o) = \text{fireDefault}(\phi)
\]

\[
\sigma \xrightarrow{\phi, \text{inst, inst}(\cdot), o} \sigma'
\]

where:

- \textit{isInitial}(s_{\text{top}}, \phi) checks whether an execution state for $\phi$ is in its initial state:

\[
\text{isInitial}(s_{\text{top}}, \phi) = (\text{getState}(s_{\text{top}}, \phi).\omega.l = \text{type}(\phi).\text{SM}.\text{locationInit})
\]

- \textit{isInitialTop}(s_{\text{top}}, \phi) checks whether $\phi$ is an execution part that has no ancestor in the initial state (it is the top most part to be initialized). The algorithm for this check is shown in Algorithm 6. We check whether the part is indeed initial. If so, we remove the last part from the path, so we move to the parent. If we find the parent that is initial (line 4) we return a negative answer.
Algorithm 7 An outline of function fireDefault(φ, σ).

Require: an execution path φ
Require: a current execution state σ
Ensure: σ_result is an execution state which has the default transition of φ fired
Ensure: o is an output sequence resulting from firing the default transition

for all part ∈ type(φ).parts do
φ_part ← φ ⊙ ⟨part⟩
3: if isInitial(σ.top, φ_part) then
   (σ_result, o_result) ← fireDefault(φ_part, σ)
   o ← add(o, o_result)
6: σ ← σ_result
sφ ← getState(σ.top, φ)
trans ← sφ.ω.l.outgoing
9: case ← trans.function.cases
   (σ_result, o_result) ← eval(case.effect, σ, φ, sφ.ω.v)
   o ← add(o, o_result)
12: σ ← σ_result
getState(σ_result.top, φ).ω.l ← trans.target
return (σ_result, o)

- fireDefault(φ) performs the default transition in a given part and all its initialized parts. The order of execution is bottom-up, that is, we start from the innermost part. This mimics the UML-RT semantics in which the inner parts must be initialized first. The algorithm to perform this function is given in Algorithm 7. Lines 1 to 6 are responsible for initializing the inner parts and storing the results. In line 8 we retrieve a default transition (assuming that there is just one outgoing transition from the initial location) and, in the next line, the case. We assume that there is just one case, since this is an initial transition so there are no input variables and attributes which have their values set. Next, we take care of each case in the function attached to the transition. Finally, in line 13 the current location in the execution state is set to the target location of the default transition.

Example 9. Let us assume that the execution state we are exploring is σ₀ = S₁ as in Figure 4.7 (model introduced in Example 5). In this state the default rule applies to two parts: walk and cars. Let us consider the application of the rule to the latter part. We have φ = ⟨top,cars⟩, the initial transition is trans = (init,init(),true,function,Blinking). The function function has only one case: case = (true,{out([carsStarted()])}). The function fireDefault(φ, σ₀) proceeds to line 7, because there are no internal parts to initialize. The evaluation of the case leads to enqueuing the signal carsStarted() in the queue for top part. Firing the initial transition trans changes the current location in the execution state for φ to the target location Blinking. The resulting execution state is S₃. The initialization of the part walk results in a state S₂.
The next rule is the *match rule*, and it deals with matching signals present in a queue with triggers on transitions. This rule, like all the following rules, is applied only if there is a part and its state machine has a transition with a trigger and this triggering signal is at the head of the queue.

**Rule 2 (Match rule).** Let $\sigma = (m, s_{top}, \Phi)$ be an execution state of some model with some execution part $\phi$. The rule for generating a transition $\sigma \xrightarrow{\phi, \text{match}, \lambda, o} \sigma'$ is:
∀ φ ∈ Φ : ¬isInitial(s\textsubscript{top}, φ) ∧
∃ φ ∈ Φ : s\textsubscript{φ} = getState(s\textsubscript{top}, φ) ∧
q = m(queuesMap(type(φ))) ∧
λ = dequeue(q, φ) ∧
m'(queuesMap(type(φ))) = q ∧
val\textsubscript{c} = s\textsubscript{φ}.ω.v ∪ λ.v ∧
∃ trans ∈ s\textsubscript{φ}.ω.l.outgoing :
   λ.i = trans.trigger ∧
   trans.guard || val\textsubscript{c} || ∧
∃ case ∈ trans.function.cases :
   case.condition || val\textsubscript{c} || ∧
   (σ', o) = eval(case.effect, σ, φ, val\textsubscript{c}) ∧
   getState(σ', φ).ω.l = trans.target
\[ \sigma \xrightarrow{(φ, \text{match, λ, o})} \sigma' \]

In the above rule we first check whether all parts are initialized, since other rules may apply only after the default one. Next, we get an execution part φ from parts being currently explored Φ. We chose any part that satisfies the preconditions. Note that if there is more than one part for which the rule can be applied we apply the rule with different execution parts. For the execution part we use we get its current execution state s\textsubscript{φ} and its execution queue q. According to the map queuesMap we find the right queue for the module of φ (i.e., type(φ)) and then we find an execution queue according to m. We retrieve the first element from the queue that has φ as its receiver. The execution signal in this element is denoted with λ. The valuation in this signal (λ.v) along with the valuation of attributes (s\textsubscript{φ}.ω.v) are used to set the current valuation val\textsubscript{c}. Next, we select one of the outgoing transitions from the current location and we check whether its trigger agrees with λ.i. The current valuation val\textsubscript{c} is used to check whether the guard is satisfied as well as the condition of one of the cases in the function trans.function. The effects of this case are then evaluated, generating the new execution state σ'. Finally, the current location of the execution state of the execution part is set to the target location of the transition, i.e., trans.target.

**Example 10.** Let us consider a state σ = S.1 in Figure 4.8 and the execution part φ = (top, cars). The current location is Blinking. As shown in Figure 4.3, there are 2 outgoing transitions from this
We consider the trans = (\text{Blinking, toRed} (\text{delayY}), \text{true, function, Yellow}). Note that the first signal in the queue for type(\phi) is toRed (\text{delayY}), so it matches the trigger. The current valuation valc is \{\text{yellowD} = 20, \text{delayY} = 0\}. The function function has two cases, one for delayY > 0 and one for delayY \leq 0, with delayY being an input variable of the signal toRed. The condition is satisfied in the second case, therefore we have effect = \{\text{setTimers} (\{\text{yellowTimer}\})\}. Evaluation of this action yields a new element in the set of timers. The target state is given as \sigma' = S_2 in Figure 4.8.

![Figure 4.8: Example of the application of the match rule.](image)

The next rule is the external rule. This rule is applied if an outgoing transition of the current location of some part has an external signal as its trigger.

**Rule 3 (External rule).** Let \sigma = (m, s_{top}, \Phi) be the execution state of some model with some execution part \phi. The rule for generating a transition \sigma{(\phi, external, \lambda, o)} \sigma' is:
∀ φ ∈ Φ : ¬isInitial(s_{top}, φ)∧
∃ φ ∈ Φ : s_φ = getState(s_{top}, φ)∧
∃ trans ∈ s_φ.ω.l.outgoing :
trans.trigger.isExternal∧
∃ v_{concrete} ∈ trans.trigger.variables ∧
λ = (trans.trigger, v_{concrete})
val_c = s_φ.ω.v ∪ λ.v∧
trans.guard ∥ val_c ∥∧
∃ case ∈ trans.function.cases :
case.condition ∥ val_c ∥∧
(σ', o) = eval(case.effect, σ, φ, λ)∧
geState(σ', φ).ω.l = trans.target
σ (φ, external, λ, o) −→ σ'

In the external rule we check whether all parts are initialized and next we need to check whether the trigger trans.trigger is external. If this is the case we non-deterministically find a concrete valuation v_{concrete} that gives values to variables of the signal trans.trigger. Once this valuation is set, the following parts of the rule are the same as in case of the match rule. Note that the type of the event is \( t = \text{external} \).

**Example 11.** Consider state \( σ = S_1 \) in Figure 4.9. We have an execution part \( φ = ⟨\text{top}⟩ \), its execution state is in the location Both ready. There are two transitions possible from this location. Let us assume we choose the transition \( \text{trans} = (\text{Both ready}, \text{carsFirst}(\text{delayC}), \text{true}, \text{function}, \text{Stop walk}) \). Note that carsFirst is an external signal. We consider valuation \( v_{concrete} = \{ \text{delayC} = 150 \} \). The current valuation is \( val_c = \{ \text{carsD} = 120, \text{walkD} = 60, \text{delayC} = 150 \} \). The function function has 2 cases. Because \( \text{delayC} > 0 \) we have the following effect = \{ update(\{\text{carsD} = \text{delayC}\}),
\text{out}(\text{stopWalk()}\})\}. Their evaluation in the current context yields a new value for the attribute carsD, which is 150. Additionally, stopWalk() is added to the queue of the WalkLights module. The entire state is given as \( σ' = S_2 \) in Figure 4.9.

The **timeout rule** is responsible for simulating timeout events. The rule is applied if there is a transition triggered by a timer and the timer is currently set. Note that the proposed approach does
not consider any real time behavior, i.e., any timer can be set for any time, and a timeout event
may be used right away, only if there is a transition with the right trigger.

Rule 4 (Timeout rule). Let $\sigma = (m, s_{\text{stop}}, \Phi)$ be an execution state of some model with some
execution part $\phi$. The rule for generating a transition $\sigma \xrightarrow{(\phi, \text{timeout, timer}, o)} \sigma'$ is:

$$\forall \phi \in \Phi : \neg \text{isInitial}(s_{\text{stop}}, \phi) \land$$
$$\exists \phi \in \Phi : s_\phi = \text{getState}(s_{\text{stop}}, \phi) \land$$
$$\exists \text{trans} \in s_\phi.\omega.l.\text{outgoing} :$$
$$\text{timer} = \text{trans.trigger} \land$$
$$\text{timer} \in s_\phi.\omega.t \land$$
$$\text{val}_c = s_\phi.\omega.v \land$$
$$\text{trans.guard} \parallel \text{val}_c \parallel \land$$
$$\exists \text{case} \in \text{trans.function.cases} :$$
$$\text{case.condition} \parallel \text{val}_c \parallel \land$$
$$(\sigma', o) = \text{eval} (\text{case.effect}, \sigma, \phi, \text{timer}) \land$$
$$\text{getState}(\sigma', \phi).\omega.l = \text{trans.target}$$
$$\text{getState}(\sigma', \phi).\omega.t = s_\phi.\omega.t \setminus \{\text{timer}\}$$

$\sigma \xrightarrow{(\phi, \text{timeout, timer}, o)} \sigma'$
In this rule, as in the previous, we check initialization of all parts. In order to apply this rule we check whether the trigger of the transition \((\text{trans}.\text{trigger})\) is in the currently set timers, i.e, in \(\omega_t\). Because timers have no variables, we set the current valuation \(val_c\) to the current values of attributes as recorded in \(s_{\varphi}.\omega.\nu\). After that, the rule is similar to the external rule or the match rule.

**Example 12.** Consider an execution state \(\sigma = S_1\) in Figure 4.10. The details of the current execution state for the execution part \(\varphi = (\text{top}, \text{cars})\) is \(\omega_\varphi = (\text{Yellow}, \{\text{yellowD} = 20\}, \{\text{yellowTimer}\})\). The only outgoing transition from this state is as follows \(\text{trans} = (\text{Yellow}, \text{yellowTimer}, \text{true}, \text{function}, \text{Red})\) and the trigger of this transition is \(\text{yellowTimer}\). This timer is currently set, that is, \(\omega_\varphi.t = \{\text{yellowTimer}\}\) and obviously the condition \(\text{yellowTimer} \in \omega_\varphi.t\). The execution valuation contains only the attribute of \(\text{cars}\). The function \(\text{function}\) has one case, and we have \(\text{effect} = \{\text{out}([\text{redSet()}])\}\). The evaluation of the effects yields a new queue content for the Controller module, as can be seen in Figure 4.10.

![Figure 4.10: Example of the application of the timeout rule.](image)

The last rule to introduce is the *drop rule*, which takes care of signals on the queues that cannot be consumed by their respective parts. The rule is applied if the first signal in a queue has no matching trigger on a transition outgoing from the current location.
Rule 5 (Drop rule). Let $\sigma = (m, s_{top}, \Phi)$ be the execution state of some model. The following describes the rule for generating a transition $\sigma \xrightarrow{\phi, \text{drop}, \lambda, o} \sigma'$. 

\[
\forall \phi \in \Phi : \neg \text{isInitial}(s_{top}, \phi) \land \\
\exists \phi \in \Phi : s_\phi = \text{getState}(s_{top}, \phi) \land \\
q = m(\text{queuesMap}(\text{type}(\phi))) \land \\
\lambda = \text{dequeue}(q, \phi) \land \\
\text{val}_c = s_\phi.\omega.\nu \cup \lambda.\nu \land \\
\neg \exists \trans \in s_\phi.\omega.l.\text{outgoing} : \\
\lambda.i = \trans.\text{trigger} \land \\
\text{trans.guard} \parallel \text{val}_c \\
\]

$\sigma \xrightarrow{\phi, \text{drop}, \lambda, o} \sigma'$

The drop rule is complementary to the match rule. Its application yields only an update of the queue for the module, that is, the dropped signal ($\lambda$) is being removed. The execution state details remain the same for all parts, since no transition is fired and no effects are evaluated.

Example 13. Consider execution state $\sigma = S.1$ in Figure 4.11 and the execution part $\phi = (\text{top})$. The current location of this part is Start cars and the first element in the queue for its module redSet(). There is no outgoing transition from Start cars triggered by this signal. The application of the drop rule removes the signal from the queue, leaving all other elements unchanged.

4.3 Transforming UML-RT models to CFFSMs

The main goal of introducing CFFSMs is to represent UML-RT models. In this section we show how we can transform UML-RT models to CFFSMs. This transformation process has two phases: structural transformation and generation of functions from action code.

4.3.1 Structural transformation

As can be seen from the previous description the structure of UML-RT and of CFFSMs is very similar. Table 4.2 shows the transformation of structural elements.

There are several simplifications we introduced in CFFSMs:
<table>
<thead>
<tr>
<th>UML-RT element</th>
<th>CFFSMs element</th>
</tr>
</thead>
<tbody>
<tr>
<td>capsule</td>
<td>module</td>
</tr>
<tr>
<td>part (type)</td>
<td>part (type)</td>
</tr>
<tr>
<td>(base port, protocol, out event(variable))</td>
<td>out signal(out variable)</td>
</tr>
<tr>
<td>(conjugated port, protocol, in event(variable))</td>
<td></td>
</tr>
<tr>
<td>(base port, protocol, in event(variable))</td>
<td>in signal(in variable)</td>
</tr>
<tr>
<td>(conjugated port, protocol, out event(variable))</td>
<td></td>
</tr>
</tbody>
</table>
| (connector(base port, conjugated port, owner), protocol, out event) | connector(owner,  
out signal (base port, protocol, out event),  
owning part(base port),  
in signal (conjugated port, protocol, out event),  
owning part (conjugated port),  
variables) |
| (connector(base port, conjugated port, owner), protocol, in event) | connector(owner,  
out signal (conjugated port, protocol, in event),  
owning part(conjugated port),  
in signal (base port, protocol, in event),  
owning part (base port),  
variables) |

Table 4.2: Informal description of the mapping of the structural elements from UML-RT models to CFFSMs. Parameters of elements are given in parentheses.
we do not represent protocols and ports, thus the combination of port, protocol and event is
treated as a single signal. This means that each event in a protocol is mapped to as many
signals as there are ports that use this protocol in the original model.

- connectors in CFFSMs connect signals and sending or receiving parts, whereas connectors in
UML-RT connect ports. Therefore, for each connector in the original model, we will have
as many connectors in CFFSMs as there are signals in the protocol of the connected ports.
Additionally, we mapped the owners of connectors to the owning modules in CFFSMs.

- in the mapping we omit the mapping from variables in input signals to variables in output
variables. In UML-RT signals are associated with a single variable, so the mapping is obvious.

The mapping of state machine elements is given in Table 4.3. The CFFSMs state machines
are flat, therefore, the original UML-RT State Machines must be flattened. To realize this, a
transition chain, that is, a sequence of transitions at different levels of nesting is replaced with a
single transition. In the same way, group transitions are treated as being a transition chain with
segments added. The guard functions and functions are generated from code using the method
shown in Section 4.3.2.
### 4.3.2 Generation of functions from action code

The generation of functions is based on symbolic execution of action code. The action code is gathered from transitions and entry or exit actions on states - such code is present with code snippets. The order of code snippets follows the order in the original transition chains in the UML-RT models. In this section we describe the symbolic execution of code snippets and the transformation of its results.

Action code is present in UML-RT models as code snippets assigned to transitions or as entry or exit code from states of UML-RT State Machines. When executing a code snippet we will use attributes and input variables of transition triggers as parameters. In order to be able to transform code, we will assume that a symbolic execution engine produces the following results for each reachable path of execution:

**Definition 34 (Symbolic execution result)**

A symbolic execution result generated during symbolic execution of action code is a tuple \( \text{result} = (\text{resultConditions}, \text{resultValues}, \text{resultNew}, \text{resultDestroy}, \text{resultOutput}, \text{resultTimersSet}, \text{resultReturn}) \), where:

- \( \text{resultConditions} \) is a set of conditions (path constraints), that are formulas over attributes and inputs variables
- \( \text{resultValues} \) is a valuation of attributes, input variables and other local variables in code,
- \( \text{resultNew} \) is a set of instantiated parts,
- **resultDestroy** is a set of destroyed parts,
- **resultOutput** is a sequence of output signals with valuation of variables,
- **resultTimersSet** is a set of set timers,
- **resultReturn** is a return expression - this is a boolean expression that represents guard condition on a transition with guard action code.

We will show the execution rules of a symbolic execution engine for a subset of C++ action code, which provides us with the required results. Table 4.4 contains the rules for generating symbolic execution states in a symbolic execution tree (SET).

Table 4.4: Symbolic execution rules (for some symbolic execution state with PC, VAL, OUT, NEW, DESTROY, TIMERS, where PC are path constraints, VAL includes values of variables, OUT is a sequence of output signals, NEW and DESTROY are sets of parts to instantiate and to destroy and TIMERS is a set of timers. The operation \( \oplus \) adds a mapping for a variable, possibly overriding old value.

<table>
<thead>
<tr>
<th>Type of statement</th>
<th>Example</th>
<th>Resulting SET state (denoted with prime notation)</th>
</tr>
</thead>
</table>
| variable declaration | int k | $PC' = PC$
$VAL'(VAL \oplus k \rightarrow k_{\text{symbol}})$
$OUT' = OUT$
$NEW' = NEW$
$DESTROY' = DESTROY$
$TIMERS' = TIMERS$

| value assignment (current valuation is $v$) | $k = \text{expr}$ | $PC' = PC$
$VAL'(VAL \oplus k \rightarrow \text{expr}[v])$
$OUT' = OUT$
$NEW' = NEW$
$DESTROY' = DESTROY$
$TIMERS' = TIMERS$

| send statement | port.signal(6). send() | $PC' = PC$
$VAL' = VAL$
$OUT' = \{OUT, port.signal(6)\}
$NEW' = NEW$
$DESTROY' = DESTROY$
$TIMERS' = TIMERS$

*Continued on next page*
Table 4.4 – Continued from previous page

<table>
<thead>
<tr>
<th>Type of statement</th>
<th>Example</th>
<th>Resulting SET state (denoted with prime notation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>part instantiation</td>
<td>frame.incarnate (part)</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>part destruction</td>
<td>frame.destroy (part)</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>timer setting</td>
<td>timer.informIn (RTTimespec(10, 0))</td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
</tbody>
</table>
| if-else statement (current valuation is v) | if (cond){
  <statements>
} else {
  <statements>
} | ![Diagram](image4)                               |
| while-loop statement (current valuation is v0, K is the bound given by the user) | while (cond) {<statements>} | ![Diagram](image5)                               |

The symbolic execution of code uses the control flow graph and proceeds from line to line, including branching. Initially all attributes in a state machine have symbolic values assigned to
them. The result is a symbolic execution tree, which contains states defined as in Definition 34.

**Example 14.** Let consider the following code snippet:

```java
int k;
k = *rtdata;
if (k > 0) {
carsD = k;
}
walkManager.stopWalk().send();
```

Figure 4.12 shows the symbolic execution tree for the execution of the above code snippet. It starts with symbolic values for both attributes of the module: carsD and walkD. In the next line the new variable is added, which then is assigned the value of the input variable received with the trigger. Next, two cases are considered. If `input` is greater than zero, then the attribute `carsD` is updated, if not it remains unchanged. After that the signal `walkManager.stopWalk()` is sent.

![Symbolic Execution Tree](image)

Figure 4.12: An example of symbolic execution tree for action code.

The generation of the symbolic execution tree for code is the first step in creating functions. The second step is transforming the results into a function and a guard. This step is described below,
Definition 35 (Function generation)

Let \( \text{results} = \{\text{result}_1, \ldots, \text{result}_n\} \) be the set of leaves of a symbolic execution tree generated for a code snippet. Each leaf in this tree is defined as follows: \( \text{result}_i = (\text{resultConditions}_i, \text{resultValues}_i, \text{resultNew}_i, \text{resultDestroy}_i, \text{resultOutput}_i, \text{resultTimersSet}_i, \text{resultReturn}_i) \). The function generated from this code is \( \text{function} = \{\text{case}_1, \ldots, \text{case}_n\} \), where \( \text{case}_k \) is defined as \( \text{case}_k = (\text{condition}_k, \text{effect}_k) \) with condition defined as \( \text{condition}_k = \text{resultConditions}_k \) and \( \text{effect}_k = \{\text{update}(\text{resultValues}_k) \cup \text{new}(\text{resultNew}_k) \cup \text{destroy}(\text{resultDestroy}_k) \cup \text{out}(\text{resultOutput}_k) \cup \text{setTimers}(\text{resultTimersSet}_k)\} \). We will omit empty effects.

Definition 36 (Guard generation)

Let \( \text{results} = \{\text{result}_1, \ldots, \text{result}_n\} \) be the set of leaves of a symbolic execution tree generated for a code snippet. Each leaf in this tree is defined as follows: \( \text{result}_i = (\text{resultConditions}_i, \text{resultValues}_i, \text{resultNew}_i, \text{resultDestroy}_i, \text{resultOutput}_i, \text{resultTimersSet}_i, \text{resultReturn}_i) \). Guard expression generated from this code is defined as \( \text{guard} = (\text{resultConditions}_1 \Rightarrow \text{resultReturn}_1) \land \ldots \land (\text{resultConditions}_n \Rightarrow \text{resultReturn}_n) \).

Example 15. Consider the tree generated in Example 14. The function generated from this code is: \( \text{function} = \{\text{case}_1, \text{case}_2\} \) with \( \text{case}_1 = ((\text{input} > 0), \{\text{update}(\{\text{carsD} = \text{input}\}), \text{out}([\text{stopWalk()}])\}) \) and \( \text{case}_2 = ((\text{input} \leq 0), \{\text{out}([\text{stopWalk()}])\}) \).

4.4 Summary

In this chapter we present the formal notation we use to represent UML-RT models. The proposed formalisation is called Communicating Functional Finite State Machines (CFFSMS). First we present the structure of the CFFSMS model, which consists of nested, communicating, hierarchical modules, each having its own behavior. We also showed the semantics of CFFSMSs.

The semantics is a labeled transition system (LTS) with states recording the contents of queues and the execution state of each module. The rules that generate the transition relation in this LTS represent different kinds of change in the model:

- the default rule moves parts from their initial locations,
- the match rule processes elements from the queues,

- the external rule deals with signals provided by the environment,

- the timeout rule uses the timers,

- the drop rule removes unmatched signals from the queues.

We also showed how a UML-RT model is transformed to CFFSMs. First the structure is mapped and then the code is symbolically executed and summarized as conditional effects attached to transitions.

The transformation and execution based on the above rules is implemented as described in Chapter 7 (page 162).
Chapter 5

Types of abstractions for CFFSMs

In this chapter we define abstractions for CFFSMs. The main goal of abstractions is to simplify the execution semantics, that is, to reduce the number of execution states, making the models easier to analyze and verify. As shown in Figure 5.1 abstractions are defined by a set of rules that generate the transition relation in the execution LTS. Moreover, when a model is executed with abstraction rules, the structure of execution states might have to be adjusted. In this chapter we present changes to the execution rules and to the execution states to provide the abstract executions.

Figure 5.1: An overview of exploration modes.
Exploration in concrete execution uses the concrete execution rules presented in the previous chapter. During the abstract execution these rules are adjusted. For a given model \( \text{model} \) let us denote the LTS that is the result of concrete execution with \( \text{LTS}_{\text{model}} \) and the LTS that is a result of abstract execution with \( \hat{\text{LTS}}_{\text{model}} \). We use a notion of paths to compare the executions. In case of \( \hat{\text{LTS}}_{\text{model}} \) we assume that the definition of execution states might be different, but that we can still gather execution paths.

**Definition 37 (Execution path)**

Let \( \text{LTS} = (\Sigma, \Delta, \sigma_0, \Phi) \). A path in this LTS is defined as \( \pi = (\sigma_0, \sigma_1, \ldots) \), where \( \forall i \geq 0 : (\sigma_i, \nu, \sigma_{i+1}) \in \Delta \). All paths in this LTS are denoted as \( \Pi_{\text{LTS}} \). We assume that we can define abstract paths in an abstract LTS in the same way. We define \( \pi[i] \) to be the \( i \)th state on the path.

In order to compare different abstract executions we compare execution paths. We assume that for each kind of abstraction we have two functions \( \text{concrete}(\ldots) \) and \( \text{abstract}(\ldots) \). The first one transforms abstract execution paths to concrete execution paths. The opposite transformation, i.e., \( \text{abstract}(\ldots) \) takes a concrete path and returns an abstracted path. Those two operations enable comparison between abstract and concrete executions, which is based on relating the execution paths.

**Notation:**

We use \( \pi (\Pi) \) to denote a concrete path (a set of concrete paths), that is, an execution path in a concrete LTS, and \( \hat{\pi}^\circ (\hat{\Pi}^\circ) \) is an abstract path (a set of abstract paths) in an abstract LTS, with \( \circ \) indicating the type of abstraction. In the same way we denote abstract and concretize functions: \( \text{abstract}^\circ(\ldots) \) and \( \text{concrete}^\circ(\ldots) \).

In the sequel we assume that \( \text{concrete}^\circ(\ldots) \) and \( \text{abstract}^\circ(\ldots) \) form a Galois connection under set inclusion [44] as defined below. The condition of the connection requires that for sets of abstract execution paths and concrete execution paths we have that iff abstraction of concrete paths is a subset of abstract paths, then concrete paths are a subset of concretized abstract paths.

**Definition 38 (Galois connection)**

Let \( \Pi \) be a set of concrete execution paths and let \( \hat{\Pi}^\circ \) be a set of abstract paths. We have: \( \text{concrete}^\circ : \)}
$2^\hat{\Pi} \rightarrow 2^\Pi$ and $\text{abstract}^\circ : 2^\Pi \rightarrow 2^\hat{\Pi}$. We say that $(\text{concrete}^\circ, \text{abstract}^\circ)$ is a Galois connection iff:

$$\forall \Pi_i \in 2^\Pi \ : \ \forall \hat{\Pi}_i^\circ \in 2^\hat{\Pi} \ : \ \text{abstract}^\circ(\Pi_i) \subseteq \hat{\Pi}_i^\circ \Leftrightarrow \Pi_i \subseteq \text{concrete}^\circ(\hat{\Pi}_i^\circ)$$

Assuming that abstraction and concretization functions form the Galois connection, gives us some useful properties. For instance, if we have one function defined, we know that the other exists and we can provide it.

We distinguish three relationships between concrete and abstract executions of the same CFF-SMs model. The concrete execution is represented with $\text{LTS}_{\text{model}}$ and the abstract execution with $\hat{\text{LTS}}_{\text{model}}$. We have $\Pi$ to be the set of paths of $\text{LTS}_{\text{model}}$ and $\hat{\Pi}^\circ$ to be the paths of $\hat{\text{LTS}}$.

1. Exactness. This happens iff $\Pi = \text{concrete}^\circ(\hat{\Pi}^\circ)$ or iff $\hat{\Pi}^\circ = \text{abstract}^\circ(\Pi)$. This means that every path in a concrete $\text{LTS}$ is mapped to a concretized abstract path $\text{concrete}^\circ(\hat{\Pi}^\circ)$ or every path in an abstract $\hat{\text{LTS}}$ is mapped to an abstracted concrete path $\hat{\pi}^\circ$. Consequently, there are no extra paths introduced during abstraction process.

2. Overapproximation. This happens iff $\text{abstract}^\circ(\Pi) \subseteq \hat{\Pi}^\circ$. Because $\text{abstract}^\circ$ is a function, each path from $\text{LTS}$ has a matching path in $\hat{\Pi}^\circ$. It is possible that there might be some paths that exists in the abstract execution.

3. Underapproximation. This happens iff $\hat{\Pi}^\circ \subseteq \text{abstract}^\circ(\Pi)$. In this situation there might be a path in $\text{LTS}$, which has no matching path in $\text{concrete}^\circ(\hat{\Pi}^\circ)$.

In this work we consider three types of abstractions. They were selected to leverage the way models are represented, so to leverage the close relation between the structure of UML-RT and CFFSMs. The considered abstractions are:

- symbolic execution: this abstraction is used to deal with variables and branching of code.
  Symbolic execution is an exact abstraction for CFFSMs and is fully automatic, that is, no extra information is necessary. Because of this property we use symbolic execution to include all branches of execution, but we can also use it for other types of analyses. For instance, we can easily detect unreachable states. The other useful application is to use symbolic execution to generate test cases, because we can cover all branches.

- structural abstraction: this abstraction is used to deal with complex, hierarchical structures
of models, by abstracting out parts of a model. The structural abstraction is an overapproximation and it requires some extra information to run, i.e., a specification of the parts to be abstracted.

- state aggregation: this abstraction is particularly useful to deal with state machines that represent hierarchical and complex UML-RT State Machines by aggregating states at the same level of hierarchy. As in the case of the structural abstraction this abstraction is an overapproximation and it requires some extra information to run, i.e., states that are to be aggregated.

In this chapter we present the details of the presented abstractions. For each of them we introduce the structure of execution states, we show the execution rules and we demonstrate whether abstraction is exact, or whether it is an over- or underapproximation.

5.1 Symbolic execution

Symbolic execution is an analysis method introduced in the 70s for source code (see Section 2.2). We used this method in the previous chapter to obtain the symbolic representation of action code. In this chapter we leverage symbolic execution to operate on entire models. First we describe the structure of an execution state, then we show the execution rules and we show the mapping between concrete and symbolic states and paths.

5.1.1 Symbolic LTS

The main differences between the structure of LTS in concrete execution (see Definition 23) and the structure of LTS in symbolic execution are the elements that need to be stored to represent path constraints as well as elements that support similarity detection. We start by showing the structure of a symbolic LTS and then its internal elements.

A symbolic LTS (i.e., LTS in symbolic execution) has a structure similar to its non-symbolic counterpart.

**Definition 39** (Symbolic Labeled Transition System)

Let \( \text{model} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues}) \). The symbolic execution
of model is a symbolic labeled transition system $LTS^* = (\Sigma^*, \Delta^*, \sigma_0^*, \Phi^*)$, where:

- $\Sigma^*$ is a set of symbolic execution states (see Definition 40),

- $\Delta^*$ is a symbolic transition relation (very similar to Definition 30 with the rules defined in this section and with states from $\Sigma^*$),

- $\sigma_0^* \in \Sigma^*$ is an initial symbolic execution state (see Definition 31),

- $\Phi^*$ is a set of all execution parts in the model (the same as $\Phi$ in Definition 23).

A symbolic execution state, similarly to a concrete execution state, contains information about the contents of queues and information about symbolic states for parts. Additionally, each symbolic execution state stores the states it subsumes, that is, states that are similar to it. The definitions of execution queues, queues maps and execution signals are the same.

**Definition 40** (Symbolic execution state)

Let model = (modules, queues, queuesMap, topPart, topModule) and the symbolic execution of this model is $LTS^* = (\Sigma^*, \Delta^*, \sigma_0^*, \Phi^*)$. A symbolic execution state of model is a tuple $\sigma^* = (m^*, s_{top^*}, \Phi^*, \Sigma_{\approx})$, where:

- $m^*$ is a map the same as $m$ in Definition 25,

- $s_{top^*}$ is the symbolic execution state of the top level part and its inner parts and recursively their parts (see Definition 41),

- $\Phi^* \subseteq \Phi^*$ is a set of currently explored execution parts,

- $\Sigma_{\approx} \subseteq \Sigma^*$ is a set of states that are equivalent to this state (i.e., the states this state subsumes, see Section 5.1.2).

The structure of a symbolic execution state of an execution part is also very similar to its concrete version in Definition 28.

**Definition 41** (Symbolic execution state of a part)

Let module = (SM, parts, attributes, signals, connectors, timers, defaultValues) and let execution part
φ^s be of type module. The symbolic execution state \( s^* \) of this module is given as a tuple \((φ^s, ω^s, s_{parts}^*)\), where:

- \( φ^s \) is an execution part, which is the same as \( φ \) in Definition 28,

- \( ω^s \) contains symbolic state details (see Definition 42 below),

- \( s_{parts}^* : parts \rightarrow S^* \) maps parts of a module to their symbolic execution states \( S^* \) in the same way as \( s_{parts} \) in Definition 28.

We overload the operation \( get\text{State} \) to be applied to symbolic execution states.

The major difference between concrete and symbolic execution is in the definition of the state details of a part. Symbolic state details need to store path constraints and a mapping of symbolic variables to their original counterparts in a model. Symbolic variables are used during symbolic execution to represent the input variables received with external signals. In other words symbolic variables represent parameters of received input signals. Every time there is an external signal received, for each variable of this signal we introduce a symbolic variable that is not used anywhere else in the model. The necessity to store the mapping of the symbolic variables to the input variables (i.e., parameters) arise when checking for subsumption between states (see Definition 45 below).

**Definition 42** (Symbolic state details)

Let \( module = (SM, parts, attributes, signals, connectors, timers, defaultValues) \) and let \( SM = (locations, transitions, functions, guards, locationInit) \) be its state machine. Symbolic state details of execution are defined as \( ω^s = (l^s, v^s, t^s, pc, sv) \) or \( ω^s = ε \), where:

- \( l^s \in locations \) is the current location, which is the same as \( l \) in Definition 29,

- \( v^s \in attributes \) is a symbolic valuation of the attributes in a module,

- \( t^s \subseteq timers \) is a set of currently set timers, which is the same as \( t \) in Definition 29,

- \( pc \subseteq F \) is a set of conditions that represent path constraints with variables of formulas being symbolic variables,

- \( sv \) is a mapping of symbolic variables to the variables of a model (i.e., input variables of signals) they were created for. This mapping is a special kind of valuation in which terms are simply
variables.

- \( \varepsilon \) denotes an empty state.

Example 16. Figure 5.2 shows an example of a symbolic execution state. We present symbolic states similarly to concrete execution states. The first part of the rectangle contains queues. The second part lists the states subsumed by this one, which is empty in this case. Usually we omit this part if the set of states is empty. Finally, there are symbolic execution states for each part. Note, that the valuation for the variable \( \texttt{carsD} \) instead of some concrete value, is a symbolic variable \( \texttt{delayC.1} \), i.e., the variable created when the input signal is received and the value of its parameter is to be assigned to \( \texttt{carsD} \). We also have path constraints, which for the top level part indicate that the symbolic variable \( \texttt{delayC.1} \) must be greater than zero in order to be able to reach this state. The last piece of information is the mapping of symbolic variables in input variables. In this case \( \texttt{delayC.1} \) is mapped to the input variable \( \texttt{delayC} \) of the \texttt{carsFirst} signal.

![Figure 5.2: Example of a symbolic execution state.](image)

5.1.2 Subsumption relation

Subsumption occurs between states of execution and detects similar states. We do not explore subsumed states any more. We say that a state is subsumed if we can find another very similar state, already explored.

Definition 43 (Subsumption of execution state)

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \) be a model with symbolic execution
\(LTS^s = (\Sigma^s, \Delta^s, \sigma^s_0, \Phi^s)\) and two symbolic execution states \(\sigma^s = (m^s, s_{\text{top}}^s, \Phi^s, \Sigma^s)\) and \(\sigma^{s'} = (m^{s'}, s_{\text{top}}^{s'}, \Phi^{s'}, \Sigma^{s'})\). We say that \(\sigma^s\) subsumes \(\sigma^{s'}\), denoted as \(\sigma^s \prec \sigma^{s'}\) iff all conditions below are satisfied:

1. \(\bar{\Phi}^s = \bar{\Phi}^{s'}\) (explored parts are the same),
2. \(\forall\ queue \in \text{queues} : m^s(\text{queue}) = m^{s'}(\text{queue})\) (execution queues have the same contents),
3. \(s_{\text{top}}^s \equiv s_{\text{top}}^{s'}\) (execution states for the top level part of the model are the same, see Definition 44 below),
4. \(\sigma^{s'}\) has no outgoing transitions.

The similarity of two symbolic states for parts is based on similarity of their details and the similarity of execution states for their internal parts.

**Definition 44 (Similarity of execution states for parts)**

Let \(LTS^s = (\Sigma^s, \Delta^s, \sigma^s_0, \Phi^s)\) with two execution states for the top level part \(\phi^s: s_{\text{top}}^s = (\phi^s, \omega^s, s_{\text{parts}}^s)\) and \(s_{\text{top}}^{s'} = (\phi^{s'}, \omega^{s'}, s_{\text{parts}}^{s'})\). The two states are similar, that is, \(s_{\text{top}}^s \equiv s_{\text{top}}^{s'}\) iff \(\omega^s \equiv \omega^{s'} \land \forall p \in \text{type}(\phi^s).\text{parts} : s_{\text{parts}}^s(p) \equiv s_{\text{parts}}^{s'}(p)\).

The similarity between details of states is based on checking the current location, current values of attributes and path constraints. In case of the last two we require that conditions or terms are similar only up to input symbolic variables used, that is, we use mapping \(sv\). So we check the similarity between state details modulo symbolic variables used in formulas and terms.

**Definition 45 (Similarity of states details)**

Let \(LTS^s = (\Sigma^s, \Delta^s, \sigma^s_0, \Phi^s)\) be a symbolic LTS of \(\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})\). It contains two execution states that for the execution part \(\phi^s\) have the state details \(\omega^s = (l^s, v^s, t^s, pc, sv)\) and \(\omega^{s'} = (l^{s'}, v^{s'}, t^{s'}, pc', sv')\). These state details are similar, i.e., \(\omega^s \equiv \omega^{s'}\) iff all of the following conditions are satisfied:

1. \(l^s = l^{s'}\) (locations are the same),
2. \(\forall\ att \in \text{attributes} : v^s(\text{att}) \parallel sv\parallel = v^{s'}(\text{att}) \parallel sv'\parallel\) (values of attributes are the same after replacement of symbolic variables by their input parameters),
(3) \( pc \parallel sv \parallel = pc' \parallel sv' \parallel \), where \( pc \parallel sv \parallel = \{ c \parallel sv \parallel \mid c \in pc \} \) and similarly for \( pc' \) we have \( pc' \parallel sv' \parallel = \{ c' \parallel sv' \parallel \mid c' \in pc' \} \) and we require that both sets are the same (i.e., path constraints are the same modulo mapping of symbolic variables).

Note that we require that constraints are exactly matched. It is possible to use a weaker relation: \( pc \parallel sv \parallel \subseteq pc' \parallel sv' \parallel \). In this case all paths are explored, but some unreachable paths might be created. This is because we would have a path in which some of the path constraints are not continued along the entire path. If we have a state that subsumes another state and the subsumed state has more paths constraints, all the states that follow it are the state that follow the subsuming state and contain only a subset of path constraints.

5.1.3 Symbolic execution rules

The important part of the execution semantics presented in the previous chapter is the evaluation of functions. Here, we use the fact that this evaluation is the same for the symbolic execution. This is possible, because a concrete valuation used in the concrete execution is just a special case of valuations and we can use them in the same way.

The initialization of a LTS* is very similar to the one presented in Definition 32, because the initial state uses the initial values of all attributes given by the default valuation. The only adjustment is in the initialization of symbolic state details elements specific for the symbolic execution state, that is, path constraints and mapping of symbolic variables. Both of these are initialized to empty sets.

The transition relation in LTS* is given with rules. They are similar to the rules presented for the concrete execution in Section 4.2.3 and here we compare the concrete and symbolic versions of the execution rules.

We start with the symbolic version of default rule. We use the fact that there are no input variables associated with the default transition and the initial values of attributes are concrete. Consequently, no branching is possible and the definition of the default rule is exactly the same in both cases as shown in Table 5.1.

The matching rule for both types of execution is presented in Table 5.2. First, we need to make sure that the state we are exploring is not subsumed. The next part of the rule is very similar to that of the concrete version and differs only in the evaluation of guard and case conditions. During
CHAPTER 5. TYPES OF ABSTRACTIONS FOR CFFSMS

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default rule</td>
<td>Default rule</td>
</tr>
</tbody>
</table>
| \( \exists \phi \in \bar{\Phi} : \text{isInitial}(s_{\text{top}}, \phi) \land 
  \text{isInitialTop}(s_{\text{top}}, \phi) \land 
  (\sigma', o) = \text{fireDefault}(\phi) \) \rightarrow \sigma' | \( \exists \phi^s \in \bar{\Phi}^s : \text{isInitial}(s_{\text{top}}^s, \phi^s) \land 
  \text{isInitialTop}(s_{\text{top}}^s, \phi^s) \land 
  (\sigma^{s'}, o) = \text{fireDefault}(\phi) \) \rightarrow \sigma^{s'} |

Table 5.1: Comparison of the default rule between concrete and symbolic execution.

Concrete execution they could be evaluated to true or false, but during a symbolic execution it is not always possible. However, it is possible to check whether the evaluated guard (or case condition) is satisfiable, that is, if there exists a concrete valuation of symbolic variables that makes them true or false. This is why we define satisfiability checking for formulas \( \text{isSat} \).

**Definition 46** (Satisfaction checking)

Let \( f \in \mathcal{F}[V] \) with variables \( \text{vars}(f) \subseteq V \). We define satisfaction as: \( \text{isSat}(f) = (\exists \text{val} \in \bar{V} : f \parallel \text{val}) \). So the formula is satisfiable iff there is a valuation of its variables that makes the formula true.

Additionally, because a valuation of input variables received with an input signal \( \lambda \) may contain symbolic variables, we have to add mappings of those symbolic variables to \( sv \) of the currently explored execution part. In order to be able to find the necessary mapping we use the following function. For each symbolic variable used in some execution part, we find the input variable that this symbolic variable is mapped to.

**Definition 47** (Getting symbolic variables mapping)

Let \( V \) be a set of variables and let \( \sigma^s = (m^s, s_{\text{top}}^s, \bar{\Phi}^s, \Sigma^s_<) \) be an execution state. We define a function \( \text{getSVs}(s_{\text{top}}^s, V) \) to return a map of symbolic variables to input variables: \( \{v \mapsto v_i \mid v \in V \wedge \exists \phi^s \in \bar{\Phi}^s : \text{getState}(s_{\text{top}}^s, \phi^s).\omega^s.\text{sv}(v) = v_i\} \).
<table>
<thead>
<tr>
<th>Concrete</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match rule</td>
<td>Rule 6.</td>
</tr>
</tbody>
</table>
| ∀ φ ∈ \( \Phi \) : \neg isInitial(s_{top}, φ) ∧ \exists φ ∈ \( \Phi \) : s_{φ} = getState(s_{top}, φ) ∧ q = m(queuesMap(type(φ))) ∧ \lambda = dequeue(q, φ) ∧ m'(queuesMap(type(φ))) = q ∧ val_c = s_{φ}.\omega.v \cup \lambda.v ∧ \exists trans ∈ s_{φ}.\omega.l.outgoing : \lambda.i = trans.trigger ∧ trans.guard ||val_c|| ∧ \exists case ∈ trans.function.cases : case.condition ||val_c|| ∧ \((\sigma^{*'}, o) = eval\) (case.effect, σ, φ, val_c) ∧ getState(σ' , φ).\omega.l = trans.target ∧ getState(σ' , φ).\omega.pc = s_{φ}.\omega.pc \cup \{trans.guard ||val_c||\} \cup \{case.condition ||val_c||\} ∧ getState(σ' , φ).\omega.sv = s_{φ}.\omega.sv \cup \{getSVs(s_{top}, vars(\lambda.v))\} ∧ isSatisfiablePC(\sigma^{*'}) \sigma^{*'} \xrightarrow{(\phi, match, \lambda, o)} σ^{*'}

Table 5.2: Comparison of the match rule between concrete and symbolic execution.
Finally, we require that path constraints from all symbolic state details are not contradicting. So the conjunction of all path constraints must be satisfiable.

**Definition 48** (Combined PC checking)

Let \( \sigma^s = (m^s, s_{top}^s, \Phi^s, \Sigma^s) \) be an execution state. We say that its combined path constraints are consistent if \( \text{isSatisfiablePC}(\sigma^s) \) iff the following formula

\[
f_{pc} = \forall \phi^s \in \Phi^s : \bigwedge_{cst \in \text{getState}(s_{top}^s, \phi^s), \omega^s_{pc}} \text{cst}
\]

is satisfiable. This means there exists a valuation of variables in \( f_{pc} \) that makes \( f_{pc} \) true, so \( \exists \text{val} \in \text{vars}(f_{pc}) : f_{pc} \parallel \text{val} \). We also use an extended version of this check \( \text{isSatisfiablePC}(\sigma^s, f) \), which checks satisfiability of the formula \( f_{pc} \land f \).

**Example 17.** Figure 5.3 shows an example of the application of the matching rule (model from Example 6). Let us consider execution part \( <\text{top,cars}> \). In the queue for its module there is one signal \text{toRed}. The value of its variable is a symbolic variable \text{delayW1} (stored in the symbolic variable mapping \text{SV} of the part \text{top}). This signal triggers a transition between \text{Blinking} and \text{Yellow} states in the \text{CarLights} state machine. There are two cases in the function associated with this transition. The evaluation of the condition of the first case yields \( \text{delayW1} > 0 \) and of the second \( \text{delayW1} \leq 0 \). Both of these conditions are satisfiable and they are added to path constraints of \text{cars} part. However, when the conjunction of all path constraints is checked for satisfiability, only the first one is satisfiable, because there is already a path constraint \( \text{delayW1} > 0 \) present in the \text{top} part. This means only one state is generated, as shown in Figure 5.3. The receiving part, i.e., \text{cars} adds the mapping of the received symbolic variable according to the mapping stored in \text{sv} of the top part. This mapping of the received symbolic variable \text{delayW1} is already given in the \text{top} part.

The external rule for symbolic execution is compared with its concrete counterpart in Table 5.3. As in the case of the concrete version of the external rule, the valuation of input variables assigned to the triggering signal must be created. In this case the valuation consists of symbolic variables. In order to do so we use the following function.

**Definition 49** (Create symbolic variables)

Let \( V \) be the set of variables of some input signal. The operation \( \text{val} = \text{createExtVal}(V) \) returns a valuation of those variables such that \( \text{val} \in \bar{V} \). The created valuation contains only fresh symbolic
### Table 5.3: Comparison of the external rule in symbolic and concrete execution.

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>External rule</strong></td>
<td><strong>Rule 7.</strong></td>
</tr>
<tr>
<td>∀ φ ∈ Φ : ¬isInitial(s\textsubscript{top}, φ)∧</td>
<td>∀ φ\textsuperscript{s} ∈ Φ\textsuperscript{s} : ¬isInitial(s\textsubscript{top}, φ)∧</td>
</tr>
<tr>
<td>∃ φ ∈ Φ : sφ = getState(s\textsubscript{top}, φ)∧</td>
<td>∃ φ\textsuperscript{s} = getState(s\textsubscript{top}, φ)∧</td>
</tr>
<tr>
<td>∃ trans ∈ sφ.ω.l.outgoing :</td>
<td></td>
</tr>
<tr>
<td>trans.trigger.isExternal∧</td>
<td></td>
</tr>
<tr>
<td>∃ v\textsubscript{concrete} ∈ trans.trigger.variables ∧</td>
<td>v\textsubscript{symbolic} = createExtVal(trigger.V)∧</td>
</tr>
<tr>
<td>λ = (trans.trigger, v\textsubscript{concrete})</td>
<td>λ = (trans.trigger, v\textsubscript{symbolic})</td>
</tr>
<tr>
<td>val\textsubscript{c} = sφ.ω.v ∪ λ.v∧</td>
<td>val\textsubscript{c} = s\textsuperscript{s}φ.ω.v ∪ λ.v∧</td>
</tr>
<tr>
<td>trans.guard ∥ val\textsubscript{c}∥∧</td>
<td>isSat(trans.guard ∥ val\textsubscript{c})∥∧</td>
</tr>
<tr>
<td>∃ case ∈ trans.function.cases :</td>
<td>∃ case ∈ trans.function.cases :</td>
</tr>
<tr>
<td>case.condition ∥ val\textsubscript{c}∥∧</td>
<td>isSat(case.condition ∥ val\textsubscript{c})∥∧</td>
</tr>
<tr>
<td>(σ′, o) =</td>
<td>(σ\textsuperscript{s′}, o) =</td>
</tr>
<tr>
<td>eval(case.effect, σ, φ, λ)∧</td>
<td>eval(case.effect, σ, φ, λ)∧</td>
</tr>
<tr>
<td>getState(σ′, φ),ω.l =</td>
<td>getState(σ\textsuperscript{s′}, φ\textsuperscript{s}),ω\textsuperscript{s}.l\textsuperscript{s} =</td>
</tr>
<tr>
<td>trans.target∧</td>
<td>trans.target∧</td>
</tr>
<tr>
<td>getState(σ\textsuperscript{s′}, φ\textsuperscript{s}),ω\textsuperscript{s}.pc =</td>
<td>getState(σ\textsuperscript{s′}, φ\textsuperscript{s}),ω\textsuperscript{s}.pc =</td>
</tr>
<tr>
<td>sφ,ω\textsuperscript{s}.pc ∪</td>
<td>s\textsuperscript{s}φ,ω\textsuperscript{s}.pc ∪</td>
</tr>
<tr>
<td>{trans.guard ∥ val\textsubscript{c}} ∪</td>
<td>{trans.guard ∥ val\textsubscript{c}} ∪</td>
</tr>
<tr>
<td>{case.condition ∥ val\textsubscript{c}} ∪</td>
<td>{case.condition ∥ val\textsubscript{c}} ∪</td>
</tr>
<tr>
<td>getState(σ\textsuperscript{s′}, φ),ω\textsuperscript{s}.sv =</td>
<td>getState(σ\textsuperscript{s′}, φ),ω\textsuperscript{s}.sv =</td>
</tr>
<tr>
<td>s\textsuperscript{s}φ,ω\textsuperscript{s}.sv ∪</td>
<td>s\textsuperscript{s}φ,ω\textsuperscript{s}.sv ∪</td>
</tr>
<tr>
<td>{reverse(v\textsubscript{symbolic})}∥∧</td>
<td>{reverse(v\textsubscript{symbolic})}∥∧</td>
</tr>
<tr>
<td>isSatisfiablePC(σ\textsuperscript{s′})</td>
<td>isSatisfiablePC(σ\textsuperscript{s′})</td>
</tr>
</tbody>
</table>

σ (φ, external, λ, o) \[\rightarrow\] σ′

σ\textsuperscript{s} (φ, external, λ, o) \[\rightarrow\] σ\textsuperscript{s′}
variables as terms in $\text{val}(v)$ for all $v \in V$, that is, variables that have never been used in the model. These symbolic variables are of the same type as the original variables.

Similarly to the match rule, guard and case conditions are checked for satisfiability and then added to the set of path constraints. Additionally, we add the mapping of the new symbolic variables, obtained by reversing the newly created valuation $v_{\text{symbolic}}$ with the reverse operation. We are allowed to do this, because the valuation $v_{\text{symbolic}}$ is a one-to-one mapping.

**Definition 50** (Reverse valuation)

Let $V$ be a set of variables and let $\text{val} \in \vec{V}$ such that all terms are variables: $\forall v \in V : \text{val}(v) \in V$. We define reverse($\text{val}$) to be a mapping \{symbolicVariable $\mapsto v \mid v \in V \land \text{val}(v) = \text{symbolicVariable}\}.

This means that the variable that serves as value becomes the variable that is mapped.

**Example 18.** Figure 5.4 shows an example of the application of the external rule (for the model introduced in Example 6). The exploration is done for the top part and the transition to Stop walk. There are two states generated, one for each case of the function associated with the transition. In the first case the condition is that the symbolic input variable $\text{delayC}_1$ must be positive. This new
symbolic variable in state $S_2$ is used to update the value of $\text{carsD}$ and it is used in the mapping $\text{delayC}_1 \rightarrow \text{delayC}$. The case condition $\text{delayC}_1 > 0$ is added to path constraints of the top part. In the second case, the attribute $\text{carsD}$ is not updated and path constraints of the top part are set to $\text{delayC}_2 \leq 0$.

The comparison of the timeout rule in concrete and symbolic executions is shown in Table 5.4. The timeout rule requires changes similar to the match rule, which includes the check for satisfiability of the guard and the case condition, adding path constraints and a check for the satisfiability of the combined constraints.

Table 5.5 compares concrete and symbolic version of the drop rule. Because there is no transition fired, the only change is the removal of the first element from the queue.

**Example 19.** Figure 5.5 shows an example of the application of the timeout rule to the model introduced in Example 6. The part that is being considered is $\text{cars}$, the timer is $\text{yellowT}$ and the transition is from $\text{Yellow}$ to $\text{Red}$. In the resulting state $S_2$ the timer has been removed from the set timers. Note that, because there are no guards and case conditions, the path constraints remain the same. Figure 5.6 shows an example of the application of the drop rule. Let us assume that there is
### Rule 8.

\[
\neg \exists \sigma^{s'} \in \Sigma^s : \sigma^{s'} < \sigma^s \land
\forall \phi^s \in \Phi^s : \neg \text{isInitial}(s_{top}^s, \phi^s) \land
\exists \phi^s \in \Phi^s : s^s = \text{getState}(s_{top}^s, \phi^s) \land
\exists \text{trans} \in s^s.\omega.l.\text{outgoing} : 
\begin{align*}
&\text{timer} = \text{trans}.\text{trigger} \land \\
&t = \text{getState}(\text{trans}.\omega.l, \text{trans}.\text{target}) \land \\
&\text{val}_c = \text{getState}(\text{trans}.\phi, \text{trans}.\omega.l) \land \\
&\text{trans}.\text{guard} || \text{val}_c || \land \\
&\exists \text{case} \in \text{trans}.\text{function}.\text{cases} : \\
&\text{isSat}(\text{case}.\text{condition} || \text{val}_c ||) \land \\
&\text{eval}(\text{case}.\text{effect}, \sigma^s, \phi^s, \text{trans}.\text{guard} || \text{val}_c ||) \land \\
&\text{getState}(\sigma^{s'}, \phi^s).\omega.l = \\
&\text{trans}.\text{target} \land \\
&\text{getState}(\sigma^{s'}, \phi^s).\omega.t = s^s.\omega.t \\
&\{\text{timer}\} \land \\
&\{\text{trans}.\text{guard} || \text{val}_c ||\} \cup \\
&\{\text{case}.\text{condition} || \text{val}_c ||\} \land \\
&\text{isSatisfiablePC}(\sigma^{s'})
\end{align*}
\]

\[
\sigma^s (\phi, \text{timeout}, \text{timer}, o) \xrightarrow{} \sigma^{s'}
\]

### Table 5.4: Comparison of the timeout rules in symbolic and concrete execution.
### Concrete

\[
\forall \phi \in \bar{\Phi} : 
\neg \text{isInitial}(s_{\text{top}}, \phi) \land \\
\exists \phi \in \bar{\Phi} : 
\phi = \text{getState}(s_{\text{top}}, \phi) \land \\
q = m(\text{queuesMap}(\text{type}(\phi))) \land \\
\lambda = \text{dequeue}(q, \phi) \land \\
val_c = s_\phi.\omega \cup \lambda.\nu \land \\
\neg \exists \text{trans} \in s_\phi.\omega.\lambda.\text{outgoing} : \\
\lambda.\iota = \text{trans}.\text{trigger} \land \\
\text{trans}.\text{guard} \parallel val_c 
\]

\[
\sigma (\phi, \text{drop}, \lambda, o) \quad \xrightarrow{\mathcal{R}} \quad \sigma' 
\]

### Symbolic

\[
\neg \exists \sigma^{st} \in \Sigma^{st} : 
\sigma^{st} \prec \sigma^s \land \\
\neg \exists \sigma^{st} \in \bar{\Phi} \leftarrow \bar{\Phi} : 
\neg \text{isInitial}(s_{\text{top}}^s, \phi^s) \land \\
\exists \phi^s \in \bar{\Phi}^s : 
\phi^s = \text{getState}(s_{\text{top}}^s, \phi^s) \land \\
q = m(\text{queuesMap}(\text{type}(\phi^s))) \land \\
\lambda = \text{dequeue}(q, \phi^s) \land \\
val_c = s_\phi^s.\omega^s \cup \lambda.\nu^s \land \\
\neg \exists \text{trans} \in s_\phi^s.\omega^s.\lambda^s.\text{outgoing} : \\
\lambda.\iota = \text{trans}.\text{trigger} \land \\
\text{isSat}(\text{trans}.\text{guard} \parallel val_c) \land \\
\text{isSatisfiablePC} \\
\sigma^s (\phi, \text{trans}.\text{guard} \parallel val_c) \quad \xrightarrow{\mathcal{R}} \quad \sigma^{st} 
\]

**Rule 9.**

\[
\neg \exists \sigma^{st} \in \Sigma^{st} : 
\sigma^{st} \prec \sigma^s \land \\
\forall \phi^s \in \bar{\Phi}^s : 
\neg \text{isInitial}(s_{\text{top}}^s, \phi^s) \land \\
\exists \phi^s \in \bar{\Phi}^s : 
\phi^s = \text{getState}(s_{\text{top}}^s, \phi^s) \land \\
q = m(\text{queuesMap}(\text{type}(\phi^s))) \land \\
\lambda = \text{dequeue}(q, \phi^s) \land \\
val_c = s_\phi^s.\omega^s \cup \lambda.\nu^s \land \\
\neg \exists \text{trans} \in s_\phi^s.\omega^s.\lambda^s.\text{outgoing} : \\
\lambda.\iota = \text{trans}.\text{trigger} \land \\
\text{isSat}(\text{trans}.\text{guard} \parallel val_c) \land \\
\text{isSatisfiablePC} \\
\sigma^s (\phi, \text{trans}.\text{guard} \parallel val_c) \quad \xrightarrow{\mathcal{R}} \quad \sigma^{st} 
\]

**Table 5.5:** Comparison of the drop rules in symbolic and concrete execution.
a guard \( \text{delayY} = 0 \) on the transition between Green and Yellow states in the CarLights state machine (compare with Figure 4.3). The part we are considering is cars. The signal in the queue is toRed with a value of its variable being a symbolic variable \( \text{delayW}_1 \). The guard is satisfiable, because 

\[
\text{delayY} = 0 \lor \{ \text{delayY} \mapsto \text{delayW}_1 \} \lor \text{delayW}_1 = 0
\]

However, this condition is in contradiction with existing path constraints, which state that \( \text{delayW}_1 > 0 \). Therefore, there is no transition that can satisfy the guard and path constraints and we have to drop a signal.

Finally, we need a rule to build the subsumption relation. This rule does not have a counterpart in the concrete rules and is defined as follows. In the rule the subsumption of a state is checked and then the state is added to the \( \Sigma_\prec \) of the subsuming state.

**Rule 10** (Subsumption rule). Let \( \sigma^s = (m^s, s_{\text{stop}}^s, \Phi^s, \Sigma^s_\prec) \) be the execution state. The following is the rule for generating a transition \( \sigma^s \xrightarrow{\text{subsumes}} \sigma^{s'} \).
Figure 5.6: Example of an application of the drop rule (assume guard delayY == 0 on the transition between Green and Yellow).

Example 20. Figure 5.7 shows an example of the application of the subsumption rule. Note that the two presented states have the same queue contents. The locations of state machines are the same for each part. If we use the mapping sv as the replacement for variables in the path constraints and values of attributes they also become the same. There are no outgoing transitions from $S_2$, so $S_1$ subsumes $S_2$. 
CHAPTER 5. TYPES OF ABSTRACTIONS FOR CFFSMS

Figure 5.7: An example of application of the subsumption rule.
5.1.4 Relationship between symbolic and concrete execution of CFFSMS models

At the beginning of this chapter we indicated that symbolic execution is an exact type of abstraction. We demonstrate below that this is the case. We start by defining the notion of symbolic execution paths in a symbolic LTS. Next, we introduce the \textit{concrete} operation on symbolic execution paths. Finally, we show that concretized paths are exactly the same as concrete paths. We initially assume that the subsumption is not checked.

The symbolic execution path is a counterpart of its concrete version.

\textbf{Definition 51} (Symbolic execution path)
Let $\text{LTS}^s = (\Sigma^s, \Delta^s, \sigma^s_0, \Phi^s)$. A path in a symbolic LTS is defined as $\hat{\pi}^s = (\sigma^s_0, \sigma^s_1, ...)$, where $\forall i \geq 0 : (\sigma^s_i, \nu, \sigma^s_{i+1}) \in \Delta^s$. We will denote all paths in $\text{LTS}^s$ with $\hat{\Pi}^s$.

Next, we show that in each state we have only symbolic variables as variables in formulas and terms used to represent values of attributes and path constraints.

\textbf{Lemma 2.} In each symbolic execution state variables that are in path constraints and in values of attributes are symbolic variables.

\textit{Proof.} According to the definition of functions (Definition 21) we know that variables used in conditions and in updates of attributes are only attributes or input variables. Attributes initially get concrete values, so during any update they can either get some concrete value (from the initial value of the attribute) or symbolic variable (from input variables). Path constraints are built out of guards and conditions and these also may only contain attributes and input variables replaced by concrete values or by symbolic variables.

Based on the above lemma, if we provide concrete values for all symbolic variables used in a symbolic execution path, we can eliminate all variables from valuations and constraints present on the path. Let us define an operation that uses this fact and makes a concrete version of a symbolic execution path, given the valuation of symbolic variables. We require that such valuation must satisfy all path constraints on the path. We know that a satisfying valuation exists because path constrains must be satisfiable (see the symbolic execution rules). In the operation we replace all symbolic variables encountered in any execution state (in terms used to represent attributes and input variables in queues) with a value given by the selected valuation.
Definition 52 (Concretization of execution path)
Let $\hat{\pi}^s = (\sigma_0^s, \sigma_1^s, \ldots)$ be a symbolic execution path of some $LTS^s$ and let $val$ be the valuation of all symbolic variables on this path, such that they satisfy the path constraints. The operation $\text{concrete}^s(\hat{\pi}^s, val)$ returns an execution path $\pi = (\sigma_0, \sigma_1, \ldots)$ such that $\forall i \geq 0 : \sigma_i = (m_i, s_{top_i}, \Phi)$ is defined as follows:

- $m_i$ contains all the execution queues of $m_i^s$, but whenever there is a term $t$ used to represent an input variable value it is replaced with $t \parallel val$,
- $s_{top_i}$ contains the state details $\omega = (l, v, t)$ which are taken from symbolic state details $\sigma_i^s.s_{top^s}.\omega^s = (l^s, v^s, t^s, pc, sv)$ with location $l = l^s$, valuation of attributes $v$ is $v^s$ where symbolic variables replaced by $val$ and timers are $t = t^s$. The same is applied to internal parts in $\sigma_i^s.s_{top^s}.s_{parts}^s$ and then recursively to their parts,
- $\Phi$ is the same as $\sigma_i^s.\Phi^s$.

Based on the above operation we can define an operation that provides all concrete paths from a symbolic execution path. This means we use all possible valuations of input variables.

Definition 53 (General concretization of symbolic execution path)
Let $\hat{\pi}^s = (\sigma_0^s, \sigma_1^s, \ldots)$ be a symbolic execution path of some $LTS^s$ and let $VS$ be a set of all symbolic variables in $\hat{\pi}^s$. The operation $\text{concrete}^s(\hat{\pi}^s)$ returns a set of all possible execution paths $\{\pi | \exists val \in VS : \pi = \text{concrete}^s(\hat{\pi}^s, val)\}$. We extend the above operation to sets of paths, that is, if $\hat{\Pi}^s$ is a set of symbolic execution paths we have $\text{concrete}^s(\hat{\Pi}^s) = \{\pi | \pi \in \text{concrete}^s(\hat{\Pi}^s) \land \hat{\pi}^s \in \hat{\Pi}^s\}$.

Note that defining $\text{concrete}^s$ is enough to define $\text{abstract}^s$ such that the functions form a Galois connection.

Now let us show that the paths obtained by concretization of all symbolic execution paths are the same paths that can be obtained by concrete execution of the same model.

Theorem 1. Let $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ and let its concrete execution LTS be given by $LTS$ and its symbolic execution LTS be given $LTS^s$. Let $\Pi$ represent all the paths in $LTS$ and let $\hat{\Pi}^s$ be all the paths in $LTS^s$. We have $\Pi = \text{concrete}^s(\hat{\Pi}^s)$. 
Proof sketch. We prove the theorem using the induction on the length of paths.

Base case. The base case is for paths of length 1. Note that the initial state in each path is the initial state of the execution. This execution state is created in the same way in case of both concrete and symbolic execution. The concretization operation removes path constraints and the mapping of symbolic variables in state details and makes the initial state the same as in the concrete execution. Thus, paths of the length one are the same.

Inductive step. We assume that for paths of the length n we have $\Pi = \text{concrete}^*(\hat{\Pi}^s)$. Let us show that the same holds for paths of the length $n + 1$. The illustration of the proof is given in Figure 5.8. We create a path of the length $n + 1$ by applying one of the rules to both concrete paths $\pi \in \Pi$ and symbolic paths $\hat{\pi}^s \in \hat{\Pi}^s$. Based on the inductive assumption we have $\pi = \text{concrete}^*(\hat{\pi}^s, \text{val})$ with val being some valuation of symbolic variables. In order to demonstrate the same relation holds for paths of the length $n + 1$ we will show that $\pi' = \text{concrete}^*(\hat{\pi}^s', \text{val}')$ for some val'. In order to do so, we will show that if a rule is applied to a concrete path, it will be also applicable to a symbolic path as well as the other way. We will show that states resulting from application of the rules are the same, i.e., a concrete execution state is the same as the concretized symbolic execution execution state. Since $\pi' = \text{concrete}^*(\hat{\pi}^s', \text{val}')$ holds for arbitrary paths we may extend the results and say that $\Pi = \text{concrete}^*(\hat{\Pi}^s)$ for paths of length $n + 1$. Let us now consider each rule:

- the default rule

(⇒ direction): Let us assume we can apply the default rule to the last state of $\pi$. The default rule will be also applied to the last state of the symbolic path, because if there exists uninitialized execution part in concretized symbolic state, this path is also uninitialized in the
original symbolic state. Because the rule is exactly the same for both executions and changes in target states are the same the concretization of the target symbolic state will result in the target concrete state. Consequently we have \( \pi' = \text{concrete}^s(\hat{\pi}^s, \text{val}') \) with \( \text{val}' = \text{val} \), where \( \text{val} \) is the valuation used to concretize a path of the length \( n \).

(\( \Leftarrow \) direction): Let us assume that the default rule is applied to the last state of \( \hat{\pi}^s \). The default rule is also applied to all concrete states obtained by concretization of this symbolic state, because these concrete execution states must have the same execution part uninitialized. As in the other direction the concretization of the created symbolic state with any valuation gives us the state which is the same as the concrete execution state after the default rule application. So for any \( \text{val} \) used in the path of the length \( N \) we have \( \pi' = \text{concrete}^s(\hat{\pi}^s, \text{val}) \).

- the match rule

(\( \Rightarrow \) direction): Let us assume we apply the concrete version of a match rule (Rule 2) to the last state in \( \pi \) resulting in \( \pi' \). The match rule is also applicable to the last symbolic execution state of \( \hat{\pi}^s \). This is because the matching signal must be in the respective queue in both states, the outgoing transition triggered by this signal must be enabled in both cases, and since guard and condition cases are satisfied, this means they are satisfiable. The newly generated symbolic state has the matching signal removed from the queue (as has the new concrete state), it has the new target location (as has the new concrete state), unchanged set of timers and the same effects are evaluated (as has the new concrete state). The only difference is that the received signal may contain symbolic variables, which may appear in the valuation of attributes and of sent variables. However, note that if this is the case these symbolic variables must have appeared earlier in the execution (since no new symbolic variables are introduced) and must have been used during concretization of the original \( \hat{\pi}^s \). If the same values of those variables are used in concretization of \( \hat{\pi}^s' \), the last state of this concrete path and the last state of \( \pi' \) are the same, and both paths \( \pi' \) and \( \text{concrete}^s(\hat{\pi}^s, \text{val}) \) are the same.

(\( \Leftarrow \) direction): Let us assume that we apply the symbolic match rule to the last state of \( \hat{\pi}^s \). If we use any valuation to perform concretization of this path, we obtain a concrete execution state that also has the match rule enabled. This is because concretization keeps the contents of the queues and current locations of all execution parts. Moreover, if the guard and case
condition are satisfiable in the symbolic state, they will be satisfied in the concretized state, because this is the condition of the valuation \( val \). As showed previously, the resulting state of concrete rule application is the same as the state resulting from the application of the symbolic rule.

- the external rule

\( \Rightarrow \) direction: Let us assume that the concrete version of the external rule (Rule 3) has been applied to the last state of \( \pi \). We assume that the valuation \( v_{\text{concrete}} \) was used to assign concrete values to possible input variables of an external signal. Now let us assume that the symbolic version of the rule (Rule 7) is applied to the last state of \( \hat{\pi}^s \). The rule is applicable, because the location is the same in both cases, and the outgoing transition with the external signal will be enabled in both execution states. The guard and case conditions are satisfied for the concrete case, so they are satisfiable as required by the symbolic rule. The resulting symbolic execution state has the same target location and the same effects are evaluated. The resulting symbolic path \( \hat{\pi}^s \) contains variables defined in valuation \( \text{val} \) and additionally symbolic variables \( SV \) created for external input variables from the last transition. Let us assume that the symbolic variable mapping containing these variables is \( sv \). We will define the new valuation used in concretization as \( \text{val}_{\text{external}} = \{v \mapsto c \mid v \in SV \land c = v_{\text{concrete}}(sv(v))\} \).

We take each symbolic variable, find its original input variable and use its value from \( v_{\text{concrete}} \). This valuation combined with the original valuation gives us \( \text{val}' = \text{val} \cup \text{val}_{\text{external}} \) and is applied to the last generated symbolic execution state during concretization. Because both rules are similar in creating the common elements of execution states, it is easy to see that the concretized state is the same as the new concrete execution state.

\( \Leftarrow \) direction: Let us assume we applied the external symbolic rule to the last state of \( \pi^s \). If there are variables assigned to the external signal they get new symbolic variables. Let us generate all possible concrete values for these new variables, which do not violate paths constraints in the created target symbolic execution state. If we use these values to create valuations of input variables \( v_{\text{concrete}} \) and then use this valuation in the application of the concrete external rule to the last state of \( \pi \), we exhaust all the possible values that can be used during the application of the concrete version of the rule. This is because the valuations
of input variables must satisfy the guard and case condition, the same way as valuations of symbolic variables must satisfy path constraints, which are built of guards and conditions. If we use each valuation $val$ and the created valuation to concretize $\hat{\pi}'$ we will get the same same new target state and in turn new path as $\pi'$.

- the timeout rule

($\Rightarrow$ direction): The application of the timeout rule (Rule 4) to the last state of $\pi$ yields a new execution state and a new execution path $\pi'$. We can apply the symbolic timeout rule to the last symbolic execution state of $\pi^s$, because the timers must be the same in both last states, and if the conditions are satisfied in the concrete state they are satisfiable in the symbolic state. The application of the symbolic version of the rule to the last state of $\pi^s$ generates the state in which the location is changed (as in the concrete rule), the timer is removed from the timers set (as in the concrete rule) and the effects are executed (as in the concrete rule). If the valuation $val'$ the same as $val$ is used to make this state concrete, we obtain the same state as the new concrete execution state. The concretization of the new symbolic state gives the same state as the concrete one, so we have that $\pi'$ is the same as concrete$^s(\hat{\pi}'$, $val'$).

($\Leftarrow$ direction): If we apply the rule to the last state of $\hat{\pi}'$ we will get a new path $\hat{\pi}'$. We can apply the same rule to the respective concrete state, because if the conditions are satisfiable for the state, they will be satisfied when we use the original valuation $val$. So they are also satisfied for the corresponding concrete state. The resulting concretized symbolic execution state must be the same as the resulting concrete state.

- the drop rule: as in the previous case, there are no new variables involved and the only change is in the removal of the dropped signal, which means that common elements of both rules are changed in the same way. The application of concrete and symbolic version of the rule is the same and we can apply the reasoning from the default rule.

The above rules do not consider subsumption. In order to consider it we have to adjust a notion of a path to exclude transitions of subsumption type. In this way the paths reflect only the regular states of execution.
Theorem 2. Symbolic execution is an exact abstraction.

Proof. Let $LTS_{\text{model}}$ be the execution LTS of $\text{model}$ and $LTS^*_\text{model}$ be the symbolic execution LTS. Let $\Pi$ be the set of paths of $LTS_{\text{model}}$ and $\hat{\Pi}^*$ be the paths in $LTS^*_\text{model}$. The abstraction is exact iff $\Pi = \text{concrete}^o(\hat{\Pi}^*)$. If we use $\text{concrete}^*$ as $\text{concrete}^o$, then such relation follows from Theorem 1. ■

5.2 Structural abstraction

Structural abstraction allows abstracting out some execution parts of the model. The abstracted parts are treated as if they have not been explored, i.e., their behavior is omitted. In this abstraction we assume that all signals that could be received from abstracted parts are available, so this abstraction is not the same as simply removing the parts. Assumption about availability of signals makes this abstraction an overapproximation, because abstracted parts may not always actually deliver the signals. This abstraction is parametrized and the parameters are the execution parts that are abstracted.

Structural abstraction is very useful to represent the behavior of models with complex hierarchical structures. By abstracting some parts we may simplify the execution and focus on selected parts only, with their communication and interactions.

5.2.1 Execution LTS and execution rules

The execution states and transitions of the LTS used for structural abstraction have the same structure as the states and transitions of the concrete execution LTS (Definition 23 and the subsequent ones) – the only difference is the contents of the set all possible execution parts $\Phi^a$.

Definition 54 (Structural abstraction LTS)
Let $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ and let the set of abstracted execution parts be $\Gamma$. The structural abstraction of $\text{model}$ is a labeled transition system $LTS^a = (\Sigma^a, \Delta^a, \sigma^a_0, \Phi^a)$, where:

- $\Sigma^a$ is a set of execution states (the same as in Definition 23),
- $\Delta^a$ is a transition relation (the same as in Definition 23),
- $\sigma^a_0 \in \Sigma^a$ is an initial state (see Definition 55),
- $\Phi_a$ is a set of execution parts in the model that is explored. This means that it is $\Phi$ from Definition 23 with the exclusion of parts $\Gamma$, i.e., $\Phi_a = \Phi \setminus \Gamma$.

The creation of the initial execution state is also slightly different. This is because the top level part (or ancestor part) might be abstracted. In this case we create execution states with dummy state details, so the hierarchical structure of the model remains similar to the original model.

**Definition 55** (Initial structural abstraction state)

Let $\text{model} = (\text{modules, queues, queuesMap, topPart, topModule})$ with a set of abstracted execution parts $\Gamma$. The execution is defined as $\text{LTS}^a = (\Sigma^a, \Delta^a, \sigma^a_0, \Phi^a)$. Let us define a set $\Phi^a_{\text{top}} \subseteq \Phi^a$ to be the set of parts that are not abstracted, but which have their ancestors abstracted. The initial state of its execution is a state $\sigma^a_0 = (m, s_{\text{top}}, \Phi)$ with:

- $m$ is a map of queues as in Definition 31,

- $s_{\text{top}} = init^a(\Phi^a_{\text{top}}, \text{topPart})$ is the result of the initialization of the distinguished $\Phi^a_{\text{top}}$ part (see Algorithm 8),

- $\Phi = initialized(s_{\text{top}})$ is the result of gathering all execution parts that were initialized (as in Definition 31).

The initialization function $init^a$ for a set of parts $\Phi^a_{\text{top}}$ is responsible for initializing these parts. We initialize all top, non-abstracted parts starting from the top part of the model and if the part is abstracted out it is initialized to an empty state. In this way we keep the hierarchy of execution parts of the model and we can reuse already defined operations, for instance, to navigate the execution states with $\text{getState()}$. Therefore, we create empty execution states for all parts 'above' the ones in $\Phi^a_{\text{top}}$. The algorithm to generate such a top level state is given in Algorithm 8. If the current execution part is in the parts we want to initialize ($\Phi$), the initialization is performed as usual (line 3). If this is not the case we need to create a state $s$ with empty state details (line 4). If an inner part can lead us to one of the execution parts we are initializing, so if it is a prefix in one of the execution parts $\Phi$, we call the initialization recursively. Once we check all parts we return the state for the part (line 10).
Algorithm 8 An outline of function \( \text{init}^a(\Phi, \phi_{\text{current}}) \).

Require: a set of execution parts \( \Phi \)
Require: a current execution part \( \phi_{\text{current}} \)

if \( \phi_{\text{current}} \in \Phi \) then
    \( \Phi \leftarrow \Phi \setminus \{ \phi_{\text{current}} \} \)
3: return \( \text{init}(\phi_{\text{current}}) \)

for all \( \text{part} \in \text{type}(\phi_{\text{current}}) \) do
6: \( \phi_{\text{new}} \leftarrow \phi_{\text{current}} \odot \langle \text{part} \rangle \)
    if \( \exists \phi \in \Phi : \text{isPrefix}(\phi_{\text{new}}, \phi) \) then
7: \( \eta_{\text{part}} \leftarrow \text{init}^a(\Phi, \phi_{\text{new}}) \)
9: \( s_{\text{parts}} \leftarrow s_{\text{parts}} \cup \{ \text{part}, \eta_{\text{part}} \} \)
10: return \( s \)

In case of structural abstraction we can use the definition of the evaluation function in Definition 33 with the following adjustment. This adjustment is that any new execution part generated with the effect \( \text{new} \) must be in \( \Phi^a \) and the recipient of any signal generated with the \( \text{out} \) effect must also be in \( \Phi^a \). In this way we make sure that abstracted parts, i.e., parts outside the set \( \Phi^a \) are not initialized and signals sent to them are not stored in the queues. We want to avoid putting signals in queues for abstracted parts, because such signals are never consumed.

The set of rules that generate the transition relation is the same as the set of rules for the concrete execution presented in Section 4.2.3. We add an extra rule to deal with signals from parts that are abstracted. We assume that such signals are always available and a transition that is triggered by them always fires as if the required signal were in the queue. The rule below is very similar to the external rule, but we check whether a triggering signal is sent from an abstracted part.

Rule 11 (Abstracted signals). Let \( \sigma = (m, s_{\text{top}}, \Phi) \) be an execution state of the execution LTS of some model obtained by structural abstraction. The following describes the rule for generating a transition \( \sigma \stackrel{\text{abstract}, \lambda, o}{\longrightarrow} \sigma' \).
∀ φ ∈ ˚Φ : ¬isInitial(s_{top}, φ)∧
∃ φ ∈ ˚Φ : s_φ = getState(s_{top}, φ)∧
∃ trans ∈ s_φ.ω.l.outgoing :
    isAbstractProvider(trans.trigger, φ)∧
    ∃ v_{concrete} ∈ trans.trigger.variables ∧
    λ = (trans.trigger, v_{concrete})
    val_c = s_φ.ω.v ∪ λ.υ∧
    trans.guard ⊥ val_c ⊥∧
∃ case ∈ trans.function.cases :
    case.condition ⊥ val_c ⊥∧
    (σ′, o) = eval(case.effect, σ, φ, λ)∧
    getState(σ′, φ).ω.l = trans.target

σ (abstract, λ, o) −→ σ′

where:

- isAbstractProvider(signal, φ) checks whether a signal is provided by an abstract part. In order
to do so, we need to find the connector that contains the signal as signalIn. We follow a very
similar procedure as the one in Algorithm 5 (page 61), but this time we have an input signal
and we want to find an output signal. We have to change the condition in line 4 to:

    (connector.signalIn = signal ∧ connector.receiver = suffix)

Let the result of the call to this targetConn’ be (φ_{context}, connector) = targetConn(signal,
φ, ⟨⟩). We say that the signal is abstracted iff it is sent from an abstracted part, that is,
isAbstractProvider(signal, φ) ⊥ φ_{context} ⊥ ⟨connector.sender⟩ ∈ Γ.

Example 21. Let us consider the model from Example 5. Let us assume that the abstracted part
is ⟨top, cars⟩. The first several states of the execution of structural abstraction of this model are
shown in Figure 5.9. In the initial state S_1 the default rule is applied. In the concrete execution the
evaluation of effects for this transition would result in the initialization of two parts cars and walk.
However, because ⟨top, cars⟩ is abstracted we do not initialize it. We initialize the state of walk and
the resulting execution state is S_2. In the location of the Controller state machine there are two
possible transitions to Walk ready and Cars ready. The first transition is a regular application of
the match rule. The second transition is the result of the application of the new abstracted signals rule. The input signal of this transition is $\text{carsStarted}()$ and this signal is provided by the part $\text{cars}$, which is abstracted. Therefore, we can treat the signal as if it were external.

![Figure 5.9: An example of a structural abstraction.](image)

### 5.2.2 Relationship between structural abstraction execution and concrete execution of CFFSMs models

According to the execution semantics defined in the previous section, structural abstraction differs from concrete execution, because it does not take into account the behavior of abstracted execution
parts. Consequently, we can define a mapping from an execution state to an abstracted state by simply removing execution states of the abstracted parts. In the definition below we remove the execution state for an abstracted part \( \phi \) by removing the reference to it in the map \( s_{\text{parts}} \) of the parent. We know that the execution part of the parent is \( \phi_{\text{parent}} = \text{levelUp}(\phi) \), and we wish to access its execution state, i.e., \( \text{getState}(s_{\text{top}}^a, \phi_{\text{parent}}) \). From the mapping \( s_{\text{parts}} \) of this state we will remove element for a part \( \text{last}(\phi) \).

**Definition 56 (Abstraction of states)**

Let \( \sigma = (m, s_{\text{top}}, \Phi) \) be a concrete execution state of some model and let \( \Gamma \) be a set of abstracted parts. We have \( \text{abstract}^a(\sigma) = (m^a, s_{\text{top}}^a, \Phi^a) \) iff

\[
\forall \phi \in \Gamma : \text{getState}(s_{\text{top}}^a, \text{levelUp}(\phi)).s_{\text{parts}} \{ \text{last}(\phi) \mapsto \text{getState}(s_{\text{top}}^a, \phi) \}
\]

The definition of abstraction of states is extended to deal with execution paths. The main feature of this mapping is that if there is an execution transition between states, which is for an execution part that is abstracted, the target state of this transition is removed from the path.

**Definition 57 (Abstraction of paths)**

Let \( \pi = (\sigma_0, \sigma_1, \ldots) \) be an execution path for \( \text{LTS} = (\Sigma, \Delta, \sigma_0, \Phi) \). We define abstraction of paths \( \text{abstract}^a(\pi)_{\text{base}} = (\sigma_0^a, \sigma_1^a, \ldots) \) to be a path such that \( \forall i \geq 0 : \sigma_i^a = \text{abstract}^a(\sigma_i) \). Next, we define a new path \( \text{abstract}^a(\pi) = (\sigma_0^a, \sigma_1^a, \ldots) \) such that \( \forall i \geq 0 : \sigma_i^a \stackrel{\delta}{\rightarrow} \sigma_{i+1}^a = \text{next}(i, \hat{\pi}^a) \). The operation \( \text{next} \) gets the next execution state in the path for which the incoming transition is not for an abstracted part if such state exists. The operation is defined as:

\[
\text{next}(i, \hat{\pi}^a) = \begin{cases} 
\hat{\pi}[i+1] & \text{if } \exists (\hat{\pi}[i], \delta, \hat{\pi}[i+1]) \in \Delta : \delta.\phi \notin \Gamma \\
\text{next}(i+1, \hat{\pi}^a) & \text{if } \exists (\hat{\pi}[i], \delta, \hat{\pi}[i+1]) \in \Delta : \delta.\phi \in \Gamma \\
\emptyset & \text{if } \forall j \geq i : (\hat{\pi}[j], \delta, \hat{\pi}[j+1]) \land \delta.\phi \notin \Gamma \end{cases}
\]

We have \( \text{abstract}^a(\pi) = \text{abstract}^a(\pi) \). The operation \( \text{abstract}^a \) can be extended to paths \( \text{abstract}^a(\Pi) \) and return a set of paths that are abstracted paths from \( \Pi \).

**Lemma 3.** The abstraction operation \( \text{abstract}^a(\pi) \) is a function.

**Proof sketch.** Note that the abstraction simply replaces states in a given path and then removes some of them. Consequently abstraction always maps a given concrete path to the same abstracted path and this operation can be performed on all concrete paths.
CHAPTER 5. TYPES OF ABSTRACTIONS FOR CFFSMS

The following theorem states that the abstracted concrete paths are included in the paths obtained by structural abstraction. The proof of this property shows that any rule applied on the last state of a given abstracted path is applied on its abstract version.

**Theorem 3.** Let $LTS$ be a concrete execution LTS with paths $\Pi$ for some model and let $LTS^a$ be an abstraction with abstracted parts $\Gamma$ and with paths $\hat{\Pi}^a$. Let $\text{abstract}^a(\Pi)$ be paths obtained with the $\text{abstract}^a(\pi)$ operation for all $\pi \in \Pi$. The following relation between paths holds: $\text{abstract}^a(\Pi) \subseteq \hat{\Pi}^a$.

**Proof sketch.** The proof is induction on the length of the paths.

*Base case.* For a path of length 1, we have only one such path. The abstraction procedure on this single state removes all initialized states for execution parts that are abstracted. This is exactly the same state we obtain by the initialization procedure outlined earlier in this section.

*Inductive step.* The inductive hypothesis is that $\text{abstract}^a(\Pi) \subseteq \hat{\Pi}^a$ holds for any paths of the length $n$. Figure 5.10 illustrates the inductive step. We check whether for all concrete paths $\pi'$ of the length $n + 1$ there is a path $\hat{\pi}'$ equal to $\text{abstract}^a(\pi')$. Let us consider two cases:

(i) Let us assume that we apply any rule to the last state of $\pi$ except for the match rule for which a trigger is provided by an abstracted part. Application of this rule results in a transition $\delta$ and some target execution state. Assume that the rule is applied to a non-abstracted part, which creates transition $\delta.\phi \notin \Gamma$, the same type of rule in its structural abstraction version is applicable to $\hat{\pi}^a$, since the rules for non-abstracted parts are exactly the same, so the conditions and effects of their application must be the same. This also ensures that the last
created execution states are the same up to non-abstracted parts. Assume now the rule is applied to an abstracted part, which results in a transition $\delta.\phi \in \Gamma$, the rule is not applied to the last state of $\hat{\pi}^a$. However, we remove the created transition and its target state during abstraction operations. In both cases the relation $\text{abstract}^a(\Pi) \subseteq \hat{\Pi}^a$ holds.

(ii) Let us assume that the rule is the match rule and the triggering signal has been provided by an abstracted part, such rule is used only on the last state of $\pi$, and not on the last state of $\hat{\pi}^a$. But in this case we apply the abstracted signals rule (Rule 11) assuming that the part we are considering is not abstracted. So if the resulting transition has $\delta.\phi \notin \Gamma$ we have a new state added to $\pi$, in its abstracted version $\text{abstract}^a(\pi)$ and in $\hat{\pi}^a$. The new concrete and abstracted states are the same because Rule 11 and Rule 2 are only different in the source of the trigger. Consequently, the generated state is exactly the same for non-abstracted parts. If the transition is for an execution part in the abstracted set then it is removed during abstraction. Note that in case of concrete execution and the match rule the input variables are bound to the concrete values provided by the signal. In case of structural abstraction and the abstracted signals rule we need to consider all possible values, including the same one as for the concrete rule. The relation holds $\text{abstract}^a(\Pi) \subseteq \hat{\Pi}^a$.

**Theorem 4.** Structural abstraction is an overapproximation.

*Proof sketch.* The definition of overapproximation in the context of structural abstraction states that if $\Pi$ is a set of paths of $LTS_{\text{model}}$ and $\hat{\Pi}^a$ is a set of paths of $LTS^a$, the following must be satisfied $\text{abstract}(\Pi) \subseteq \hat{\Pi}^a$. Because $\text{abstract}^a$ is a function (Lemma 3) we use $\text{abstract}^a$ as $\text{abstract}^o$ and the required relation follows directly from Theorem 3.

5.3 State aggregation

State aggregation is a type of abstraction that groups together several locations in a state machine. During execution such a group of locations is treated as though the locations were just one location. This means that in an abstract state we enable all transitions outgoing from aggregated locations, whereas in a concrete state only transitions outgoing from a current location are enabled.
abstraction is also parametrized. The parameter to abstraction consists of aggregation groups which are pairs of execution parts and sets of locations. The sets of locations for the same execution part cannot be overlapping.

**Definition 58 (Aggregation groups)**

Let \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \) be some model. We define a parameter to a state aggregation \( \Theta = \{\theta_1, \ldots, \theta_n\} \) and we call \( \theta_i \) an aggregation group. An aggregation group is a pair \((\phi, L)\), where \( L \) is a subset of locations (excluding initial locations) in a state machine of \( \text{type}(\phi) \), i.e., \( \theta_i.L \subseteq \text{type}(\theta_i.\phi) \cdot \text{SM}.\text{locations} \setminus \{\text{type}(\theta_i.\phi) \cdot \text{SM}.\text{locationInit}\} \). Additionally, we require that aggregation groups for the same execution parts are non-overlapping, that is,

\[
\forall \theta_i \in \Theta : \forall \theta_j \in \Theta : i \neq j \land \theta_i.\phi = \theta_j.\phi \implies \theta_i.L \cap \theta_j.L = \emptyset.
\]

### 5.3.1 Abstract LTS and execution rules

The main difference between a concrete execution LTS and the aggregation execution LTS is the possibility of having multiple locations in state details. We therefore adjust only Definition 25 of execution state details to include a set of locations.

**Definition 59 (Aggregation state details)**

Let \( \text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues}) \) with its state machine \( \text{SM} = (\text{locations}, \text{transitions}, \text{functions}, \text{guards}, \text{locationInit}) \). The state details of the execution is defined as \( \omega^g = (L^g, v^g, t^g) \), where:

- \( L^g \subseteq \text{locations} \) is a non-empty set of current locations,
- \( v^g \in \text{attributes} \) is a valuation of attributes in a module (the same as in Definition 29),
- \( t^g \subseteq \text{timers} \) is a set of currently set timers (the same as in Definition 29).

Because an execution part can be in a set of locations, some of the execution rules need to be updated. Table 5.6 summarizes the necessary changes. The drop rule is unaffected, because no transition is fired. The remaining rules all have similar changes (shown only for the match rule). First the transition to be fired can be taken from any location in \( L^g \). Next, the target execution location is a set computed with \( \text{aggrLocations} \) as shown in Definition 60. There are two cases to
consider. The first one is if a target location of a transition is in an aggregation group and then this
group becomes a target execution location. Otherwise the target state becomes a singleton set.

Table 5.6: Comparison of rules in concrete an state aggregation execution.

<table>
<thead>
<tr>
<th>Concrete</th>
<th>State aggregation abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default rule</td>
<td><strong>Rule 12.</strong> Rule 1 with source and target updates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 12. Rule 1 with source and target updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td> Rule 1 with source and target updates</td>
</tr>
<tr>
<td>Match rule</td>
<td><strong>Rule 13.</strong></td>
</tr>
<tr>
<td>∀ φ ∈ ¯Φ : ¬isInitial(stop, φ)∧</td>
<td>∀ φ ∈ ¯Φ : ¬isInitial(stop, φ)∧</td>
</tr>
<tr>
<td>∃ φ ∈ ¯Φ : sφ = getState(stop, φ)∧</td>
<td>∃ φ ∈ ¯Φ : sφ = getState(stop, φ)∧</td>
</tr>
<tr>
<td>q = m(queuesMap(type(φ)))∧</td>
<td>q = m(queuesMap(type(φ)))∧</td>
</tr>
<tr>
<td>λ = dequeue(q, φ)∧</td>
<td>λ = dequeue(q, φ)∧</td>
</tr>
<tr>
<td>m’(queuesMap(type(φ))) = q∧</td>
<td>m’(queuesMap(type(φ))) = q∧</td>
</tr>
<tr>
<td>valc = sφ.ω.v ∪ λ.v∧</td>
<td>valc = sφ.ω.v ∪ λ.v∧</td>
</tr>
<tr>
<td>∃ trans ∈ sφ.ω.l.outgoing :</td>
<td>∃ trans ∈ sφ.ω.l.outgoing :</td>
</tr>
<tr>
<td>λ.i = trans.trigger∧</td>
<td>λ.i = trans.trigger∧</td>
</tr>
<tr>
<td>trans.guard</td>
<td></td>
</tr>
<tr>
<td>∃ case ∈ trans.function.cases :</td>
<td>∃ case ∈ trans.function.cases :</td>
</tr>
<tr>
<td>case.condition</td>
<td></td>
</tr>
<tr>
<td>(σ’, o) =</td>
<td>(σ’, o) =</td>
</tr>
<tr>
<td>eval(case.effect, σ, φ, valc)∧</td>
<td>eval(case.effect, σ, φ, valc)∧</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>σ (φ, match, λ, o)</th>
<th>σ’</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ (φ, match, λ, o)</td>
<td>σ’</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External rule</th>
<th><strong>Rule 14.</strong> Rule 3 with source and target updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 3</td>
<td>Rule 14. Rule 3 with source and target updates</td>
</tr>
<tr>
<td>Timeout rule</td>
<td><strong>Rule 15.</strong> Rule 4 with source and target updates</td>
</tr>
<tr>
<td>Rule 4</td>
<td>Rule 15. Rule 4 with source and target updates</td>
</tr>
</tbody>
</table>

*Continued on next page*
Definition 60 (Locations aggregation)
Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a set of aggregation groups. We define an operation that returns the aggregation group that a location belongs to, if any:

$$ \text{aggrLocations}(\text{location}) = \begin{cases} \theta_i.L & \text{if location} \in \theta_i.L \\ \{\text{location}\} & \text{otherwise} \end{cases} $$

Example 22. As an example of the state aggregation we consider the part of execution shown in Figure 5.11 assuming that $\Theta = \{\theta_1\}$ and $\theta_1 = \langle \text{top}, \{\text{Stop walk, Start cars}\} \rangle$ and that $\text{redSet}$ is an external signal (see Figure 4.2 and Example 6). We start the execution in which we can fire a transition with the $\text{carsFirst}$ signal with a value of its variable equal to zero (obviously there are other possible execution transitions, including the ones resulting from the reception of an external signal $\text{walkFirst}$). The target location is $\text{Stop walk}$, which is in the abstraction group $\theta_1$. This means that the target execution state $S_2$ contains all the states from $\theta_1.L$. When checking the applicability of the rules for the $\text{top}$ part we consider all the states this part is in. In turn we can create the execution state $S_4$, because the $\text{redSet}$ is external signal. The target of the fired transition is $\text{Start walk}$ and it is outside of $\theta_1$. The execution state $S_3$ is obtained by the usual application of the match rule for the signal $\text{walkStopped()}$.

5.3.2 Relationship between state aggregation execution and concrete execution of CFFSMs models

As indicated at the beginning of this chapter the state aggregation is an overapproximation. In order to demonstrate this property, we define an operation that maps a concrete execution state into an abstract one. We extend this operation to paths. Next we show that the path obtained from such an operation is in the abstract execution LTS.
Figure 5.11: An example of the state aggregation abstraction assuming redSet is an external signal.
Definition 61 (Aggregation of states in execution states)
Let $\sigma$ be a concrete execution state and let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a set of aggregation groups. We define the abstraction operation $\text{abstract}^g(\sigma) = \sigma^g$ to produce an abstract execution state $\sigma^g$, the same as $\sigma$ with the following change to the locations sets: $\forall \phi \in \sigma, \tilde{\Phi} : \text{getState}(\sigma^g, s_{top}, \phi) \cdot \omega \cdot L^g = \text{aggrLocations(getState(}\sigma^g, s_{top}, \phi) \cdot \omega \cdot l)$. So in each execution state for explored parts we aggregate locations.

The abstraction of paths simply uses abstraction of each state in the path.

Definition 62 (Abstraction of paths)
Let $LTS$ be an execution LTS and let $\Pi$ be a set of paths in the LTS and let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a set of aggregation groups. We define $\text{abstract}^g(\pi)$ iff $\forall i > 0 : \text{abstract}^g(\pi)[i] = \text{abstract}^g(\pi'[i])$. The operation $\text{abstract}^g$ is extended to sets of paths.

Lemma 4. For a set of paths $\Pi$ and $\Theta = \{\theta_1, \ldots, \theta_n\}$ the operation $\text{abstract}^g$ is a function.

Proof sketch. Operation $\text{abstract}^g$ can be applied to any path and it yields only one result for a given path.

Theorem 5. Let $LTS$ be a concrete execution LTS with paths $\Pi$ for some model and let $LTS^g$ be an abstraction with $\Theta = (\theta_1, \ldots, \theta_n)$ and with paths $\hat{\Pi}^g$. Let $\text{abstract}^g(\Pi)$ be the paths resulting from application of $\text{abstract}^g(\Pi)$. The following relationship between paths holds: $\text{abstract}^g(\Pi) \subseteq \hat{\Pi}^g$.

Proof sketch. The proof is induction on the length of the paths.

Base case: In case of a path of length 1 the states are the same, because initial locations cannot be part of aggregation groups.

Inductive step: Let us assume that for paths of the length $n$ the following holds: $\text{abstract}^g(\Pi) \subseteq \hat{\Pi}^g$. The inductive step is similar to the step for structural abstraction and is illustrated in Figure 5.10. To have a path of length $n + 1$ we need to apply one of the rules:

(i) if the drop rule is applied it is applicable in concrete and in abstract execution and it generates a concrete state whose abstraction is equal to the abstract state generated in the abstract LTS.

(ii) let us assume we apply some rule to the last state of $\pi$, it results in a new execution state and in a new execution path $\pi'$. The same type of rule is applied in the abstract execution to
the last state of $\hat{\pi}^g$. This is because a set of transitions that can be fired in the abstract case must include all transition that can be fired in concrete case, since the current location in the concrete case is always included in locations of the abstract case. After the transition is fired, the target location of the transition (which was set in the concrete application of the rule) is replaced by a set of locations according to $aggrLocations$. The resulting state is added and the path is $\hat{\pi}^g'$. Note that if the abstraction is applied to the last state of $\pi'$ the states are updated accordingly to $aggrLocations$ and we have $abstract^g(\pi')$. The last states of $abstract^g(\pi')$ and $\hat{\pi}^g'$ are the same which make the paths also the same.

\[ \square \]

**Theorem 6.** State aggregation is an overapproximation.

*Proof sketch.* Since the operation $abstract^g$ is a function (Lemma 4) we can use it as $abstract^g$. From Theorem 5 follows the necessary path inclusion.

\[ \square \]

### 5.4 Composing abstractions

In the previous sections we defined different types of abstractions. Here we show that abstractions can be composed. Table 5.7 summarizes the most important characteristics of compositions of abstractions.

The structure of the execution LTS is changed to the symbolic representation if the symbolic execution is applied. The way an initial state is created is changed if we use structural abstraction, since abstracted parts are not created. The default and drop rules are not changed. Other rules, that is, match, external and timeout are changed to symbolic versions or/and to deal with multiple states. Extra rules are added for symbolic execution (the rule for subsumption) and for structural abstraction (the abstract signal rule).

The combinations of different abstractions are always overapproximations - since we always use overapproximation in structural abstraction or state aggregation.
<table>
<thead>
<tr>
<th>Element</th>
<th>SAgg &amp; SAbs</th>
<th>SE &amp; SAgg</th>
<th>SE &amp; SAbs</th>
<th>SE &amp; SAgg &amp; SAbs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>θ, Θ</td>
<td>Θ</td>
<td>Θ</td>
<td>θ, Θ</td>
</tr>
<tr>
<td>LTS</td>
<td>Def. 23 with</td>
<td>Def. 39</td>
<td>Def. 39</td>
<td>Def. 39 with</td>
</tr>
<tr>
<td></td>
<td>Def. 59</td>
<td>Def. 59</td>
<td></td>
<td>Def. 59</td>
</tr>
<tr>
<td>Initial state</td>
<td>Def. 55</td>
<td>Def. 31</td>
<td>Def. 55</td>
<td>Def. 55</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Def. 33</td>
<td>Def. 33</td>
<td>Def. 33</td>
<td>Def. 33</td>
</tr>
<tr>
<td>function</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule default</td>
<td>Rule 12</td>
<td>Rule 12</td>
<td>Rule 1</td>
<td>Rule 12</td>
</tr>
<tr>
<td>Rule match</td>
<td>Rule 13</td>
<td>Rule 6 with</td>
<td>Rule 6</td>
<td>Rule 6 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule 13</td>
<td></td>
<td>Rule 13</td>
</tr>
<tr>
<td>Rule external</td>
<td>Rule 14</td>
<td>Rule 7 with</td>
<td>Rule 7</td>
<td>Rule 7 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule 14</td>
<td></td>
<td>Rule 14</td>
</tr>
<tr>
<td>Rule timeout</td>
<td>Rule 15</td>
<td>Rule 8 with</td>
<td>Rule 8</td>
<td>Rule 8 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule 15</td>
<td></td>
<td>Rule 15</td>
</tr>
<tr>
<td>Rule drop</td>
<td>Rule 5</td>
<td>Rule 5</td>
<td>Rule 5</td>
<td>Rule 5</td>
</tr>
<tr>
<td>Other rules</td>
<td>Rule 11</td>
<td>Rule 10</td>
<td>Rule 10</td>
<td>Rule 11</td>
</tr>
</tbody>
</table>

Table 5.7: The specification of the execution for composed abstractions (SAgg = state aggregation, SE = symbolic execution, SAbs = structure abstraction).
CHAPTER 5. TYPES OF ABSTRACTIONS FOR CFFSMS

5.5 Summary

In this chapter we introduced abstractions for CFFSMSs. We distinguished: symbolic execution, structural abstraction and state aggregation. For each type of abstraction we discussed the representation of the respective execution LTSs, the rules used to generate the LTS and whether the abstraction is exact, or an over- or underapproximation.

The main idea behind symbolic execution is to use symbols instead of concrete values of variables. Using symbols impacts execution LTSs because the values of variables may now be symbols and the constraints on those values must be also explicitly stored. Rules that generate the transition relation LTS must also be updated: new input variables get new symbols and guards and conditions are stored as path constraints. Because symbolic and concrete states are very similar up to received input variables, symbolic execution is an exact abstraction.

Structural abstraction is a way to abstract out the internal parts. In order to do so we must adjust the rules so that abstracted parts are never initialized. Consequently, those parts never participate in the execution of the model, except for being signal providers, i.e., if there is a part that needs a signal that is connected to a signal provided by some abstracted part this signal is assumed to be available. Because in the concrete execution this signal might not have been generated, structural abstraction is an overapproximation.

Finally, state aggregation is based on treating several states in some state machine as one state. Outgoing transitions from aggregated states are treated as available, and might be fired if at least one of the aggregated states enables it.

Abstractions may be composed to provide composed abstractions. Composing abstraction requires specification of LTSs and rules, which usually means composing different changes.
Chapter 6

Verification of CFFSMs

In the previous chapters we described syntax and semantics of a formal representation of UML-RT models, that is, of CFFSMs. In this chapter we will show how CFFSMs can be verified, i.e., how we can ask analysis questions about them and how can such questions are answered. The questions in the context of verification are properties of models. In this work we consider temporal properties and we start the chapter by presenting a temporal logic that is used to express the properties. Next, we show an on-the-fly checking algorithm and how it is used with abstractions. Finally, we introduce an exploration method that performs lazy composition of modules.

6.1 Logic for properties of CFFSMs

The logic used to represent properties of CFFSMs is is an extension of Computation Tree Logic (CTL) [35]. In this logic atomic propositions are basic properties of execution states. Such atomic propositions can be then combined using logical and temporal connectives.

Atomic propositions in our logic describe properties of execution states. We focus on several of them: being in a certain location, having execution queues with certain contents, satisfying constraints on variables and having outgoing transitions with certain input and output signals. The grammar of the atomic propositions is given.

**Definition 63** (Atomic propositions)
The atomic propositions have the following grammar:

\[
\text{atomic proposition ::= state.property:='@' part}
\]

\[
\text{state.property ::= l | '('cst') | '<signal.in.Sequential'> | signal.in '['signal.out.Sequential']'}
\]

\[
\text{signal.in.Sequential ::= signal.in | signal.in','signal.in.Sequential}
\]

\[
\text{signal.out.Sequential ::= signal.out | signal.out','signal.out.Sequential}
\]

\[
\text{signal.in ::= 'any' | signal.in.model('()'| signal.in.model('val')) | timer}
\]

\[
\text{signal.out ::= 'any' | signal.out.model('()'| signal.out.model('val'))}
\]

\[
\text{val ::= var='value | var='value','val}
\]

where:

- part is an execution part,

- l is a location,

- cst is a constraint over attributes,

- signal.in.model, signal.out.model is an input or output signal defined for one of the modules,

- var is a variable in signal.input or signal.output,

- value is a value of some variable,

- timer is a timer.

The satisfaction of atomic propositions is defined in the context of an execution state and it is explained below.

**Definition 64 (Satisfaction of atomic propositions)**

Let \( \text{model} = (\text{modules, queues, queuesMap, topPart, topModule}) \) be a model and let \( \sigma = (m, s_{\text{top}}, \Phi) \) be an execution state (see Definition 25) and let \( \phi \) be an execution part with execution state \( s_{\phi} = \text{getState}(s_{\text{top}}, \phi) \).

Given an atomic proposition \( ap \) we define \( \sigma \models ap \) as follows:

- \( \sigma \models (1:0:0) \) iff \( s_{\phi}.w.l = 1 \) (the current location of \( \phi \) is 1)
- \( \sigma \models (((\text{cst}) \otimes \phi) \iff (\text{cst}) \parallel s_{\phi}.\omega.val_c) = \text{true} \) (constraint \text{cst} with the attributes replaced by their current values is satisfied),

- \( \sigma \models <s_{i_1}, s_{i_2}, \ldots, s_{i_n}> \otimes \phi \iff \text{si}_1, \text{si}_2, \ldots, \text{si}_n \) is the sequence of signals in the map \( m(\text{queuesMap}(\text{type}(\phi))) \) (the contents of the execution queue for the module of the execution part is the same as the provided sequence \( \text{si}_1, \text{si}_2, \ldots, \text{si}_n \) with any being any signal),

- \( \sigma \models s_{i}[so_{i_1}, so_{i_2}, \ldots, so_{i_n}] \otimes \phi \iff \exists \,(\sigma, \nu, \sigma') \in \Delta : \nu.g = \text{si} \land \nu.o = so_{i_1}, so_{i_2}, \ldots, so_{i_n} \land \nu.\phi = \phi \) (there exists an outgoing transition which is triggered by \( \text{si} \) and which has the output sequence \( so_{i_1}, so_{i_2}, \ldots, so_{i_n} \))

**Example 23.** Consider the execution state shown in Figure 6.1. Here are the atomic propositions that are satisfied by this state:

- \( \text{Stop walk} \otimes <\text{top}> \): the top part is in the location \( \text{Stop walk} \),

- \( (\text{yellowD} > 0) \otimes <\text{top}, \text{cars}> \): the value of attribute \( \text{yellowD} \) in the execution state of the part \( \text{cars} \) is greater than zero,

- \( <\text{any}> \otimes <\text{top}, \text{walk}> \): the queue of the module that is the type of the execution part \( <\text{top}, \text{walk}> \) has exactly one element,

- \( \text{stopWalk()}[] \otimes <\text{top}, \text{walk}> \): there is an outgoing transition with \( \text{stopWalk()} \) as a trigger and with an empty output sequence.

Here are some atomic propositions that this state does not satisfy:

- \( \text{Red} \otimes <\text{top}, \text{cars}> \): the part \( \text{cars} \) is not in the state \( \text{Red} \),

- \( (\text{carsD} = \text{walkD}) \otimes <\text{top}> \): the values of attributes \( \text{carsD} \) and \( \text{walkD} \) are not equal,

- \( <\text{stopWalk()}, \text{stopWalk()}> \otimes <\text{top}, \text{walk}> \): in the queue for a module of \( \text{walk} \) there is only one \( \text{stopWalk()} \) signal.

- \( \text{yellowTimer[any]} \otimes <\text{top}, \text{cars}> \): there is no outgoing transition labeled with \( \text{yellowTimer} \) as a trigger.
Atomic propositions describe the basic properties of execution states. But during an execution we are more interested in sequences of such states, that is, in temporal properties. In this work we use Computation Tree Logic, since it is well suited for our LTS execution semantics. The formulas in the logic are given below (see also Section 2.3). We will use path formulas to describe the paths: whether it is possible to finally \((F)\) reach a state with a certain property, whether all states on a path \((G)\) have a certain property, whether the next state \((X)\) has it or whether one property holds until we reach a state in which some other property holds\((U)\). The CTL formulas describe the property of the state: whether all paths \((A)\) starting in a state have a property or whether only some of them do \((E)\).

**Definition 65 (Logic formulas)**

The formulas in CTL are following:

1. all atomic properties are formulas,

2. if \(f\) and \(g\) are formulas, \(\neg f, f \land g, f \lor g, AXf, EXf, AFf, EFf, AGf, EGf, E[f U g]\) and \(E[f U g]\) are also formulas,

The semantics of CTL is presented in Section 2.3. Here we present some examples using an LTS. As in the standard CTL we say that \(LTS = (\Sigma, \Delta, \sigma_0, \Phi)\) satisfies a formula \(f\), i.e., \(LTS \models f\) iff \(\sigma_0 \models f\).

**Example 24.** Consider the LTS given in Figure 6.2. We have:

- \(S_1 \models AF(Both\ ready\ \emptyset <top>)\), because all paths starting from \(S_1\) reach a state in which the
top part is in the Both ready state,

- \( S_1 \not\models AF (<\text{carsStarted()}, \text{walkStarted()} \oplus \top>) \), because along one of the paths no execution state ever has the signal in this order in the queue and we assume that this property will not hold in the not shown portion of the state space,

- \( S_1 \models EF (<\text{carsStarted()}, \text{walkStarted()} \oplus \top, \text{cars}) \), because the outgoing transition from \( S_5 \) has as a trigger \( \text{carsStarted()} \),

- \( S_1 \not\models EF ((\text{init} \oplus \top) \land (\text{init} \oplus \top, \text{walk})) \), because there is no state in which both of the parts are in their initial state and we assume that this property will not hold in the not shown portion of the state space,

- \( S_1 \models AG ((\text{walkD} = 60) \oplus \top)) \), because in all shown states it is the case that the constraint \( \text{walkD} = 60 \) is satisfied and we assume that this property will hold in all of the states of the not shown portion of the state space,

- \( S_1 \not\models EG (\text{Starting} \oplus \top)) \), because initially \( \top \) is in the init state,

- \( S_2 \models AX (\text{Starting} \oplus \top)) \), because in both successors of \( S_2 \) the top part is in the Starting state,

- \( S_2 \models EX (\text{Blinking} \oplus \top, \text{cars}) \), because in one of the successors of \( S_2 \) the part \( \text{cars} \) is in the Blinking state,

- \( S_2 \models A[(\text{init} \oplus \top, \text{cars}) \cup (\text{Blinking} \oplus \top, \text{cars})] \), because starting from \( S_2 \) the part \( \text{cars} \) is in the init state until it changes to the Blinking state,

- \( S_1 \not\models E[(\text{init} \oplus \top, \text{cars}) \cup (\text{Blinking} \oplus \top, \text{cars})] \), because in the state \( S_1 \), the part \( \text{cars} \) is not initialized, and therefore it is not in the init and not in the Blinking state.

### 6.2 On-the-fly model checking of CFFSMSs

In the previous section we showed an extension of CTL logic that can be used to express properties of CFFSMSs. In this section we introduce an algorithm that checks the satisfaction of the formulas. The
Figure 6.2: An example of LTS used to check formulas.
proposed algorithm is an on-the-fly version of the standard labeling algorithm shown in Section 2.3. We start the presentation with the preliminary assumptions and operations. Then we show the algorithm with the necessary elements. Finally, we demonstrate the correctness and complexity of the algorithm.

6.2.1 Preliminaries

The algorithm presented below assumes that the execution of CFFSMSs is finite, that is, the exploration of the LTS eventually stops. The first step to achieve finiteness is a mechanism to detect whether newly generated states have been explored before, so a mechanism to detect equality of states. Two execution states are equal if they have the same queues and the same contents of parts for each active execution part.

Definition 66 (Equal execution states)

Let \( \sigma = (m, s_{top}, \Phi) \) and \( \sigma' = (m', s_{top}', \Phi') \) be the two execution states of \( \text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule}) \). We say that \( \sigma \equiv \sigma' \) iff all conditions below are satisfied:

- \( \Phi = \Phi' \), i.e., the explored execution parts are the same,

- \( \forall \ \text{queue} \in \text{queues} : m(\text{queue}) = m'(\text{queue}) \) the contents of the queues is the same (meaning the sequence of execution signals along with their values of input variables and receiving parts are the same).

- \( s_{top} \equiv s_{top}' \) the execution states for top parts are the same as in Definition 67.

Definition 67 (Equal execution states of part)

Let \( s = (\phi, \omega, s_{parts}) \) and \( s' = (\phi', \omega', s'_{parts}) \) be two execution states of the same execution part \( \phi \) of type \( \text{module} = (\text{SM}, \text{parts}, \text{attributes}, \text{signals}, \text{connectors}, \text{timers}, \text{defaultValues}) \). We say these states are equivalent iff all conditions below are satisfied:

- \( \phi = \phi' \), i.e., execution parts are the same,

- \( \omega = \omega' \), i.e., state details are the same, that is, all elements are the same: \( \omega.l = \omega'.l \land \omega.v = \omega'.v \lor \omega.t = \omega'.t \),

- \( \forall \ p \in \text{parts} : s_{parts}(p) \equiv s'_{parts}(p) \) i.e., execution states of all inner parts are the same.
The equality of states is checked against the states that have been already generated in the execution LTS. We define the function isNew to check whether the state $\sigma$ in some LTS does not have any equivalent state, so all states in the LTS (that are not this state) are different from $\sigma$.

**Definition 68**

Let $LTS = (\Sigma, \Delta, \sigma_0, \Phi)$ be an execution LTS generated so far and let $\sigma$ be an execution state. We say that $isNew(LTS, \sigma)$ iff $\forall \sigma' \in \Sigma \setminus \{\sigma\} : \sigma' \not\equiv \sigma$.

The consequence of equality detection is that the states may have several predecessors and in the set of successors we must include successors of an equal state.

**Definition 69 (Predecessors)**

Let $\sigma$ be an execution state in $LTS = (\Sigma, \Delta, \sigma_0, \Phi)$. We define the set of predecessor states $pre(\sigma)$ to be:

$$pre(\sigma) = \{\sigma' \mid ((\sigma', \nu, \sigma) \in \Delta) \lor ((\sigma', \nu', \sigma'') \in \Delta \land (\sigma \equiv \sigma'))\}$$

**Definition 70 (Successors)**

Let $\sigma$ be an execution state in $LTS = (\Sigma, \Delta, \sigma_0, \Phi)$. We define the set of the successors states $post(\sigma)$ as:

$$post(\sigma) = \{\sigma' \mid ((\sigma, \nu, \sigma') \in \Delta) \lor ((\sigma'', \nu, \sigma') \land (\sigma \equiv \sigma''))\}$$

The similarity detection between execution states is only one of the necessary steps to make the execution LTS finite. The other two assumptions are:

- the queues are bounded with bounds specified by the user: if there is an operation that enqueues an element to a queue that is full, this element is not added. This assumption is not very restricting in practice, because we should avoid models with queues that have no constraints on their size,

- the universes of variables are finite: consequently if we have a module with attributes, there exist only a finite number of values for these attributes. Similarly, input variables have only a finite number of values. Consequently the maximal number of states that can be generated
with such values is finite. Again this assumption does not conflict with practice, since in any
implementation the domains of values are finite.

As indicated in Section 2.3 some of the temporal operators can be expressed with others. In the
model checking algorithm proposed in this section we use only formulas with logical operators \( \neg \) and
\( \lor \) and with temporal operators \( AF \), \( EU \) and \( EX \). Note that we replace \( EG \) operator with \( AF \).
It is possible, because:
\[
AFp = \neg EG\neg p.
\]

The CTL checking algorithm requires labeling functions. These functions typically store the
information about whether a given state satisfies a certain set of formulas. We use two labeling
functions. A positive one which records all formulas that are known to hold in the state and a
negative one which records all formulas that are not satisfied in the state. Because our algorithm is
on-the-fly, we need to distinguish three situations:

(i) there is sufficient information to conclude that the execution state satisfies a formula and the
formula is then put into positive labels associated with the state,

(ii) there is sufficient information to conclude that the execution states does not satisfy a formula
and the formula is then put into negative labels associated with the state,

(iii) the currently available information is not sufficient to conclude that the state satisfies or does
not satisfy a formula and the formula is not put into positive nor into negative labels associated
with the state.

This leads to the following definition of the labeling function.

**Definition 71** (Labeling functions)

Let \( \Sigma \) be a set of execution states and let \( F \) be a set of formulas. We define \( label_\oplus : \Sigma \rightarrow 2^F \) and
\( label_\ominus : \Sigma \rightarrow 2^F \), where \( 2^F \) is the powerset of \( F \). For a given formula \( f \in F \) and a state \( \sigma \in \Sigma \) we
define \( checkLabel(f, \sigma) \) to be:

\[
checkLabel(f, \sigma) = \begin{cases} 
true & \text{if } f \in label_\oplus(\sigma) \\
false & \text{if } f \in label_\ominus(\sigma) \\
null & \text{otherwise}
\end{cases}
\]
We also use the notion of subformulas of a given formula. Subformulas are simply inner formulas for temporal and logical connectives. Because we want to check all subformulas before a given formula, we have a sequence of subformulas, that is, we need to maintain their order.

**Definition 72**

Let $f$ be a formula and let $::$ be concatenation operator for sequences. Subformulas of a formula are defined recursively as:

\[
sub(f) = \begin{cases}
  \text{AF}p :: sub(p) & \text{if } f \text{ is } \text{AF}p \\
  \text{EX}p :: sub(p) & \text{if } f \text{ is } \text{EX}p \\
  E[p U q] :: sub(p) :: sub(q) & \text{if } f \text{ is } E[p U q] \\
  \lnot p :: sub(p) & \text{if } f \text{ is } \lnot p \\
  p \lor q :: sub(p) :: sub(q) & \text{if } f \text{ is } p \lor q \\
  [f] & \text{if } f \text{ is atomic proposition}
\end{cases}
\]

In the following algorithms we assume that an operation `explore` exists. This operation takes an execution state, applies all applicable rules of the execution semantics and returns all new states that the rules generate.

**Definition 73 (Exploration)**

Let $\sigma$ be an execution state in $LTS$ and let $\mathcal{R}$ be a set of rules. We define $\text{explore}(\sigma, LTS, \mathcal{R})$ to be the set of new execution states that are the result of applying all possible rules from $\mathcal{R}$.

Finally, because the proposed algorithm is on-the-fly and the exploration is performed step by step, we need to be able to check whether a given execution state has already been explored. Therefore we adjust the definition of an execution state by adding an extra element.

**Definition 74 (Execution state with exploration)**

An execution state during a step-by-step exploration of some model $= (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ is defined: $\sigma = ((m, s_{\text{top}}, \Phi, e))$, where $m, s_{\text{top}}, \Phi$ are as in Definition 25 and $e \in \{\text{true}, \text{false}\}$ indicates whether the state has already been explored.
CHAPTER 6. VERIFICATION OF CFFSMS

Algorithm 9 An outline of an algorithm check\((f, \text{model})\).

Require: formula \(f\), a model \(\text{model}\)
Require: a set of rules \(R\)
Ensure: true if \(f\) is satisfied in \(\text{model}\) and false otherwise

\[
\text{LTS} \leftarrow \text{initialize execution}
\]

1: \(T = [\sigma_0]\)
2: \(S_f \leftarrow \text{sub}(f)\)
3: \(\text{label}_\oplus \leftarrow \{\}\)
4: \(\text{label}_\ominus \leftarrow \{\}\)
5: while \(T \neq \emptyset\) do
6: \(\sigma \leftarrow \text{remove the first element from } T\)
7: \(\sigma, e \leftarrow \text{true}\)
8: if isNew\((\sigma)\) then
9: \(\text{N} \leftarrow \text{explore}(\sigma, \text{LTS}, R)\)
10: \(\text{label}(\text{N}, S_f, \text{label}_\oplus, \text{label}_\ominus)\)
11: else
12: \(\text{checkCycles}(\sigma, \text{LTS}, S_f, \text{label}_\oplus, \text{label}_\ominus)\)
13: end if
14: \(\text{result} \leftarrow \text{checkLabel}(f, \sigma_0)\)
15: if result is not null then
16: return result
17: end if
18: end if
19: \(T \leftarrow T :: N\)
20: end while
21: end while

6.2.2 On-the-fly checking algorithm

We start presentation of an on-the-fly algorithm with a high level overview of the checking and exploration and then we introduce the labeling algorithms for atomic propositions and for composite formulas. Finally, we show support for cycle detection.

The overall exploration and checking algorithm is given in Algorithm 9. It requires a formula to be checked and a model. In line 1 an LTS is initialized, that is, its initial state is created and possible execution parts are gathered (see Definition 31). Next, in line 3, subformulas of \(f\) are generated according to Definition 72. The exploration is breadth-first, so we maintain a list \(T\) of states that are to be explored. In the exploration loop starting in line 6 we get the state to be explored and if it is new (see Definition 68) we explore it (see Definition 73) and we label the newly explored states. If the state that we plan to explore is not new, then in line 14 we check whether it is a part of a cycle.

The main labeling algorithm is shown in Algorithm 10. The algorithm iterates through all subformulas and it performs different labeling operations depending on the type of each subformula. Each such operation returns a set of execution states \(C\) for which the labeling has changed. The set of changed states is a subset of \(N\), that is, a set of newly explored states. If the formula is an atomic proposition, the labeling operation in line 4 is called. If the top-most connective is propositional (i.e., \(\neg\) or \(\lor\)) the labeling operation in line 6 is called. Finally, if the formula has temporal connectives the labeling operation in line 8 is called. During the labeling we maintain the
Algorithm 10 An outline of an algorithm \( \text{label}(N, \text{LTS}, S_f, \text{label}_\oplus, \text{label}_\ominus) \).

Require: a list subformulas \( S_f = \{g_1, \ldots, g_n\} \).

Require: a set of execution states \( N \) in an execution LTS \( \text{LTS} \)

Require: labeling functions \( \text{label}_\oplus \) and \( \text{label}_\ominus \)

Ensure: updates labeling functions

\( \text{changedLabels} \leftarrow \{ (\sigma \mapsto \{g\}) \mid g \in S_f \} \)

for \( i = 0 \) to \( i < n \) do

3: if \( g_i \) is atomic proposition then

\( C \leftarrow \text{labelAP}(g, N, \text{label}_\oplus, \text{label}_\ominus) \)

if \( g_i \) is \( \neg g_1 \) or \( g_1 \lor g_2 \) then

6: \( C \leftarrow \text{labelPropositional}(g, \text{changedLabels}, \text{label}_\oplus, \text{label}_\ominus) \)

if \( g \) is \( \text{EX}g_1 \) or \( \text{AF}g_1 \) or \( \text{E}[g_1 \cup g_2] \) then

9: \( \text{changedLabels}(g) \leftarrow \text{changedLabels}(g) \cup C \)

Algorithm 11 An outline of an algorithm \( \text{labelAP}(g, N, \text{label}_\oplus, \text{label}_\ominus) \).

Require: an atomic proposition \( g \)

Require: labeling functions \( \text{label}_\oplus \) and \( \text{label}_\ominus \)

Require: a set of new state \( N \)

Ensure: updates labeling functions

Ensure: result is a set of states with changed labels

for all \( \sigma \in N \) do

3: if \( \sigma \models g \) then

\( \text{label}_\oplus(\sigma) \leftarrow \text{label}_\oplus(\sigma) \cup \{g\} \)

else

\( \text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \cup \{g\} \)

6: return \( N \)

map \( \text{changedLabels} \) (initialized in 1), which maps each subformula to a set of states whose labeling changed when checking for satisfaction of this subformula. The map is updated in line 9 to a set of states returned by one of the labeling operations.

Labeling states with atomic propositions in \( \text{labelAP} \) depends on the type of proposition as shown in Algorithm 11. Checking for atomic propositions is straightforward, because we just need to check one of the newly explored execution states and we can determine the satisfaction right away. This is performed in line 2 and we use the satisfaction relation introduced in Definition 63. Because labeling is changed for all new execution states, all the states in \( N \) have their labeling functions affected, and this set is returned.

Example 25. Let us consider two states shown in Figure 6.3 and let us assume that initially \( \text{label}_\oplus \) and \( \text{label}_\ominus \) are both empty. We have:

- let \( g \) be \text{Blinking} @ \langle \text{top, walk} \rangle, after:

\[
\text{labelAP}(g, \{S_1\}, \text{label}_\oplus, \text{label}_\ominus)
\]

we have \( \text{label}_\oplus = \{(S_1 \mapsto \{g\})\} \), \( \text{label}_\ominus = \emptyset \) and \( C = \{S_1\} \),
CHAPTER 6. VERIFICATION OF CFFSMS

- let $g$ be $\text{Blinking} \oplus <\text{top,walk}>$, after:

$$\text{labelAP}(g, \{S.2\}, \text{label}_\oplus, \text{label}_\ominus)$$

we have $\text{label}_\oplus = \{\}, \text{label}_\ominus = \{(S.2 \mapsto \{g\})\}$ and $C = \{S.2\}$.

Figure 6.3: An example of execution states used for atomic propositions labeling

Labeling operation $\text{labelPropositional}$ is presented in Algorithm 12. It is similar to checking atomic propositions, because formulas with propositional connectives refer only to the state we check. In order to determine which states are to be checked, we gather the states for which the satisfaction of subformulas has changed (lines 3 and 12). In case of disjunction formula $g$ we have to check whether either of the two subformulas is satisfied and if so label the state positively with $g$. If both subformulas are not satisfied, then the formula $g$ cannot be satisfied and it is added to the negative labels. In all other cases we cannot say and the labels remain unchanged. In case of negation we simply check whether the subformula is in labels of the state. If so, we add the formula to the opposite labels.

The third labeling operation in Algorithm 10 is defined for formulas that are temporal, that is formulas of the form $\text{EX}, \text{AF}$ or $\text{EU}$. The algorithm used to do it is outlined in Algorithm 13. First, formulas of the form $\text{EX}g_1$ are taken care of. In this case we just need to iterate through states, whose labels have been changed when labeling with $g_1$. We check each such execution state
CHAPTER 6. VERIFICATION OF CFFSMS

Algorithm 12 An outline of an algorithm \textit{labelPropositional}(g, changedLabels, label_{⊕}, label_{⊖}).

\begin{algorithm}
\begin{algorithmic}
  \REQUIRE a formula \( g \)
  \REQUIRE a map of states with updated labels \( \text{changedLabels} \)
  \REQUIRE labeling functions \( \text{label}_{⊕} \) and \( \text{label}_{⊖} \)
  \ENSURE updates labeling functions
  \ENSURE return a set of states with changed labeling
  \STATE \textbf{C} ← \{ \}
  \IF \( g \) is \( g_{1} \lor g_{2} \)
    \STATE \textbf{TC} ← \text{changedLabels}(g_{1}) \cup \text{changedLabels}(g_{2})
    \FOR ALL \( \sigma \in \text{TC} \)
      \STATE if \( g_{1} \in \text{label}_{⊕}(\sigma) \lor g_{2} \in \text{label}_{⊕}(\sigma) \)
        \STATE \text{label}_{⊕}(\sigma) ← \text{label}_{⊕}(\sigma) \cup \{ g \}
        \STATE \textbf{C} ← \textbf{C} \cup \{ \sigma \}
      \IF \( g_{1} \in \text{label}_{⊖}(\sigma) \land g_{2} \in \text{label}_{⊖}(\sigma) \)
        \STATE \text{label}_{⊖}(\sigma) ← \text{label}_{⊖}(\sigma) \cup \{ g \}
        \STATE \textbf{C} ← \textbf{C} \cup \{ \sigma \}
    \ENDIF
  \ENDIF
  \STATE \textbf{TC} ← \text{changedLabels}(g_{1})
  \FOR ALL \( \sigma \in \text{TC} \)
    \STATE if \( g_{1} \in \text{label}_{⊕}(\sigma) \)
      \STATE \text{label}_{⊖}(\sigma) ← \text{label}_{⊖}(\sigma) \cup \{ g \}
      \STATE \textbf{C} ← \textbf{C} \cup \{ \sigma \}
    \STATE if \( g_{1} \in \text{label}_{⊖}(\sigma) \)
      \STATE \text{label}_{⊕}(\sigma) ← \text{label}_{⊕}(\sigma) \cup \{ g \}
      \STATE \textbf{C} ← \textbf{C} \cup \{ \sigma \}
  \ENDIF
  \STATE return \textbf{C}
\end{algorithmic}
\end{algorithm}

and all states for which the labels have been changed are returned. As it is shown later this change is limited to the direct predecessors of a checked state. Labeling with other types of formulas is more complex, because direct and indirect predecessors may be affected. In the algorithm in the set \( \text{TC} \) we gather states whose labels have been changed when labeling with subformulas \( g_{1} \) and possibly \( g_{2} \) (line 8). For each state in \( \text{TC} \), we initialize a predecessor list \( P \) (line 13). We iterate through \( P \) with \( \sigma' \) and we label each state using the respective labeling operation (line 17 and 19). We keep checking the predecessors until there are no more predecessors. Note that if we encounter a state from \( \text{TC} \) as a parent, we prevent double labeling by removing such state from \( \text{TC} \) (line 23).

In order to label with formulas of the form \( EXg \) we use the satisfaction relation defined in Definition 75 with an algorithm shown in Algorithm 14.

\begin{definition}
(Satisfaction \( EX \))
\end{definition}

Let \( \sigma \) be an execution state and let \( f = EXg \). We have:

- \( \sigma \models EXg \) iff \( \exists \sigma' \in \text{post}(\sigma) : \sigma' \models g \);
- \( \sigma \not\models EXg \) iff \( \forall \sigma' \in \text{post}(\sigma) : \sigma' \not\models g \)

It requires checking predecessors of a given state gathered in \( \text{TC} \). For each such state \( \sigma' \) we check whether it is already labeled with \( f \) (line 4). If not, we iterate through its successors (line 8)
CHAPTER 6. VERIFICATION OF CFFSMS

Algorithm 13 An outline of an algorithm \( \text{labelTemp}(g, \text{changedLabels}, \text{label}_0, \text{label}_0) \).

- **Require:** a formula \( g \).
- **Require:** labeling functions \( \text{label}_0 \) and \( \text{label}_0 \).
- **Require:** a map of changed states \( \text{changed} \).

- **Ensure:** updates labeling functions
- **Ensure:** returns a set \( C \) with states that have their labels changed

\[
C \leftarrow \emptyset \\
\text{if } g \text{ is } \text{EX}_{g_1} \text{ then} \\
3: \quad TC \leftarrow \text{changedLabels}(g_1) \\
\quad \text{for all } \sigma \in TC \text{ do} \\
6: \quad C \leftarrow C \cup C_{\text{EX}} \\
\text{if } g \text{ is } \text{AF}_{g_1} \text{ or } E[g_1 \ U g_2] \text{ then} \\
9: \quad TC \leftarrow TC \cup \text{changedLabels}(g_2) \\
\quad \text{while } TC \neq \emptyset \text{ do} \\
12: \quad \sigma \leftarrow \text{remove a state from } TC \\
15: \quad P \leftarrow \{ \sigma \} \\
\quad \text{while } P \neq \emptyset \text{ do} \\
18: \quad \text{if } g \text{ is } \text{AF}_{g_1} \text{ then} \\
\quad hasLabelsChanged \leftarrow \text{labelAF}(g, \sigma, \text{label}_0, \text{label}_0) \\
\quad \text{if } hasLabelsChanged \text{ then} \\
21: \quad C \leftarrow C \cup \{ \sigma' \} \\
\quad P \leftarrow P \cup \text{pre}(\sigma') \\
\quad TC \leftarrow TC \setminus \{ \sigma' \} \\
24: \quad \text{return } C
\]

and if there is one labeled with \( g \) we can label its predecessor (i.e., \( \sigma' \)) with \( \text{EX}_g \) (line 11). If all of the successors of \( \sigma' \) have negative labels for \( g \), that is, there are no positive and unknown labels, we can be sure that \( \sigma' \) cannot satisfy \( \text{EX}_g \) and we add this formula to negative labels (line 17).

**Example 26.** Let us consider the example in Figure 6.4. We label with the formula \( \text{EX}(\text{EX}p) \) where \( p \) is an atomic proposition, so the subformulas are \([p, \text{EX}p, \text{EX}(\text{EX}p)]\). The states \( S_3, S_4, S_5 \) are newly explored states. According to the labeling algorithm (Algorithm 10) we start by labeling with \( p \), which is an atomic proposition and the labeling is as indicated: positively in \( S_5 \) and negatively in \( S_3, S_4 \). The result is shown in Figure 6.4a. The map of changed states \( \text{changedLabels} \) has one entry \( \{p \mapsto \{S_3, S_4, S_5\}\} \). The next formula to label is \( \text{EX}p \) and we first get the states changed when labeling with its subformula \( p \), that is, all the new states in the dashed outline. We start with \( S_3 \), and we move to its only predecessor \( S_2 \). We iterate through all its children and we find that \( p \) is satisfied in one of them, i.e., in \( S_5 \). Consequently, we label \( S_2 \) with \( \text{EX}p \) and we return this state as the changed state. Now, we check the remaining two newly explored states. Again we go to their single parent \( S_2 \), but because it has been already labeled with \( \text{EX}p \) (see line 4 of Algorithm 14) we do nothing. The result is shown in Figure 6.4c. The last subformula to label is the original formula \( f = \text{EX}(\text{EX}p) \). There is only one state changed when labeled with its subformula,
Algorithm 14 An outline of an algorithm $\text{labelEX}(f, \sigma, \textit{label}^{\oplus}, \textit{label}^{\ominus})$.

\begin{align*}
\text{Require:} & \quad \text{a formula } f = EXg, \\
\text{Require:} & \quad \text{an execution state } \sigma, \\
\text{Require:} & \quad \text{labeling functions } \textit{label}^{\oplus} \text{ and } \textit{label}^{\ominus}, \\
\text{Ensure:} & \quad \text{updates labeling functions} \\
\text{Ensure:} & \quad \text{returns states with changed labels}
\end{align*}

\begin{algorithmic}
\State $\text{TC} \leftarrow \text{pre}(\sigma)$
\State $C \leftarrow \emptyset$
\ForAll {$\sigma' \in \text{TC}$}
\State $\text{isPositive} \leftarrow \text{false}$
\State $\text{isUnknown} \leftarrow \text{false}$
\ForAll {$\sigma'' \in \text{post}(\sigma')$}
\State $\text{checkLabel}(f, \sigma'') \in \text{null}$
\State $\text{isPositive} \leftarrow \text{true}$
\State $C \leftarrow C \cup \{\sigma''\}$
\EndFor
\State $\text{label}^{\oplus}(\sigma') \leftarrow \text{label}^{\oplus}(\sigma') \cup \{EXg\}$
\EndFor
\State $\text{isPositive} \leftarrow \text{true}$
\State $\text{isUnknown} \leftarrow \text{false}$
\If {$\text{label}^{\ominus} = \text{null}$}
\State $\text{label}^{\oplus}(\sigma') \leftarrow \text{label}^{\oplus}(\sigma') \cup \{EXg\}$
\State $C \leftarrow C \cup \{\sigma'\}$
\EndIf
\EndFor
\Return $C$
\end{algorithmic}

that is, $EXp$. We go to the parent of this state and we are able to label it with $EX(EXp)$ as shown in Figure 6.4c.

![Figure 6.4: An example of labeling with a formula $EX(EXp)$](image)

Let us move on to the labeling with $AFp$ formulas. We first define satisfaction and dissatisfaction of $AF$ in Definition 76.

**Definition 76**

Let $\sigma$ be an execution state and let $AFg$ be a formula. We have:
Algorithm 15 An outline of an algorithm \( \text{labelAF}(f, \sigma, \text{label}_\oplus, \text{label}_\ominus) \).

\begin{algorithmic}
\Require a formula \( f = \text{AF}g \).
\Require an execution state \( \sigma \) whose label \( g \) has changed.
\Require labeling functions \( \text{label}_\oplus \) and \( \text{label}_\ominus \).
\Ensure updates labeling functions.
\Ensure returns true if labeling functions changed.
\State \( \text{label}_\oplus g \leftarrow \text{checkLabel}(g, \sigma) \).
\If{\( \text{label}_\oplus g = \text{true} \)} \Return \text{toPositive}(\text{AF}g, \sigma, \text{label}_\oplus, \text{label}_\ominus) \EndIf
\If{\( \neg \sigma.e \)} \Return false \EndIf
\If{\( \text{label}_\ominus g = \text{false} \)} \State \( \text{isUnknown} \leftarrow \text{false} \)
\ForAll{\( \sigma' \in \text{post}(\sigma) \)} \State \( \text{label}_f \leftarrow \text{checkLabel}(\text{AF}g, \sigma') \).
\If{\( \text{label}_f = \text{false} \)} \Return \text{toNegative}(\text{AF}g, \sigma, \text{label}_\oplus, \text{label}_\ominus) \EndIf
\EndFor
\If{\( \neg \text{isUnknown} \)} \Return \text{toPositive}(\text{AF}g, \sigma, \text{label}_\oplus, \text{label}_\ominus) \EndIf
\EndIf
\If{\( \text{label}_\ominus g = \text{null} \)} \State \( \text{isUnknown} \leftarrow \text{true} \)
\ForAll{\( \sigma' \in \text{post}(\sigma) \)} \State \( \text{label}_f \leftarrow \text{checkLabel}(\text{AF}g, \sigma') \).
\If{\( \text{label}_f \neq \text{true} \)} \Return false \EndIf
\EndFor
\EndIf
\Return \text{false} \EndIf
\end{algorithmic}

- \( \sigma \models \text{AF}g \) iff for all paths \((\sigma_1, \sigma_2, \ldots)\) such that \( \sigma = \sigma_1, \exists i > 0: \sigma_i \models g \)

- \( \sigma \not\models \text{AF}g \) iff exists a path \((\sigma_1, \sigma_2, \ldots)\) such that \( \sigma = \sigma_1, \forall i > 0: \sigma_i \not\models g \)

In order to label with a formula \( \text{AF}g \) some execution state we use the following expansion laws:
\( \text{AF}g = g \lor (\text{AX}(\text{AF}g)) \) and \( \neg \text{AF}g = \neg g \land (\text{EX}(\neg \text{AF}g)) \). We add \( \text{AF}g \) to positive labels if \( g \) is satisfied in this state or \( \text{AF}g \) is satisfied in all successors. We add \( \text{AF}g \) to negative labels if \( g \) is not satisfied in a given state and \( \text{AF}g \) is not satisfied for one of the successors. The outline of the labeling is given in Algorithm 15. In the algorithm we start by checking for satisfaction of \( g \) in a given execution state \( \sigma \). Based on this satisfaction we can distinguish three cases:

(i) if \( g \) is satisfied (line 2): In this case \( \text{AF}g \) is added to labels of \( \sigma \),

(ii) if \( g \) is not satisfied (line 6): In this case we need to iterate through successors of \( \sigma \) (line 8). If we find at least one them not satisfying \( \text{AF}g \) we can label \( \sigma \) negatively with \( \text{AF}g \) (line 10). If all successors are neither negative nor unknown, they all satisfy \( \text{AF}g \) and therefore \( \sigma \) satisfy \( \text{AF}g \) and we can label it positively (line 14).

(iii) if satisfaction of \( g \) cannot be determined (line 16): In this case we have to iterate through all successors of \( \sigma \). If we find all of them satisfying \( \text{AF}g \), we can label \( \sigma \) the same. In all other
Algorithm 16 An outline of an algorithm toPositive\((f, \sigma, \text{label}_\oplus, \text{label}_\ominus)\).

Require: a formula \(f\).
Require: an execution state \(\sigma\)
Require: label functions \(\text{label}_\oplus\) and \(\text{label}_\ominus\)
Ensure: updated labeling functions
Ensure: returns true if the positive labels has changed, and false otherwise

\[
\text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \cup \{f\}
\]
if \(\text{label}_\ominus(\sigma) = \text{null}\) then
return \text{true}
if \(\text{label}_\ominus(\sigma) = \text{false}\) then
\[
\text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \cup \{f\}
\]
return \text{true}

Algorithm 17 An outline of an algorithm toNegative\((f, \sigma, \text{label}_\oplus, \text{label}_\ominus)\).

Require: a formula \(f\).
Require: an execution state \(\sigma\)
Require: labeling functions \(\text{label}_\oplus\) and \(\text{label}_\ominus\)
Ensure: updated labeling functions
Ensure: returns true if the labels has changed, and false otherwise

\[
\text{label}_\oplus(\sigma) \leftarrow \text{checkLabel}(f, \sigma)
\]
if \(\text{label}_\oplus(\sigma) = \text{null}\) then
3: \[
\text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \cup \{f\}
\]
return \text{true}
if \(\text{label}_\ominus(\sigma) = \text{false}\) then
6: if \(\text{label}_\ominus(\sigma) = \text{true}\) then
\[
\text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \setminus \{f\}
\]
return \text{false}

if \(\text{label}_\ominus(\sigma) = \text{false}\) then
9: \[
\text{label}_\ominus(\sigma) \leftarrow \text{label}_\ominus(\sigma) \cup \{f\}
\]
return \text{true}

cases (line 22) we cannot determine the satisfiability of \(\mathbf{AF}g\) because \(\sigma\) may be later labeled with \(g\).

If a state is not labeled with \(g\) and it has not yet been explored we cannot label it and we return from the procedure (line 5).

In Algorithm 15 we use operations toPositive and toNegative. They are introduced in Algorithms 16 and 17 and they simply check whether the formula is present in the correct labeling function. If so, nothing changes. If not, we add the formula and, if necessary, we remove it from the incorrect labeling function. These procedures are required, because the labeling of states with formulas of the form \(\mathbf{AF}\) and \(\mathbf{EU}\) may change, when we encounter cycles as explained later.

Example 27. Let us consider an example in Figure 6.5 when labeling with \(\mathbf{AF}p\), where \(p\) is an atomic proposition. We start with the newly discovered states \(S_4\) and \(S_5\) in Figure 6.5a. We label them: \(S_4\) with \(\neg p\) and \(S_5\) with \(p\). We now move to labeling with \(\mathbf{AF}p\). The set of changed states for \(p\) is \(\{S_4, S_5\}\). Let us consider \(S_4\). This state is not explored and is not labeled positively with \(p\), so
we do not change anything (line 5 of Algorithm 15) and, in turn, we do not consider its predecessors (line 22 in Algorithm 13 is not executed). We proceed to $S_4$. This state is labeled with $p$ so we can label it with $AFp$ and we move to its predecessor $S_2$. This state labeled with $!p$ and explored so we can check its successors $\{S_3, S_4\}$. Because only $S_4$ is labeled with $AFp$ we do not label $S_2$, because $isUnknown$ is set to true (failed check in line 14 of Algorithm 15). Now let us assume that we explored $S_3$ with the result shown in Figure 6.5b. The set of new states contains only $S_5$, which is labeled with $p$, so we can label this state with $AFp$ (line 2) and proceed to the predecessor $S_3$. It has only one child, which w.r.t. formula $AFp$ is neither negative nor unknown, so it is labeled with $AFp$ (line 14). The next predecessor is $S_2$, which can be also labeled with $AFp$, because both of its children have $AFp$ in their positive labels (line 14). This continues up to the root as shown in Figure 6.5c.

![Diagram](image)

Figure 6.5: An example of labeling with a formula $AFp$ (black states indicate satisfaction of $p$, states inside dashed ovals are newly discovered)

Checking satisfaction for $EU$ is slightly more complex, because it involves checking for 2 subformulas. First, we define a satisfaction and dissatisfaction relation in Definition 77.

**Definition 77** (Satisfaction $EU$)

Let $\sigma$ be an execution state and let $E[g_1 U g_2]$ be a formula.

- $\sigma \models E[g_1 U g_2]$ iff for some paths $\pi = \sigma_1, \sigma_2, \ldots$ with $\sigma_1 = \sigma$, $\exists i > 0 : \pi[i] \models g_2 \land \forall j < i : \pi[j] \models g_1$. 

Algorithm 18 An outline of an algorithm \texttt{labelEU}(f, \sigma, \text{label}_0, \text{label}_\text{\textregistered}).

\begin{algorithm}
  \begin{algorithmic}
    \Require \begin{itemize}
      \item a formula \( f = E[g_1 U g_2] \)
    \end{itemize}
    \Ensure \begin{itemize}
      \item labeling functions \texttt{label}_0 \text{ and } \texttt{label}_\text{\textregistered}
    \end{itemize}
    \Require \begin{itemize}
      \item an execution state \( \sigma \) whose labels of \( g_1 \) and \( g_2 \) changed
    \end{itemize}
    \Ensure \begin{itemize}
      \item returns updated labeling functions
    \end{itemize}
    \Require \begin{itemize}
      \item \( \texttt{label}_0 \text{ and } \texttt{label}_\text{\textregistered} \)
    \end{itemize}
    \Ensure \begin{itemize}
      \item \texttt{returns} returned if the labels has changed, and false otherwise
    \end{itemize}
    \State label\_\texttt{g} = \text{checkLabel}(g_2, \sigma) \\
    \If{label\_\texttt{g} = true} \\
    \State toPositive\( (E[g_1 U g_2], \sigma, \texttt{label}_0, \texttt{label}_\text{\textregistered}) \) \\
    \EndIf \\
    \State label\_\texttt{g} = \text{checkLabel}(g_1, \sigma) \\
    \If{label\_\texttt{g} = false \land \texttt{label\_\texttt{g}} = false} \\
    \State toNegative\( (E[g_1 U g_2], \sigma, \texttt{label}_0, \texttt{label}_\text{\textregistered}) \) \\
    \EndIf \\
    \State return false \\
    \EndIf \\
    \EndAlgorithm
  \end{algorithmic}
\end{algorithm}

- \( \sigma \not\models E[g_1 U g_2] \) iff for all paths \( \pi = \sigma_1, \sigma_2, \ldots \) with \( \sigma_1 = \sigma, \forall i > 0 : \pi[i] \not\models g_2 \lor \exists i > 0 : \pi[i] \not\models g_1 \land \pi[i] \not\models g_2 \)

The labeling uses the following expansion laws: \( E[g_1 U g_2] = g_2 \lor (g_1 \land EX(E[g_1 U g_2])) \) and \( \neg E[g_1 U g_2] = \neg g_2 \land (\neg g_1 \lor AX \neg E[g_1 U g_2]) \). We add \( E[g_2 U g_2] \) to the positive labels of a state if \( g_2 \) is satisfied in this state or \( g_1 \) is satisfied and there is at least one successor with satisfying \( E[g_1 U g_2] \). We add \( E[g_1 U g_2] \) to the negative labels if \( g_2 \) is not satisfied and either \( g_1 \) is not satisfied or all successor states do not satisfy \( E[g_1 U g_2] \). The labeling algorithm is shown in Algorithm 18.

First, we consider two situations in which only the checked state needs to be considered. If \( g_2 \) is satisfied, we can add \( f \) to the positive labels (line 2). If neither \( g_1 \) nor \( g_2 \) are satisfied, we can add the formula to the negative labels (line 5). If at this point we cannot determine satisfiability of \( f \) and the current state has not been yet explored, we do not have enough information about the execution state and we do not label it. Now, we consider the case if \( g_1 \) is satisfied and we iterate through its successors. If we find at least one of them satisfying \( E[g_1 U g_2] \), we can add the formula to the positive labels of \( \sigma \) (line 13). If all successors do not satisfy \( E[g_1 U g_2] \), that is, none of them has a positive or the \texttt{null} label, we can negatively label the current state (line 17). Finally,
we can consider a case in which satisfaction of $g_1$ cannot be determined in the current state. In this situation, we can only check for the negative result (line 24). This happens if in the current state $g_2$ is not satisfied and all successors have negative labels. Note that in such a case the satisfaction of $g_1$ is irrelevant, because even if it becomes satisfied, none of its successors is labeled with $E[g_1 U g_2]$, therefore, the current state would not be labeled positively.

**Example 28.** Let us consider the example in Figure 6.6 and let us assume we label with a formula $E[p U q]$, where $p$ and $q$ are atomic propositions. Consider the states in Figure 6.6a with new states \{S3, S4\} and labeled with atomic proposition $p$ and $q$. Now let us move to labeling with the temporal formula in states S3 and S4. In S3 we have a positive label for $p$ and a negative label $q$, which means we move to checking whether the state is explored (line 8). Because it is not, we cannot determine the satisfiability yet. In S4 both subformulas $p$ and $q$ are false and we can label the state negatively (line 5). The labeling of the current state has changed, so we move to the predecessor S2. In this state $p$ is satisfied and we check the successors. Neither of them has a positive label for $E[p U q]$ so we cannot label it with $E[p U q]$. But not all of them have negative labels (S3 has neither a positive nor a negative label), so we cannot label S2. Now, let us assume the state has been explored as shown in Figure 6.6b. We label the new state S5 with $p$ and $q$. We start labeling with the formula. Because $q$ is satisfied in this state, we can label it with $E[p U q]$ (line 2). We move to S3 and we check its successors (in this state $p$ is satisfied, but $q$ is not). Because one of the successors has a positive label, we also label S3 positively, and move to its predecessor and so on. The result is given in Figure 6.6c.

The important part of the labeling algorithm is the correct treatment of cycles. In the proposed approach cycles are the result of detecting similar states and are detected when a state to explore is equivalent to some other already explored state. In Algorithm 19 we present the overall algorithm for the procedure used in Algorithm 9 at line 14. First, we detect whether we have a cycle (line 1). If there is a cycle we check the labeling of $AF$ and $EU$ formulas (line 5), since all other formulas and propositions are unaffected by the presence of cycles. If the state has not yet been labeled with $AF$ or $EU$ we add the negative label (line 7). The negative labeling is in this case tentative and if we explore more of the model we may find a path that satisfies $EU$ or $AF$. We assume that the labeling becomes fixed as soon as labels of a given state are the same as the labels of its equivalent
CHAPTER 6. VERIFICATION OF CFFSMS

143

(a) \textit{labelEU}(E[p U q])

(b) explore()

(c) \textit{labelEU}(E[p U q])

Figure 6.6: An example of labeling with a formula $E[p U q]$ (black states indicate satisfaction of $q$, grey states of $p$ and dashed oval includes new states)

Algorithm 19 An outline of an algorithm \textit{checkCycles}(\sigma, \textit{LTS}, S_f, \textit{label}_\text{⊕}, \textit{label}_\text{⊖}).

\begin{enumerate}
\item \textbf{Require:} a new execution state $\sigma$ in \textit{LTS}
\item \textbf{Require:} subformulas $S_f$
\item \textbf{Require:} labeling functions $\textit{label}_\text{⊕}$ and $\textit{label}_\text{⊖}$
\item \textbf{Ensure:} updated labeling functions
\begin{enumerate}
\item $\sigma' \leftarrow \textit{detectCycle}(\sigma)$
\item \textbf{if} $\sigma' \neq \text{null}$ \textbf{then}
\begin{enumerate}
\item \textbf{for all} $g \in S_f$ \textbf{do}
\begin{enumerate}
\item \textbf{if} $g$ is $\textit{AF}g_1$ or $E[g_1 \text{ U } g_2]$ \textbf{then}
\begin{enumerate}
\item $\textit{label}_g \leftarrow \textit{checkLabel}(g, \sigma)$

\item \textbf{if} $\textit{label}_g = \text{null}$ \textbf{then}
\begin{enumerate}
\item $\textit{label}_\text{⊕}(\sigma) \leftarrow \textit{label}_\text{⊕}(\sigma) \cup \{g\}$
\item $\textit{label}(\text{null}, S_f, \textit{label}_\text{⊕}, \textit{label}_\text{⊖})$
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}

Detecting a cycle is straightforward and is shown in Algorithm 20. The algorithm iterates through all the predecessors of a given state. If one of them is an equivalent state, this means that there is a cycle.

Example 29. Let us consider the example in Figure 6.7 and labeling with formula $\textit{AF}p$, where $p$ is an atomic proposition. Let us assume that the newly discovered state is $S_5$ and it is equivalent

Algorithm 20 An outline of an algorithm \textit{detectCycle}(\sigma, \textit{LTS}).

\begin{enumerate}
\item \textbf{Require:} an execution state $\sigma$ in \textit{LTS}
\item \textbf{Ensure:} returns an equivalent state on the cycle with $\sigma$ or null if there is no cycle
\begin{enumerate}
\item $P \leftarrow \{\textit{pre}(\sigma)\}$
\item \textbf{while} $P \neq \text{null}$ \textbf{do}
\begin{enumerate}
\item $\sigma' \leftarrow \text{remove an element from } P$
\item \textbf{if} $\sigma' \equiv \sigma$ \textbf{then}
\begin{enumerate}
\item \textbf{return} $\sigma'$
\end{enumerate}
\end{enumerate}
\item $P \leftarrow P \cup \textit{pre}(\sigma')$
\item \textbf{return} \text{null}
\end{enumerate}
\end{enumerate}
to $S_1$. Because this state was not previously labeled with $AFp$, we add the negative label and we proceed to $S_4$. Because this state is labeled with $p$, and already with $AFp$, its labels do not change. At this point labeling is only tentative because the labeling between $S_5$ and $S_1$ is different. Let us assume that after exploration the newly discovered state is $S_6$ and it is labeled with $p$ as presented in Figure 6.7b. This means we can also label it with $AFp$. This labeling is repeated with the predecessors $S_3$, $S_2$ and $S_1$. Now note that the predecessor of $S_2$ is also $S_5$. The only successor of this state is $S_2$ and consequently we can change the label $AFp$ from negative to positive. The resulting labeling is given in Figure 6.7c. This labeling becomes fixed, because both $S_5$ and $S_1$ have the same labeling.

![Diagram](https://via.placeholder.com/150)

Figure 6.7: An example of labeling with a formula $AFp$ in the presence of cycles (black states indicate satisfaction of $p$, states inside dashed ovals are newly discovered and states connected with an edge with two arrow heads are equivalent)

### 6.2.3 Correctness and complexity

In order to prove that the proposed labeling algorithm is correct we compare it with the traditional labeling shown in Section 2.3. We demonstrate that both labeling methods are consistent, that is, the produced labeling results are not contradicting. Next, we show that the proposed labeling labels the initial state exactly the same as the traditional labeling. Consequently, satisfaction or dissatisfaction of a formula is the same in both cases.
First, we define consistency between labeling functions. We assume that the labeling function that is the result of the traditional methods is denoted with \( \text{label} \) and that the labeling with atomic proposition is given. The consistency condition requires that we label a state positively only if the label is present in \( \text{label} \) and we label a state negatively if \( \text{label} \) does not label the state.

**Definition 78**

Let \( \Sigma \) be a set of execution states and let \( f \) be a temporal formula, with subformulas \( S_f \). We have labeling functions \( \text{label}_\oplus \), \( \text{label}_\ominus \) and \( \text{label} \). We say that \( \text{label}_\oplus \) and \( \text{label}_\ominus \) are consistent with \( \text{label} \) iff \( \forall \sigma \in \Sigma : \text{label}_\oplus(\sigma) \subseteq \text{label}(\sigma) \wedge \text{label}_\ominus(\sigma) \subseteq S_f \setminus \text{label}(\sigma) \).

Now we show that the labeling obtained using Algorithm 10 is consistent with the traditional labeling for the set of states that are passed to the function. Note that we are comparing our algorithm to Algorithm 1 on page 21 (checking formulas of the form \( \text{EX} \)), Algorithm 2 on page 22 (checking formulas of the form \( \text{EU} \)) and Algorithm 4 on page 25 (checking formulas of the form \( \text{AF} \)).

**Theorem 7.** Let \( \text{LTS} \) be an execution LTS of a model and let \( f \) be a formula with subformulas \( S_f \). We have labeling functions \( \text{label}_\oplus \), \( \text{label}_\ominus \) and \( \text{label} \) defined on its execution states \( \Sigma \) and subformulas \( S_f \). Labeling functions \( \text{label}_\oplus \) and \( \text{label}_\ominus \) are the result of labeling with Algorithm 10. Labeling function \( \text{label} \) is the result of the standard algorithm shown in Section 2.3. Labeling functions \( \text{label}_\oplus \) and \( \text{label}_\ominus \) are consistent with \( \text{label} \).

**Proof sketch.** We prove the theorem by induction on the structure of a formula.

**Base case:** The base case in CTL formulas consists of atomic propositions. Labeling with atomic propositions in Algorithm 11 (as part of Algorithm 10) always yields an update in \( \text{label}_\oplus \) or \( \text{label}_\ominus \) and, since we assume it is given in the standard algorithm, the labeling is consistent.

**Inductive step:** The inductive assumption is that for all subformulas \( g \) of \( f \) and all states \( \sigma \) we have that if \( g \in \text{label}_\oplus(\sigma) \) then \( g \in \text{label}(\sigma) \) and if \( g \in \text{label}_\ominus(\sigma) \) then \( g \notin \text{label}(\sigma) \). We want to show that for all states \( f \in \text{label}_\oplus(\sigma) \implies f \in \text{label}(\sigma) \) and \( f \in \text{label}_\ominus \implies f \notin \text{label}(\sigma) \). Let us consider all forms of \( f \):

- \( \neg g \): if \( g \) is in the positive labels of some state then \( f \) goes to negative labels (see Algorithm 12).

  Following the assumption, in this case \( g \) is in \( \text{label}(\sigma) \) and \( f \notin \text{label}(\sigma) \). If \( g \) is in negative
labels, the $f$ is added to the positive labels. In this case $g \notin \text{label}(\sigma)$, but we have $f \in \text{label}(\sigma)$. Finally, if $g$ is in neither labels $f$ is not added. In all cases the consistency is maintained.

- $g_1 \lor g_2$: let us assume that $g_1$ or $g_2$ is in the positive labels of some state, then $f$ is also added to the positive labels (see Algorithm 12). Based on the inductive assumption we have that $g_1$ or $g_2$ is in label of the state, and obviously $f$ is also added to label. If both of $g_1$ and $g_2$ are in negative labels of some state, then we add $f$ to the negative label. We have that neither $g_1$ nor $g_2$ are in label of this state, and thus, $f$ is not in label. In all other cases $f$ is added to positive nor negative labels. Therefore, the consistency is kept.

- $\mathbf{EX}g$: let us assume that for some state $\sigma$ $g$ is in its positive labels. According to Algorithm 14 we go to the predecessors of $\sigma$ and we label them with $f$. We know that $g \in \text{label}(\sigma)$ and as in Algorithm 1 we have $\sigma \in T$. We check all its predecessors $S$ (exactly the same states as previously) and we label them with $f$. Now, according to Algorithm 14 if for some state $\sigma$ all its successors have $g$ in their negative labels we add $f$ to its negative label. This means in Algorithm 1 that all successors of $\sigma$ are not in $T$, because none of them have $g$ in their label (based on assumption), and we cannot add $f$ to label($\sigma$). This means that the consistency of labeling is maintained.

- $\mathbf{AF}g$: let us assume that a state $\sigma$ has $g$ in its positive labels. This state is positively labeled with $\mathbf{AF}g$ (see Algorithm 15) and its predecessors are checked (see Algorithm 13). Based on the inductive assumption we have $g$ in label($\sigma$). According to Algorithm 4 we have $\sigma \in T$ and we label it with $\mathbf{AF}g$ and also move to its predecessors in $S$. Now according to Algorithm 15, for each predecessor if it has all successors labeled positively with $\mathbf{AF}g$ we also label it positively. The same is true in Algorithm 4 as we iterate through the states in Post. Now if at least one successor has a negative label we label the state negatively. Similarly if one successor state in Post does not have $\mathbf{AF}g$ in the label, we do not label its predecessor. Let us consider cycles. If in a cycle all states have negative labels for $g$, we label them all negatively with $\mathbf{AF}g$. In Algorithm 4 none of the states can be in $T$ and if they are checked as states in $S$ all of them have a successor (i.e., a state in the cycle) not labeled with $\mathbf{AF}g$. However, if there is at least one state in a cycle that has a positive label of $g$ we cannot automatically label all states
negatively and nothing changes. The above consideration deals with all possibilities of negative and positive labeling with $AFg$, and as shown the labeling functions remain consistent.

- $E[g_1 U g_2]$: let us consider a state $\sigma$ which has $g_2$ in its positive labels. This state is positively labeled with $f$ as in Algorithm 18. Based on the inductive assumption the state has $g_2$ in its label. Consequently, it is in the initial set $T$ as in Algorithm 2. In both algorithms we move to predecessors (see Algorithm 13). Let us assume that predecessor $\sigma$ is labeled positively with $g_1$, according to Algorithm 18, since it has at least one successor labeled with $f$ we also positively label it. Similarly in Algorithm 2, because $g_1$ is in $\text{label}(\sigma)$, we label the predecessor. If both $g_1$ and $g_2$ are in negative labels of $\sigma$ we label this state negatively. In Algorithm 2, we also do not label such a state: it is not in the initial $T$ and even if the state is in $S$, because $g_1 \notin \text{label}(\sigma)$ we do not add $f$ to its labels. Another situation in Algorithm 18 in which we label $\sigma$ negatively, if the state has $g_2$ in the negative labels and all successors have $f$ in negative labels. This means that the state has $g_2 \notin \text{label}(\sigma)$, so it cannot be in initial $T$. Because all its successors do not have $f$ in labels, they are also not in $T$, so their predecessor state $\sigma$ is not labeled with $f$. Let us now consider cycles. If all states in a cycle have $g_1$ in their positive labels and all of them have only negative successors we add $f$ to negative cycles. In terms of Algorithm 2 this situation is the same as the previous one and we do not add $f$ to labels. The above includes all possible changes in positive or negative labels and we showed they do not break consistency of labeling.

We use consistency of labeling to prove that the result of checking of satisfaction of a given formula is the same in both methods. In the traditional algorithm in order to check the satisfaction we need to check whether a formula is in the labels of an initial state. Here, since labeling functions are consistent it is enough to show that at the end of the checking, a formula is in the positive or negative labels of an initial state.

**Theorem 8.** Let $f$ be a formula to be checked and let model be $\text{model}$. Once the operation $\text{check}(f, \text{model})$ in Algorithm 9 terminates we have $f \in \text{label}_\oplus(\sigma_0) \cup \text{label}_\ominus(\sigma_0)$, where $\sigma_0$ is an initial state in $\text{LTS}_{\text{model}}$. 
Proof sketch. According to Algorithm 9 the checking terminates in two cases. In the first case the labeling functions of the initial state contain the formula, which obviously satisfies the condition. The second case is if there are no more states to explore (i.e., $T$ is empty, line 6 in Algorithm 9). We show that in this case all the states, including initial one, must be labeled. For all explored states $\Sigma$ we show that $\forall \sigma \in \Sigma : S_\ell \subseteq \text{label}_\oplus(\sigma) \cup \text{label}_\ominus(\sigma)$. Therefore, $\text{checkLabel}(f, \sigma)$ cannot return null for any $f$ and $\sigma$. This means that after exploring all states we will have the result of checking, i.e., $f \in \text{label}_\oplus(\sigma_0) \cup \text{label}_\ominus(\sigma_0)$. The proof is by induction on the structure of the formula.

Base case: The base case considers atomic propositions. Labeling with atomic propositions always adds them to the labels of a state.

Inductive step: The inductive assumption is that for all subformulas $g$ of a formula $f$ we have $\forall \sigma \in \Sigma : g \in \text{label}_\oplus(\sigma) \cup \text{label}_\ominus(\sigma)$. Let us consider all forms of $f:

- $\neg g$: assuming that $g$ is in positive or negative labels, we have that $f$ must be also in labels,
- $g_1 \lor g_2$: since $g_1$ and $g_2$ are in labels, $f$ must be in labels as well,
- $\text{EX}g$: because $g$ is in labels, and all states are explored, we add $\text{EX}g$ to positive labels to all predecessors of states that have $g$ in positive labels. We add $\text{EX}g$ to negative labels in all other states,
- $\text{AF}g$: because $g$ is satisfied or dissatisfied in all states and all states are explored, we label all states positively or negatively starting from state satisfying $g$,
- $\text{E}[g_1 \mathbf{U} g_2]$: since $g_1$ and $g_2$ are in all states in their labels and all states are explored, we label all states positively or negatively with $f$ starting from the state satisfying $g_2$.

The complexity of the proposed algorithm is in the worst case the same as of the labeling algorithm, that is $O(f(S+T))$, where $f$ is the size of a formula, $S$ is number of execution states and $T$ is the number of transitions [35]. Note that the number of execution states $S$ might be different in our algorithm than in the original algorithm, because we may have the same states twice in the execution.
CHAPTER 6. VERIFICATION OF CFFSMS

The gain in running time of the proposed algorithm occurs because we can reduce $S$ in some of the cases. The number of states is the number of states necessary to explore the paths up until the satisfaction can be decided. This depends on a formula: to prove $AF$ requires checking all paths, but to disprove just one negative path, whereas $EU$ requires to find just one suitable path and to disprove we need to check all of them.

6.3 Checking with abstractions

In the previous section we presented an algorithm to check properties of the concrete execution of CFFSMSs. In this section we will show how the algorithm can be used together with abstractions. We will start with symbolic execution and then we will move to structural abstraction and state aggregation.

6.3.1 Symbolic execution

Using a model checking algorithm on $LTS^*$ is potentially very beneficial, since the number of states is typically smaller than for concrete execution. We need two adjustments to adapt the algorithm necessary to use the proposed algorithm.

The first adjustment deals with the satisfaction of atomic propositions, more specifically with checking constraints because they include attributes, which may use symbolic variables. The other atomic propositions use signals and locations, so they have the same definition as in the concrete case. As shown in Section 5.1 each symbolic execution state, due to the use of symbolic variables, represents a number of concrete execution states. Consequently, it is possible that the same constraint is satisfied by some of the concrete states represented by the symbolic execution state and dissatisfied by others, depending on the valuation used for symbolic variables in the symbolic execution state. In the context of CTL checking we need to distinguish a situation if all concrete states represented by a symbolic state satisfy the constraint, if only some of them satisfy the constraint, or if none of them satisfy it. In order to do so, we assume that satisfaction of a given constraint $\text{cst}$ represents the situation in which all concrete states satisfy $\text{cst}$, that is, $\text{cst}$ is satisfied universally. In order to check whether it is true, we check if the negation of conjugated $\text{cst}$ and path constraints cannot be satisfied with any valuation. Additionally, we introduce checking for existential version of $\text{cst}$
denoted with $E(\text{cst})$. This formula is satisfied if only some of the concrete states satisfy it. When checking the satisfaction of this proposition we check if there exists a concrete state represented by it, which satisfies the proposition, i.e., one concrete valuation of symbolic variables that satisfies a conjunction of $\text{cst}$ and path constraints. Obviously, the universal satisfaction $\text{cst}$ implies existential satisfaction $E(\text{cst})$. The definition of the satisfaction of atomic propositions, which updates the satisfaction of constraints, follows below.

**Definition 79** (Satisfaction of atomic propositions)

Let $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ be a model and let $\sigma^* = (m^*, s_{top}^*, \Phi^*, \Sigma^*)$ be a symbolic execution state (see Definition 40) and let $\phi$ be an execution part with its execution state $s^*_\phi = \text{getState}(s_{top}^*, \phi)$.

Given an atomic proposition $ap$ we define $\sigma \models ap$ as follows:

- $\sigma \models (1 \& \phi)$: same as Definition 64,
- $\sigma \models ((\text{cst}) \& \phi)$: iff $\neg \text{isSat}(-(\text{cst} \| s_{\phi}.\omega.\text{val_c} \|) \land f_{pc})),\text{where}$
  \[
  f_{pc} = \forall \phi^* \in \Phi^* : \bigwedge_{c \in \text{getState}(s_{top}^*, \phi^*)}.\omega^*.pc \text{ (the negation of the conjunction of constraint } \text{cst} \text{ and all path constraints } f_{pc} \text{ is not satisfiable)},
  \]
- $\sigma \models E((\text{cst}) \& \phi)$: iff $\text{isSat}((\text{cst} \| s_{\phi}.\omega.\text{val_c} \|) \land f_{pc})),\text{where}$ $f_{pc}$ is the same as previously (the conjunction of $\text{cst}$ and all path constraints $f_{pc}$ is satisfiable),
- $\sigma \models <s_{i_1},s_{i_2},\ldots,s_{i_n}> \& \phi$: same as Definition 64,
- $\sigma \models s_{[s_{o_1},s_{o_2},\ldots,s_{o_n}]} \& \phi$: same as Definition 64.

Using the updated notion of the satisfaction changes the labeling of atomic proposition (Algorithm 11 on page 133). This results in the following three cases of labeling the state $\sigma^*$ with the constraint $\text{cst}$:

(i) $\text{cst}$ is satisfied for all valuations of symbolic variables in $\sigma^*$, so $\text{cst}$ is in the positive labels, i.e. $\text{cst} \in label_\oplus(\sigma^*)$, which implies that $E(\text{cst})$ is also in the positive labels,

(ii) $\text{cst}$ is satisfied for some (but not all) valuations of symbolic variables, so $E(\text{cst})$ is in the positive labels and $\text{cst}$ is in the negative labels, i.e., $E(\text{cst}) \in label_\oplus(\sigma^*)$ and $\text{cst} \in label_\ominus(\sigma^*)$, 
(iii) \( \text{cst} \) is not satisfied for any valuation, so \( E(\text{cst}) \) is in the negative labels \( E(\text{cst}) \in \text{label}_\ominus(\sigma^*) \), which implies that \( \text{cst} \in \text{label}_\ominus(\sigma^*) \).

In order for the labeling with existential formulas to be correct, we place an additional requirement on constraints. We require that if we have more than one atomic proposition that is a constraint, these atomic propositions cannot share variables over which they are defined. This means that each constraint in a formula must have different variables. Sharing variables between constraints can result in incorrect checking for composing existential formulas, especially in case of the conjugation. For instance, if there exists a valuation that satisfies \( \text{cst}_1 \) and a valuation that satisfies \( \text{cst}_2 \), this does not imply that there exists a valuation that satisfies \( \text{cst}_1 \land \text{cst}_2 \) if they have common variables.

**Example 30.** Let us consider the example of a symbolic state in Figure 6.8. Let us assume two atomic propositions: \( \text{ap}_1 = (\text{carsD} > 10) @ <\text{top}> \) and \( \text{ap}_2 = (\text{carD} <= 0) @ <\text{top}> \). In case of \( \text{ap}_1 \) we have that \( E(\text{ap}_1) \) is satisfied, but \( \text{ap}_1 \) is not. In case of \( \text{ap}_2 \) we have that both \( E(\text{ap}_2) \) and \( \text{ap}_2 \) are not satisfied. In case of \( \text{ap}_1 \) we have the valuation \( \text{carsD} = \text{delayC}_1 \) and the condition becomes \( \text{delayC}_1 > 10 \). The conjunction with path constraints is \( (\text{delayC}_1 > 10) \land (\text{delayC}_1 > 0) \), which is satisfiable, so the state satisfies \( E((\text{carsD} > 10) @ <\text{top}>). \) However, the negation of the conjunction \( \neg((\text{delayC}_1 > 10) \land (\text{delayC}_1 > 0)) \) is also satisfiable, for instance for \( \text{delayC}_1 == 5 \). In case of \( \text{ap}_2 \) the conjunction is \( (\text{delayC}_1 \leq 0) \land (\text{delayC}_1 > 0) \), which is not satisfiable for any valuation of \( \text{delayC}_1 \).

![Figure 6.8: An example of a symbolic state.](image)
Using the existential satisfaction of atomic propositions affects labeling with composite formulas, because we need to propagate existential property. The propositional labeling in algorithm in Algorithm 12 (on page 135) must be updated. Let us assume we label with the formula $f$ in the state $\sigma^*$ and:

- $f = \neg g$: we label the state with $f$ if for all valuations $g$ is not satisfied, so $g$ is not satisfied universally and existentially, that is, $g \in \text{label}_\exists(\sigma^*)$ and $E(g) \in \text{label}_\exists(\sigma^*)$. We label the state with $E(f)$ if for some valuations $g$ is not satisfied, so $g$ is satisfied existentially, but not universally, that is, $g \in \text{label}_\exists(\sigma^*)$ and $E(g) \in \text{label}_\exists(\sigma^*)$.

- $f = g_1 \lor g_2$: we label with $f$ if $g_1$ or $g_2$ is in the positive labels, that is, $g_1 \in \text{label}_\oplus(\sigma^*)$ or $g_2 \in \text{label}_\oplus(\sigma^*)$. We label with $E(f)$ if $E(g_1)$ or $E(g_2)$ are in the positive labels, that is, $E(g_1) \in \text{label}_\oplus(\sigma^*)$ or $E(g_2) \in \text{label}_\oplus(\sigma^*)$.

We also adjust the labeling with formulas that have a temporal operator at the top level, that is, formulas of the form $EX, AF$ and $EU$. In Algorithm 13 (on page 136) we add labeling with the existential version of formulas, so:

- $EXg$: after line 6 we add another for loop with check for $E(XE(g))$, which can potentially label with $E(EXg)$.

- $AFg$ and $E[g_1 \ U g_2]$ after the while loop starting in line 11 we add another while loop to check satisfaction of $AF(E(g))$ and $E([E(g_1)] \ U [E(g_2)])$ and label with $E(AFg)$ and $E(E[g_1 \ U g_2])$, respectively,

- additionally, if the state is labeled positively with the universal formula, we can skip checking the existential version,

- we keep the changedLabels and whenever we label with the existential version of the formula we update the change as if the original (universal) version has been updated. Similarly, when we access changedLabels we use the original version of the formula. Although this may result in some redundant checks, it keeps the algorithm straightforward.

When the results of labeling are checked in the main algorithm (Algorithm 9) we need to add checking for existential version of the formula to check. This happens only if the formula is of the
form $EX$ and $EU$. The existential version of these formulas means that there exists a valuation in which the formula is satisfied, which in the initial state is equivalent to having the formula satisfied on some path. However, in case of the formula of the form $AF$ we make sure the universal version of the formula is satisfied.

The second major adjustment in the algorithm deals with the equality relation. In the context of symbolic execution we will use the existing subsumption relation (see Definition 43). The subsumption relation relate the states that are equal up to the used symbolic variables.

### 6.3.2 Structural abstraction and state aggregation

Structural abstraction and state aggregation are overapproximations and they need to be treated differently than a precise abstraction such as symbolic execution. Based on the formula we may know that the property cannot be checked, because it refers to the parts of the model that are abstracted out. Even if the property is checked it might not hold in the original model.

In order to deal with the first issue we need to check the property. We cannot check it:

- in case of structural abstraction if one of the atomic propositions is $\ldots \& part$ and part is abstracted,
- in case of structural abstraction if one of the atomic propositions is $<s_1,\ldots> \& part$ and $s_i$ is connected to a signal delivered by a part that is abstracted.
- in case of state aggregation if one of atomic propositions is $State \& part$ and $State$ is one of the aggregated states.

We need to make sure that the checking of other properties will not be affected by overapproximation:

- we cannot check for a formula whose satisfaction requires checking the parts of the execution that are not part of concrete execution (the overapproximated portion of the state space),
- if checking returns a counterexample, we need to make sure that this counterexample is feasible in the original model.
Algorithm 21 An outline of an algorithm `getInitialParts(f, model)`.

Require: formula $f$

Require: a model $M$

Ensure: returns $Init$, which is a set of initial parts

1. if $f$ is $EXg$ or $AFg$ or $\neg g$ then
   return $getInitialParts(g, model)$
2. if $f$ is $E[g_1 U g_2]$ or $g_1 \lor g_2$ then
   return $getInitialParts(g_1, model) \cup getInitialParts(g_2, model)$
3. if $f$ is $:\text{state @ part or (cst) @ part}$ then
   return $\{\text{part}\}$
4. if $f$ is $:\text{<si1,si2,...> @ part}$ then
   return $\{\text{part}, \text{sender of si1, sender of si2,..}\}$

6.4 Limited exploration

In the previous sections we introduced a model checking algorithm that is based on regular exploration. In this section we describe a method of exploration that limits it to the parts that are necessary to prove the property being checked.

The most important feature of the proposed exploration method is the laziness achieved by limiting the exploration to those execution parts of the model that can influence the satisfaction of the formula. These parts are called initial, and they are directly or indirectly mentioned in the formula. We also introduce two modes of exploration: regular and pull. During the regular exploration we do the exploration in which all non-initial parts are abstracted. If during such exploration we encounter a transition that is triggered by a signal provided by an abstracted part we pull for the signal. Pulling means that we explore only the provider of the required signal. We stop the pull exploration if we have the required signal in a queue or we need to proceed exploration with the initial part. Using the two modes is enough to limit the explored parts of an LTS to the part that is necessary to prove or disprove the satisfaction.

The proposed technique is based on the on-the-fly algorithm for model checking (see Algorithm 9). There are two differences: gathering the initial parts and extending the set of rules.

Discovering the set of initial parts $Init$ is based on the formula that is checked. The algorithm used to gather such execution parts is given in Algorithm 21. We follow the subformulas of a given formula (lines 2 and 4) and if we encounter atomic propositions we take the part they are for (line 6). In case of atomic propositions that include input signals we also check for the senders of the involved signals.
Example 31. Let us consider a formula $AF(\text{No Walk @ } \langle \text{top,walk} \rangle)$. For this formula the set of initial parts contain just one execution part, that is, $\text{Init} = \{ \langle \text{top,walk} \rangle \}$.

The exploration is similar to the already presented exploration with rules governing switching between regular and pull modes. We extend Definition 25 of an execution state to contain the information about the current pulling. If there are no signals to pull the exploration is regular.

Definition 80 (Limited execution state)

Let $\text{model} = (\text{modules}, \text{queues}, \text{queuesMap}, \text{topPart}, \text{topModule})$ be a model. A limited execution state is a tuple $\sigma^l = (m, s_{\text{top}}, \Phi, \text{Pulled})$, with $m, s_{\text{top}}, \Phi$ as in Definition 25 and $\text{Pulled}$ being a stack of pulled signals. Each element of $\text{Pulled}$ is a tuple $(\text{signal}, \phi_{\text{sender}}, \phi_{\text{receiver}})$, where $\text{signal}$ is a pulled signal and $\phi_{\text{sender}}$ is a part delivering a signal and $\phi_{\text{receiver}}$ is a part receiving the signal, that is, the part which started the pull.

We define additional rules that govern the execution based on limited exploration. These rules are responsible for switching between modes: full exploration and pulling. The first rule adds a signal to the pulled sequence and there are two situations when it happens.

In the first situation there is a transition from a current state triggered by a signal that is delivered by a part that is not one of the initial parts $\text{Init}$. In this case we add the pulled element (i.e., required trigger, sender of it and receiver) to the stack of pulled elements $\text{Pulled}'$. Moreover, we change the set of explored parts $\Phi'$ to contain only the sender.

Rule 16 (Add pull). Let $\sigma^l = (m, s_{\text{top}}, \Phi, \text{Pulled})$ be the execution state of some model. The following describes the rule for generating a transition $\sigma \xrightarrow{\text{pull}} \sigma'$.

\[
\forall \phi \in \Phi : \neg \text{isInitial}(s_{\text{top}}, \phi) \land \\
\exists \phi \in \Phi : s_{\phi} = \text{getState}(s_{\text{top}}, \phi) \land \\
\exists \text{trans} \in s_{\phi} . \omega . \text{.outgoing} : \\
\phi_{\text{sender}} = \text{sender} (\text{trans.trigger}, \phi) \\
\phi_{\text{sender}} \notin \text{Init}
\]

where:
- **sender**(*signalIn*, φ) is the operation that retrieves the sender of a signal, that is, a part that is in some connector for the input signal in connectors of this module or its ancestors. In order to do so, we need to find the connector that contains the trigger as *signalIn*. As mentioned in Rule 11 we do it by following Algorithm 5, with the condition that checks for the received signal and the received execution part. From the connector we found, we can find the sending part by concatenating the remaining φ_context and connector.sender,

- the target state σ′ has the same contents as σ with the following changes to ¯Φ and Pulled′:
  (i) ¯Φ′ = {φ_sender}, i.e., explored parts are switched to the new ones,
  (ii) stackPutOnTop(Pulled′, (signal, φ_sender, φ)), i.e., Pulled′ will get a new element to be pulled at the top of the stack.

The second pull situation is for parts that are not initialized, that is if we pull for *init()* from the parents of a given execution part. If the given part is not initialized, then we need to pull its parent for the so-called default signal, because only the parent may incarnate a given part. We do not pull if the current part is the topPart, since we assume that the top level part can always be initialized.

**Rule 17** (Add pull default). Let σ′ = (m, σ_top, ¯Φ, Pulled) be the execution state of some model. The following describes the rule for generating a transition σ pull → σ′.

\[
\exists \phi \in \bar{\Phi} : 
\begin{align*}
& \text{getState}(\sigma_\text{top}, \phi) = \text{null} \\
& \phi \neq \text{topPart} \\
& \phi_{\text{parent}} = \text{levelUp}(\phi)
\end{align*}
\]

σ pull → σ′

where the target state σ′ has the same contents as σ with the following changes to ¯Φ′ and Pulled′:

(i) ¯Φ′ = {φ_parent}, i.e., the set of explored parts is switched to the set containing the parent,
(ii) \( \text{stackPutOnTop}(\text{Pulled}', (\text{init}, \phi_{\text{parent}}, \phi)) \), i.e., \( \text{Pulled}' \) will get a new element to be pulled at the top of the stack.

The second type of rules is responsible for removing a signal or signals from the stack of pulled signals. There are two situations when it happens. In the first situation we remove a top level pulled element if the signal in it is present in the respective queue. We also need to switch the parts we are exploring. If after removal of the pull element there are some more elements to pull we switch to the execution part in the top pulled element. However, if there are no more elements to pull, we switch back to the initial signals.

Rule 18 (Remove pulled). Let \( \sigma^l = (m, s_{\text{top}}, \Phi, \text{Pulled}) \) be the execution state of some model. The following describes the rule for generating a transition \( \sigma^l \xrightarrow{\text{remove}} \sigma'^l \).

\[
\forall \phi \in \Phi : \neg \text{isInitial}(s_{\text{top}}, \phi) \land \\
\exists \phi' \in \Phi : s_{\phi} = \text{getState}(s_{\text{top}}, \phi) \land \\
(\text{signal}, \phi_{\text{sender}}, \phi_{\text{receiver}}) = \text{stackTop}(\text{Pulled}) \land \\
\exists \text{val} : (\phi_{\text{receiver}}, (\text{signal}, \text{val})) \in m(\text{queuesMap}(\text{type}(\phi)))
\]

where the target execution state \( \sigma'^l \) has the same contents as \( \sigma^l \) except for the:

(i) the top level element from \( \text{Pulled} \) is removed,

(ii) the explored execution parts are changed to the part that is the top element left in \( \text{Pulled} \), that is, \( \neg \text{stackIsEmpty}(\text{Pulled}') \Rightarrow (\text{signal}', \phi'_{\text{sender}}, \phi'_{\text{receiver}}) = \text{stackTop}(\text{Pulled}') \land \Phi' = \{\phi'_{\text{sender}}\} \) and if such element does not exist to the initial set \( \text{stackIsEmpty}(\text{Pulled}') \Rightarrow \Phi' = \text{Init} \).

The second case of removing the pulled element is if in the current execution state the current location of some explored part has a transition with a trigger which is delivered by one of the parts from the initial parts set \( \text{Init} \). In this case we simply remove all pulled elements.

Rule 19 (Remove all pulled). Let \( \sigma^l = (m, s_{\text{top}}, \Phi, \text{Pulled}) \) be the execution state of some model. The following describes the rule for generating a transition \( \sigma^l \xrightarrow{\text{remove}} \sigma'^l \).
\( \forall \phi \in \Phi : \neg \text{isInitial}(s_{\text{top}}, \phi) \wedge \exists \phi \in \Phi : s_{\phi} = \text{getState}(s_{\text{top}}, \phi) \wedge \exists \text{trans} \in s_{\phi}.\omega.l.\text{outgoing} : \text{sender}(\text{trans}.\text{trigger}) \in \text{Init} \)

where the target execution state \( \sigma^{l'} \) has the same contents as \( \sigma^l \) except for the:

(i) the pulled elements stack is emptied \( \text{Pulled}' = \emptyset \),

(ii) the explored execution parts are changed to the set of initial parts \( \Phi' = \text{Init} \)

As in the case of adding signals to the pulled stack, we define a version of a rule that removes from the pulled stack elements responsible for pulling for the \texttt{init()} signal. This happens if the part for which we pull has been initialized.

**Rule 20** (Remove pull default). Let \( \sigma^l = (m, s_{\text{top}}, \Phi, \text{Pulled}) \) be the execution state of some model. The following describes the rule for generating a transition \( \sigma^l \xrightarrow{\text{remove}} \sigma^{l'} \).

\[
\begin{align*}
\text{(init()}, \phi_{\text{parent}}, \phi) &= \text{stackTop(}\text{Pulled} ) \\
\text{getState}(s_{\text{top}}, \phi) &\neq \text{null}
\end{align*}
\]

where the target execution state \( \sigma^{l'} \) has the same contents as \( \sigma^l \) except for the:

(i) the top level element from \text{Pulled} is removed,

(ii) the explored execution parts are changed to the part that is the top element left in \text{Pulled}, that is, \( \neg \text{stackIsEmpty}(\text{Pulled'}) \implies (\text{signal'}, \phi'_{\text{sender}}, \phi'_{\text{receiver}}) = \text{stackTop(}\text{Pulled}' ) \land \Phi' = \{ \phi'_{\text{sender}} \} \) and if such element does not exists to the initial set \( \text{stackIsEmpty}(\text{Pulled'}) \implies \Phi' = \text{Init} \)

Initialization of an execution LTS is almost the same as in case of structural abstraction shown in Definition 55. The differences are as follows:
- as a set of abstracted parts we use a set $\Phi \setminus \text{Init}$, where $\Phi$ are all possible execution parts.

- in LTS we do not change $\Phi$, since we may use initially abstracted execution parts later.

Limited exploration is slightly different from the structural abstraction with abstracted parts $\Phi \setminus \text{Init}$. Firstly, as mentioned above we do not remove abstracted parts from the set of all possible execution parts $\Phi$. Potentially we need all execution parts, thanks to pulling operations. The consequence of this property is that we may possibly deliver signals to parts that are at the moment not explored, as well, as we may initialize such parts. Another difference is that if we need a signal that is delivered by a part, which is currently not explored, we do not simply assume that it is available but we do the pull for it. This means that we actually check whether the part that is a sender can deliver it.

**Example 32.** Let us consider the example in Figure 6.9 with the set of initial parts $\text{Init} = \{<\text{top,walk}>\}$. Since $<\text{top,walk}>$ is not fixed, we need to pull for the default signal from its parent. The parent is a top part, so we can put it into its initial location right away. We can execute the regular default rule. Firing the default transition initializes the inner parts $<\text{top,walk}>$ and $<\text{top,cars}>$. This is enough for the default signal to be taken from the pulled elements stack. We are back to exploring the $<\text{top,walk}>$ part, that is $\text{Init}$ and we can apply the default rule to it. This puts a signal into the queue for $<\text{top}>$ part, which although it is not currently explored may become explored later.

We now show that the satisfaction of formulas in the limited exploration is the same as satisfaction in the concrete exploration for propositional formulas and for formulas of the form $AF$ and $EU$ used in our labeling algorithm. Firstly, note that we include all exploration of parts that are in a given formula. If these parts require communication with other parts we explore these parts as well. In case of the regular exploration we gather all parts in $\bar{\Phi}$ right away and we check whether we can use the rules for all of them. In case of limited exploration we just reduce the parts that are currently being explored, since we manipulate the contents of $\bar{\Phi}$. In such a case rules are fired over several execution states: once for initial parts and once for singular parts that are signal providers. Consequently, we add intermediate states and we cannot check for formulas of type $EX$. However, adding some more states on paths does not change checking for the satisfaction of formulas of different form.
Figure 6.9: An example of limited exploration in case if \textit{initParts} = \{\textit{walk}\}
6.5 Summary

This chapter presents the verification methods that we developed for CFFSMs. First, we presented the logic used to represent important properties of CFFSMs. The logic is an extension of Computation Tree Logic in which we use properties of CFFSMs as atomic propositions. These properties include being in a certain location, checking whether conditions on attributes are satisfied, specifying the contents of the queues and checking the labels of outgoing transitions.

For the above logic we presented an on-the-fly algorithm that checks the satisfaction of the temporal formulas built from atomic propositions. The algorithm performs checking after each step of exploration and returns the result of checking as soon as it can be determined. We also presented how the algorithm can work with abstractions.

Finally, we presented a new type of exploration technique that enables automatic checking by keeping the exploration only to the execution parts that have to be explored.
Chapter 7

Implementation

In this chapter we outline the implementation of the concepts presented in previous sections. The implemented toolset is called Toolset for UML-RT Analysis and Verification (TUMLAV). First, we show the design of the toolset. Next, we describe its main components in more details.

The high-level design of TUMLAV is shown in Figure 7.1. There are two main steps. The first one is transformation from UML-RT to CFFSMs and the second one is execution of CFFSMs using the custom rule-based engine and, if desired, model checking.

Figure 7.1: Design of Toolset for UML-RT Analysis and Verification (TUMLAV)
The transformation in TUMLAV consists of two phases. In the first phase all elements of a
UML-RT model are transformed and in the second phase functions are generated from code. Trans-
formation of elements is implemented as an ATL transformation, which follows the transformation
introduced in Section 4.3.1 (Table 4.2 and Table 4.3). The result of the transformation is CFFSMs
with code snippets attached to transitions (this is denoted as CFFSMs (text) in Figure 7.1).

Symbolic execution of code is implemented using a custom symbolic execution engine that fol-
lows rules outlined in Section 4.3.2 (Table 4.4). As shown in the table we support basic operations
on variables, conditional statements and while loops as well as sending signals, setting timers, in-
stantiating and destroying parts. The symbolic execution for code is called for each code snippet
gathered in the previous step from transitions and states in the original UML-RT state machine.
After a symbolic execution tree is generated and its leaves are gathered they are transformed to
functions. This phase is a direct implementation of transformation shown in Section 4.3.2. The
resulting CFFSMs model is implemented as an Ecore meta model, which we show in Appendix 1. In
the implementation we require minimal effort to use other, off-the-shelf symbolic execution engines,
assuming they can provide us with the results described in Section 4.3.2.

The second step implemented in TUMLAV is execution and verification. The rule based engine
operates on CFFSMs resulting from the previous step and is responsible for generating an LTS based
on a set of provided execution rules. The outcome of the execution is an execution LTS, which is also
implemented as an Ecore model and visualized using GMF. In the implementation we visualize the
concrete execution LTS, and other executions reuse this implementation by providing the respective
details. We describe the design of the engine in the following sections.

The ATL transformation used by TUMLAV includes around 20 rules and 40 helper methods and
it has 400 lines of code. The exploration and verification part of TUMLAV consists of 19 original
plugins plus several others that are generated and used for instance for the visualization of executions.
We have plugins responsible for the symbolic execution of code. Additionally, each execution engine
has its own plugin. Finally, representing logic and performing model checking represent another
group of plugins. Because of this diversity and because some parts of the implementation are
generated, it is difficult to provide one measure. The largest plugins contain more around 60 classes,
and the smallest ones more than 5. In Appendix 2 we show a screenshot from the tool.
7.1 Design of rule-based engine

The rule-based engine executes CFFSMs using a given set of rules. This set of rules define the engine and we distinguish different kinds of engines: concrete engine, symbolic engine and so on. Figure 7.2 presents the most important classes used in engines. As mentioned, the integral part of each engine are rules. A rule contains variables, conditions and effects. A rule variable represents a type of an element of a model (such as execution part or a location). The operation `getValues` returns the currently possible values of the rule variable and these values are stored in assignments. An assignment includes pairs consisting of a rule variable and the value, that is, an actual element of execution or of the original model. A rule condition represents a condition that must be satisfied in order to fire a rule. For instance, a rule condition can be a check whether a trigger is external or whether guards are satisfied. Rule conditions are evaluated with the `match` method, which uses assignments to refer to the respective elements of an execution state. A rule effect is responsible for making changes in the execution states resulting from firing a rule, as prescribed by its method.
Algorithm 22 An outline of explore(executionState).

Require: a set of rules rules
Require: a current execution state executionState
results ← ∅
for all rule ∈ rules do
3: assignments ← generateAssignments(rule.variables, executionState)
   for all assignment ∈ assignments do
   5: if match(rule.conditions, assignment) then
6:      executionStateNew ← copy(executionState)
      fire(rule.effects, assignment, executionStateNew)
      results ← results ∪ {executionStateNew}
9: return results

fire. For instance, such an effect can be changing a location or updating values of attributes based on the update effect. Similar to rule conditions, they refer to the necessary elements of execution using rule variables and the values stored in the assignments.

Example 33. Let us assume that we have only one rule in the engine called RuleSimple. This rule has the following three rule variables: LocationVariable, TransitionVariable, CaseVariable, which refer to the respective elements of the model: LocationVariable to locations, TransitionVariable to transitions, CaseVariable to cases in functions. The rule has also one condition IsExternal, which uses TransitionVariable and one effect TargetLocation which also uses TransitionVariable as a variable.

The exploration of a model is done for a given execution state. For this state, the rules are checked whether they can be fired. An outline of the method to explore a state in some execution is given in Algorithm 22. The algorithm iterates through all rules (line 2). For each rule we generate assignments for its rule variables (line 3), that is, possible pairs of rule variable and its value. For each possible assignment we check whether it fulfills the necessary conditions (line 5) and if so we generate a new execution state. The necessary changes that need to be made are applied in line 7. This new state is then part of the returned results.

Generating assignments uses the getValues method for each variable and it is outlined in Algorithm 23. First, we create some initial empty assignment. Next, in line 2 we iterate through all variables and using getValues(), we generate values for them according to the current execution state (line 5) and to the assignments already created. Because a set of possible values might have more than one element, for each such value we need to copy all current assignments adding this value (line 6). The new assignment gets a variable and value pair (line 8).
Algorithm 23 An outline of \texttt{generateAssignments(variables, executionState)}.

Require: a set of rules variables \texttt{variables}
Require: a current execution state \texttt{executionState}
assignments ← \{emptyAssignment\}
for all \texttt{variable} ∈ \texttt{variables} do
  for all \texttt{assignment} ∈ \texttt{assignments} do
    \texttt{values} ← \texttt{variable.getValue(\texttt{executionState}, \texttt{assignment})}
  end for
  assignmentNew ← copy(\texttt{assignments})
  assignmentNew ← assignmentNew \cup \{\texttt{variable} → \texttt{value}\}
end for
return \texttt{assignments}

Example 34. Let us continue with Example 33. There are three rule variables for which we need get values and create assignments. Initially, we create an empty assignment and add it to \texttt{assignments}. Let us start with the rule variable \texttt{LocationVariable}. Let us assume that in the current execution state the method \texttt{getValue} of this rule variable returns one value, i.e., the location \texttt{Start}. We copy an empty assignment and we add to it the pair \{\texttt{LocationVariable} → \texttt{Start}\}. This pair is included in the set \texttt{assignments}. Next, we consider \texttt{TransitionVariable}. We go through the set \texttt{assignments} and we remove its only assignment. Let assume that there are 2 outgoing transitions \texttt{t1, t2} from \texttt{Start} and that \texttt{getValue} method returns them. We copy the original assignment and we add the pair \texttt{TransitionVariable} → \texttt{t1} to it, which results in \{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t1}\}. For the next value of \texttt{TransitionVariable} we first copy the existing assignment (that is of \texttt{LocationVariable}) and we add the assignment with \texttt{t2}. This gives us assignments \{\{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t1}\},\{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t2}\}\}. Now let us consider the last variable \texttt{CaseVariable}. We iterate through existing assignments. For the first one \{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t1}\} we get the values, that is, cases of the transition \texttt{t1}, which are say \texttt{c1, c2}. First we copy the current assignment and we add the pair of \texttt{CaseVariable} and \texttt{c1}. Next, we copy the current assignment again and we add the pair for the second case value. We now consider the second assignment for the first two variables, i.e., \{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t2}\} and we get the values of \texttt{CaseVariable}, which we assume is only one \texttt{c3}. We add this assignment and it gives us the following set \texttt{assignments} =
{\{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t1, CaseVariable} → \texttt{c1}\},
{\texttt{LocationVariable} → \texttt{Start, TransitionVariable} → \texttt{t1, CaseVariable} → \texttt{c2}\},
Another operation mentioned in Algorithm 22 is matching of rule conditions. Its outline is shown in Algorithm 24. In this algorithm we iterate through conditions and we check whether each of them is satisfied (line 2) in the current assignment of rule variables. If all conditions match then the operation returns \textit{true}.

\textbf{Example 35.} We continue the previous example. We consider all possible assignments and we check the only condition we defined. The rule condition is \texttt{IsExternal} and we assume that it is satisfied in case of the first transition (which is implemented in the method \texttt{match}). Consequently, we have the following assignments left after checking the rule conditions: 
\{\texttt{LocationVariable} \mapsto \texttt{Start}, \texttt{TransitionVariable} \mapsto \texttt{t2}, \texttt{CaseVariable} \mapsto \texttt{c3}\}.

Finally, in line 7 of Algorithm 22 we use the operation that provides changes in the target state. The outline of this operation is given in Algorithm 25. As in the case of the matching operation we simply iterate through effects and we apply their \texttt{fire} operation on the target state.

\textbf{Example 36.} As mentioned in Example 33 the rule has only one effect \texttt{TargetLocation}, which changes the location of the target state, which is implemented in \texttt{fire}. Because we have two assignments left to be considered, we create two target execution states.
7.2 Implementation of execution engines

We start describing the implementation of engines by making some comments about the structure of the execution LTS. The structure of the concrete execution LTS is as in Definition 23 (and the following definitions). This execution LTS serves also for other executions, however we adjust a notion of execution state in symbolic execution and in state aggregation. In the first type of abstraction we change the state details to include path constraints and mappings of symbolic variables. In the second type of abstraction the execution state details have sets of locations instead of a single location.

In the reminder of this section we describe implementations of different execution engines, by providing their rules. First, we will discuss rules for the concrete execution engine, and then we will show how they are adapted for abstraction engines.

7.2.1 Concrete execution engine

The rules that are implemented in the concrete execution engine are Default Rule, Match Rule, External Rule, Timeout Rule and Drop Rule. Their variables, conditions and effects are given in Table 7.1. For each rule we provide a set of rule variables it uses as well as conditions and effects. For conditions and effects we provide the necessary rule variables in parentheses.

The default rule needs just an execution part which is in the initial state. Initializing such a part will fire all possible initial transitions. The match rule also needs the execution part, but additionally it requires the current location of the part, a transition outgoing from this location along with guards assigned to it, case conditions and effects and the execution signal from the queue. We have several rule conditions including one that checks whether a trigger on the transition and an execution signal are the same. Additionally, case and guard conditions have to be satisfied. If this is true, the current location is updated, that is, the target location of the transition is taken, the execution signal is removed from the queue and effects in the case are executed. The external rule is very similar to the match rule, and it different from it, only because there is no queue involved and the valuation of input variables is generated (as opposed to being taken from the queue). Also in this rule case condition and guard must be satisfied and the rule effects change the location as well as evaluate the considered case. The only difference in the timeout rule is that a timeout signal is used as trigger...
and the corresponding timer must be present in the execution state of the part. Finally, the drop rule simply removes the current execution signals from the queue, if it cannot be used as trigger on the transitions.

Table 7.1: Rules in the concrete execution engine.

<table>
<thead>
<tr>
<th>Default Rule</th>
<th>Variables:</th>
<th>Part, Transition, CaseEffects, AttributesVal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions:</td>
<td>IsInitial (Part)</td>
<td></td>
</tr>
<tr>
<td>Effects:</td>
<td>Initialize (PartVar)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EvaluateInitialEffects(CaseEffects,AttributesVal)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Match Rule</th>
<th>Variables:</th>
<th>Part, Location, AttributesVal, Transition, Trigger, Guard, CaseCondition, CaseEffects, Queue, ExecutionSignal, InputVarVal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions:</td>
<td>NotIsInitial (Part)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TriggerInQueue (Trigger, ExecutionSignal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CaseSatisfied (CaseCondition, InputVariablesVal, AttributesVal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GuardSatisfied (Guard, InputVariablesVal, AttributesVal)</td>
<td></td>
</tr>
<tr>
<td>Effects:</td>
<td>TargetLocation(Part, Transition)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dequeue(Part, ExecutionSignal, Queue)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EvaluateEffects(Part, CaseEffects, InputVarVal, AttributesVal)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External Rule</th>
<th>Variables:</th>
<th>Part, Location, AttributesVal, Transition, Trigger, Guard, CaseCondition, CaseEffects, InputVarValGenerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions:</td>
<td>NotIsInitial (Part)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TriggerIsExternal (Trigger)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CaseSatisfied (CaseCondition, InputVarValGenerated, AttributesVal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GuardSatisfied (Guard, InputVarValGenerated, AttributesVal)</td>
<td></td>
</tr>
<tr>
<td>Effects:</td>
<td>TargetLocation(Part, Transition)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EvaluateEffects(Part, CaseEffects, InputVarValGenerated, AttributesVal)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timeout Rule</th>
<th>Variables:</th>
<th>Part, Location, AttributesVal, Transition, Trigger, Guard, CaseCondition, CaseEffects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions:</td>
<td>NotIsInitial (Part)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TriggerIsTimer (Trigger)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TimerIsSet (Part, Trigger)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CaseSatisfied (CaseCondition, AttributesVal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GuardSatisfied (Guard, AttributesVal)</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.1 – Continued from previous page

<table>
<thead>
<tr>
<th>Effects:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TargetLocation(Part, Transition)</td>
</tr>
<tr>
<td>EvaluateEffects(Part, CaseEffects, AttributesVal)</td>
</tr>
<tr>
<td>RemoveTimer(Part, Trigger)</td>
</tr>
</tbody>
</table>

Drop Rule

<table>
<thead>
<tr>
<th>Variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part, Location, Queue, ExecutionSignal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotIsInitial (Part)</td>
</tr>
<tr>
<td>ExecutionSignalNotTriggered (Location, ExecutionSignal)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effects:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dequeue(Queue)</td>
</tr>
</tbody>
</table>

7.2.2 Abstraction execution engines

The abstraction execution engines presented in this section use the concrete rules adjusting them as necessary. In the following we summarize these adjustments.

The rules used in the symbolic execution engine are shown in Table 7.2. The default and the drop rule are the same as in the concrete execution engine. In all other rules we update the check for satisfiability of the guard and case conditions. Instead of checking if they are satisfied in the symbolic execution engine we check whether the conditions are satisfiable. Another adjustment is for the external rule, in which we use symbolic input variables instead of concrete ones. We also check whether the combined path constraints are satisfiable. Effects in the symbolic execution engine are very similar to one in the concrete execution engine, with an additional ones, which adds case conditions and guards to the current PC.

Rules used in the structural abstraction engine are summarized in Table 7.3. They are the same as in those of the concrete engine, with an extra rule to account for using signals from abstracted parts. This rule has a condition that checks whether the sender of a trigger is one of the abstracted parts.

In Table 7.4 we present rules used in state aggregation engine. The rules are almost the same as in the concrete engine, but instead of using a single location as a rule variable we use a set of locations. Additionally, we add an extra effect which will make sure that the target locations include all locations in a given aggregation group.

We summarize the rules used in the limited exploration engine in Table 7.5. They include all rules from the concrete engine with some extra rules to support pulling. The first two rules deal
### Default Rule
- the same as in Table 7.1

### Match Rule

| Variables: | the same as in Table 7.1 |
| Conditions: | NotIsInitial (Part)  
| | TriggerInQueue (Trigger, ExecutionSignal)  
| | CaseSatisfiable (CaseCondition, InputVariablesVal, AttributesVal)  
| | GuardSatisfiable (Guard, InputVariablesVal, AttributesVal)  
| | CombinedPCSatisfiable (CaseCondition, Guard) |
| Effects: | the same as in Table 7.1 and  
| | AddToPC (Guard, CaseCondition, AttributesVal, InputVarVal) |

### External Rule

| Variables: | the same as in Table 7.1 with:  
| | InputVarValSymbolic instead of InputVarValGenerated |
| Conditions: | NotIsInitial (Part)  
| | TriggerIsExternal (Trigger)  
| | CaseSatisfiable (CaseCondition, InputVarValSymbolic, AttributesVal)  
| | GuardSatisfiable (Guard, InputVarValSymbolic, AttributesVal)  
| | CombinedPCSatisfiable (CaseCondition, Guard) |
| Effects: | the same as in Table 7.1 and  
| | AddToPC (Guard, CaseCondition, AttributesVal, InputVarValSymbolic) |

### Timeout Rule

| Variables: | the same as in Table 7.1 |
| Conditions: | NotIsInitial (Part)  
| | TriggerIsTimer (Trigger)  
| | TimerIsSet (Part, Trigger)  
| | CaseSatisfiable (CaseCondition, AttributesVal)  
| | GuardSatisfiable (Guard, AttributesVal)  
| | CombinedPCSatisfiable (CaseCondition, Guard) |
| Effects: | the same as in Table 7.1 and  
| | AddToPC (Guard, CaseCondition, AttributesVal, InputVarValSymbolic) |

### Drop Rule
- the same as in Table 7.1

Table 7.2: Rules in symbolic execution engine.
CHAPTER 7. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Default Rule</th>
<th>the same as in Table 7.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match Rule</td>
<td>the same as in Table 7.1</td>
</tr>
<tr>
<td>External Rule</td>
<td>the same as in Table 7.1</td>
</tr>
<tr>
<td>Timeout Rule</td>
<td>the same as in Table 7.1</td>
</tr>
<tr>
<td>Drop Rule</td>
<td>the same as in Table 7.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstracted Signal Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables:</td>
</tr>
<tr>
<td>Part, Location, AttributesVal, Transition, Trigger, Guard, CaseConditon, CaseEffects, InputVarValGenerated</td>
</tr>
<tr>
<td>Conditions:</td>
</tr>
<tr>
<td>NotIsInitial (Part)</td>
</tr>
<tr>
<td>TriggerSenderIsAbstracted (Trigger)</td>
</tr>
<tr>
<td>CaseSatisfied (CaseCondition,InputVarValGenerated,AttributesVal)</td>
</tr>
<tr>
<td>GuardSatisfied (Guard,InputVarValGenerated,AttributesVal)</td>
</tr>
<tr>
<td>Effects:</td>
</tr>
<tr>
<td>TargetLocation(Part,Transition)</td>
</tr>
<tr>
<td>EvaluateEffects(Part, CaseEffects, InputVarValGenerated, AttributesVal)</td>
</tr>
</tbody>
</table>

Table 7.3: Rules in structural abstraction engine.

with pulling for a trigger and pulling for initialization. In both cases we add a pull element to the currently pulled stack. Next there are three rules for removing pulled elements. First we remove elements which have been found in the respective queue. Next, we remove pulled elements if we encounter a transition triggered by a signal being provided with one of the initial parts. Finally, we remove the pull for initialization if the part is in its initial state.

As indicated in Section 5.4 we can combine several abstractions to perform more complex task. In the implementation we achieve that by reusing more “basic” engines with other abstractions. Concrete and symbolic engine are created from scratch. In order to create a structural abstraction engine we decide whether we use the concrete or symbolic engine as the basis and we add extra features to either of them. When creating a state aggregation engine we decide if we want to use concrete, symbolic or structural abstraction engine and we add the rules to either of them.

7.3 Implementation of model checking

On-the-fly model checking is implemented according to the algorithms presented in Chapter 6. Here we comment only on the most important aspects of implementation:
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Rule</td>
<td>the same as in Table 7.1</td>
</tr>
</tbody>
</table>

**Match Rule**

- **Variables:** the same as in Table 7.1 with Locations instead of Location
- **Conditions:** the same as in Table 7.1
- **Effects:** the same as in Table 7.1 with:
  - TargetAddLocations()

**External Rule**

- **Variables:** the same as in Table 7.1 with Locations instead of Location
- **Conditions:** the same as in Table 7.1
- **Effects:** the same as in Table 7.1 with:
  - TargetAddLocations()

**Timeout Rule**

- **Variables:** the same as in Table 7.1 with Locations instead of Location
- **Conditions:** the same as in Table 7.1
- **Effects:** the same as in Table 7.1 with:
  - TargetAddLocations()

**Drop Rule** the same as in Table 7.1

Table 7.4: Rules in state aggregation engine.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Rule</td>
<td>- the same as in Table 7.1</td>
</tr>
<tr>
<td>Match Rule</td>
<td>- the same as in Table 7.1</td>
</tr>
<tr>
<td>External Rule</td>
<td>- the same as in Table 7.1</td>
</tr>
<tr>
<td>Timeout Rule</td>
<td>- the same as in Table 7.1</td>
</tr>
<tr>
<td>Drop Rule</td>
<td>- the same as in Table 7.1</td>
</tr>
</tbody>
</table>

**Pull Rule**

- **Variables:** Part, Trigger
- **Conditions:** NotIsInitial (Part), TriggerSenderIsNotInInitial (Trigger)
- **Effects:** AddPullElement(Part, Trigger), SwitchExploredParts()

**Pull Default Rule**

- **Variables:** Part
- **Conditions:** NotInitialized (Part)
- **Effects:** AddDefaultPullElement(Part), SwitchExploredParts()

**Pull Remove**

- **Variables:** Part, PullElement, Queue
- **Conditions:** IsInQueue (PullElement, Queue)
- **Effects:** RemovePullElement(PullElement), SwitchExploredParts()

**Pull Remove All**

- **Variables:** Part, Location, Transition, Trigger
- **Conditions:** TriggerSenderIsInInitial (Trigger)
- **Effects:** RemoveAllPullElement(PullElement), SwitchExploredParts()

**Pull Remove Default**

- **Variables:** Part, PullElement
- **Conditions:** IsInitialized (PullElement)
- **Effects:** RemovePullElement(PullElement), SwitchExploredParts()

Table 7.5: Rules in limited exploration engine.
- CTL formulas with the proposed atomic propositions are implemented using Ecore metamodeling language and GMF \cite{2} in order to store formulas in the execution state during labeling,

- an input textual representation of CTL formulas is parsed to its Ecore representation using a JavaCC \cite{4} parser and the mapping of objects to include proper references to elements of models (such as execution parts, locations, signals),

- labeling becomes part of execution states and each state contains a set of positive and negative formulas. This is possible because models and their executions are also implemented with GMF framework.

7.4 Summary

In this chapter we present an overview of the implementation of the concepts presented in the previous chapters. We show the top level design of the tool TUMLAV. The toolset includes two steps to analyze UML-RT models: transformation and exploration with verification. In the first phase we use the ATL transformation along with a custom symbolic execution engine for code. Next, we presented the design of the rule-based execution engine for CFFSMs. We presented the structure of rules and how they are used to explore models. We introduced rule components in the context of four execution engines: concrete, symbolic, structural abstraction, state aggregation and lazy composition. We concluded the chapter with some comments on the implementation of model checking algorithm.
Chapter 8

Evaluation

In this chapter we present the evaluation of the proposed approach. The evaluation should give us answers to the following questions:

1. Can the proposed abstractions improve the analysis and understanding of UML-RT models? On which types of models and analyses do the abstractions provide the largest benefit?

2. Are the proposed verification methods usable, that is, can we check for important and interesting properties of models?

3. What is the scalability of the proposed methods?

4. Can the proposed approach be applied to industrial models?

In order to answer these evaluation questions we perform experiments using different UML-RT models. The experiments include abstractions on models and checking for satisfaction of properties in models. One set of models we use includes versions of the traffic lights example with increasing complexity. Another set of models is from the UML-RT model of a PBX system, which we obtained from our industrial partner. We perform the experiments on the subsystems of the overall model.

We performed all experiments on a standard PC with processor Intel i7 (3.07 GHz) and with 4 GB of RAM memory. In all experiments we gather the number of execution states. Typically the number of states is proportional to the running time of our experiments.
8.1 Models used in experiments

In the experiments we use three sets of models. The first one is the already introduced Controller model (see Example 5). The second set consists of variations of the traffic lights example. The last set of models includes the customized subsystems of the PBX model, which we obtained from our industrial partners.

8.1.1 Traffic lights models

We extend the traffic lights model introduced initially in Section 2.1 in Example 1 to a family of models with different complexity. We use two additional capsules with their structures given in Figure 8.1. The capsule IntersectionController is responsible for coordinating two parts of type Controller: one in the direction North-South and the other one in the directions East-West. The capsule IntersectionController is used inside the top level capsule StreetController. This capsule is responsible for managing several intersections which are on a street. The increasing number of intersection parts result in the increasing complexity of overall models. We use models with up to 3 intersections.

We omit the state machines of the new capsules. IntersectionController capsule is responsible for coordinating North-South and East-West lights. The StreetController initializes intersections and starts their respective cycles.

8.2 PBX model

We obtained the PBX model from our industrial partner. The model is a PBX (private branch exchange) system, that is, a telephone system based on extensions [6]. The model includes several layers, e.g., configuration and telephony components and some more low-level details such as data types and sockets. The layers contain several subsystems. We experimented with three of them: DeviceManager, CallControl, OAMSBSYstem and ProxyManager.

In order to use the model in TUMLA\textsc{v} we perform several modifications on the model:

- simplify action code by removing method calls, pointers and advanced data structures - they are represented with basic types and their operations,
Figure 8.1: Structure of capsules IntersectionController and StreetController
- replace plugin parts with optional parts and explicit connectors, so the capsules sharing a
  plugin part are actually connected to it,

- replace more complex variables types with the basic types.

All of the above adjustments were performed manually.

In the experiments we use the subsystems of the PBX model: CallManager, OAMSubsystem,
ProxyManager and DeviceManager. Each of them consists of up to 6 capsules with state machines
with up to 10 locations at 3 levels of nesting. We have at most 100 different signals in each sub-
system. After code generation, the adapted subsystems are between 3000 and 6500 lines of code
in C++ (which does not include the code for the UML-RT framework). We show the top level
structure of each subsystem, but we omit their UML-RT State Machines.

CallController is responsible for storing calls in a system. The structure of the capsule is shown
in Figure 8.2 and it contains two parts call1 and call2 of the type Call. Each call consists of 2
sessions (not shown in Figure 8.2): originating and terminating, which are created along with the
call. The sessions are connected to the actual devices that hold them and by receiving events from
the devices sessions and calls are managed.

OAMSubsystem is a subsystem responsible for managing extensions and initializing hardware el-
ements. The structure of this capsule is shown in Figure 8.3. It has three main parts HLR, GMSC
and SystemAdminProxy. The first part maintains the extensions and informs about them sessions
that need to be connected. GMSC is responsible for the general management of channels. Finally,
SystemAdminProxy simulates system administration by creating a predefined number of extensions.

ProxyManager is a subsystem that emulates phone related events in a system. Its structure is
given in Figure 8.4. It includes 4 parts proxy1 through proxy4 of type CellPhoneProxy, each of which
is responsible for events from a given device in the system. The maximal number of proxies is 4,
and the actual number of proxies we use can change at run-time.

Device manager manages actual devices in the system. The structure of this capsule is given in
Figure 8.5. It includes a number of device parts, in our case we set the number to 4. DeviceManager
connects them with the network (OAMSusbsytem) and with the proxies in ProxyManager that represent
phone events. Each phone might be connected to a call session in CallController, which represents
an active call between two devices in a network. Additionally, each device in a device manager is
Figure 8.2: CallController structure
Figure 8.3: OAMSSubsystem structure

Figure 8.4: ProxyManager structure
connected to an extension in a system.

8.3 Symbolic execution

In this section we present the results of experiments with the symbolic execution engine. We use this abstraction on several models and, if possible, we compare it with the concrete execution. The criterion for such comparison is the size of the generated symbolic LTS in terms of the number of states in it. When using concrete execution we use the following two assumptions to make state spaces more manageable:

- we limit the domain of all possible values of variables to some selected values. We consider possible intervals (-1,1), (-4,4) and (-10,10),

- we always perform the similarity checking on concrete states as introduced in Section 6.2.1 in Definition 66.
First, we consider the symbolic execution of the Controller model, that is, the basic model for traffic lights. Figure 8.6 gathers the results of symbolic execution of the model along with the results of the limited concrete execution with intervals (-1,1), (-4,4) and (-10,10). All experiments took less than 1 second. These bounds are given in the parentheses. The number of states in case of symbolic and the first concrete execution (i.e., column labeled 'Concrete(-1,1)' in the table) is similar, but for concrete execution grows fast with larger domains. The reason why the number of states is smaller for concrete execution with the (-1,1) interval is that during that execution not all concrete paths are explored.

In Figure 8.7 we present the results of experiments with StreetController model with 1, 2 and 3 intersections. Experiments for 1 and 2 intersections took less than a 1 minute and for 3 intersections more than 12 hours. We performed the exhaustive exploration for the first two models, the exploration for the largest model has been stopped by an OutOfMemoryException. Adding each intersection makes a model substantially more complex and the execution LTS in each execution
<table>
<thead>
<tr>
<th>Module</th>
<th>Symbolic</th>
<th>Concrete(-1,1)</th>
<th>Concrete(-4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreetController for 1 intersection</td>
<td>188</td>
<td>226</td>
<td>862</td>
</tr>
<tr>
<td>StreetController for 2 intersections</td>
<td>10 587</td>
<td>26 597</td>
<td>114 857</td>
</tr>
<tr>
<td>StreetController for 3 intersections</td>
<td>&gt;136 000</td>
<td>&gt;136 000</td>
<td>&gt;136 000</td>
</tr>
</tbody>
</table>

Figure 8.7: Number of states in the symbolic execution of StreetController model with different number of intersections.

mode is larger. In case of models for 1 and 2 traffic lights executions with concrete values are larger than with symbolic computations. The number of states in case of concrete execution for interval (-1,1) increases 20% and 150% compared to symbolic execution, whereas for interval (-4,4) the increase is 5 and 10 times.

Finally, we show the results of experiments with symbolic execution of subsystems of the PBX model in Table 8.1. The execution times for all experiments were between 12 and 24 hours. The results are only for symbolic execution, because, due to the sizes of models, we did not perform the experiments with concrete executions. Even for symbolic execution, we couldn’t perform full exploration for most of the models. In three cases we stopped at the OutOfMemoryException. The
CHAPTER 8. EVALUATION

<table>
<thead>
<tr>
<th>Model</th>
<th>CallController</th>
<th>DeviceManager</th>
<th>OAMSubsystem</th>
<th>ProxyManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of LTS</td>
<td>&gt; 350 000</td>
<td>&gt; 250 000</td>
<td>65 582</td>
<td>&gt; 200 000</td>
</tr>
</tbody>
</table>

Table 8.1: Number of states in the symbolic execution of PBX subsystems

The largest discovered state space was in case of CallController and the smallest of OAMManager.

The above experiments demonstrate that symbolic execution reduces the size of the state space. Even for 9 concrete values of variables (i.e. integer values between -4 and 4) the size of the state space increases substantially (Figure 8.6). Typically the number of integer values is much larger.

Next, we note that symbolic execution cannot be applied for more complex models, especially with multiple parts. When composing those different parts, we encounter exponential growth, due to combination of all behaviors. This property become even more apparent in the next section when we discuss the structural abstractions.

8.4 Structural abstractions

In this section we evaluate the usability of structural abstractions. We perform the structural abstraction on the experimental models and we observe the number of states in abstract execution LTS. We also show how structural abstraction can be used to analyze the behavior of a single state machine.

Figure 8.8 presents the results of performing several structural abstractions on model Controller. The experiments took less than 1 second. In the table we indicate which parts were abstracted and we report on number of states in the execution. In the first column we have results of combining structural abstraction with symbolic execution and in the second we have concrete execution, but with values limited to integers between -4 and 4. The more parts we abstract the smaller the state space becomes (in both executions it is similar). The state space reduces more if cars part is abstracted, because the CarLights state machine is larger than that of WalkLights, and it generates more interactions and more execution states. The last row in the table is for abstracting both inner parts <top,walk> and <top,cars>, so with only with the top level part left in the model, that is, the Controller module.
Abstracted parts | Symbolic | Concrete (-4,4)
---|---|---
none | 89 | 183
<top,walk> | 68 | 140
<top,cars> | 41 | 112
<top,walk>, <top,cars> | 32 | 88

Figure 8.8: Number of states in execution LTS of Controller model with different structural abstractions.
We discuss now the structural abstractions of the traffic lights model with 1 intersection when combined with symbolic execution. The results of these experiments are given in Figure 8.9. The execution times for all experiments are less than 1 second. The first row in the table is for the base case if none of the parts are abstracted. In the second row we present the results of abstracting walk parts in both NS and WE parts (these abstracted parts are denoted as $<\text{top,i1,NS,walk}>$ and $<\text{top,i1,WE,walk}>$ in the chart Y-axis). The third row includes similar abstraction for cars parts (denoted as $<\text{top,i1,NS,cars}>$ and $<\text{top,i1,WE,cars}>$ in the chart). As in the case of Controller the state space size decreases more if cars parts are abstracted, for the same reason as previously. The next row indicates that we abstract all parts that have $<\text{top,intersection1,NS}>$ or $<\text{top,intersection1,NS}>$ as their prefix (marked as $<\text{top,i1,NS,*>}$ and $<\text{top,i1,WE,*>}$ in the chart). This effectively means that we abstract all cars and walk parts. In the next row the abstracted parts additionally include $<\text{top,intersection1,NS}>$ and $<\text{top,intersection1,WE}>$, so we abstract parts with the prefix $<\text{top,intersection1}>$. This means that not abstracted parts are $<\text{top}>$ and $<\text{top,intersection1}>$. The last row abstract all parts except for $<\text{top}>$, which has StreetController for 1 intersection as its module.

In the case of the StreetController model with 2 intersections we can perform the abstractions (structural and symbolic execution) similarly and apply them either to the intersection1 part or to both intersection1 and intersection2. We present the results in Figure 8.10. All experiments took less than 1 minute. The first column in the table shows the results with abstracting parts only for the intersection1 part and in the second when abstracting both intersection parts. Obviously fewer parts are abstracted and the state spaces are larger. The difference between abstracting only intersection1 parts and both intersection is not that apparent in case of abstracting walk parts (we have reduced the state space by 12% if all walk parts are considered) but it becomes substantially larger with more parts abstracted. For instance, by adding walk and cars parts of intersection1 we reduce the state space by 56%.

When dealing with the StreetController model with 3 intersections, due to the size of the model, we gathered results only for the case when parts referring to all 3 intersections were abstracted and without the case for walk parts being abstracted. As in the experiments with previous models we combined structural abstraction and symbolic execution. The results are shown in Figure 8.11. The
Abstracted parts | Symbolic
---|---
one | 188
<top,intersection1,NS,walk>, <top,intersection1,WE,walk> | 157
<top,intersection1,NS,cars>, <top,intersection1,WE,cars> | 112
<top,intersection1,NS,*> , <top,intersection1,WE,*> | 86
<top,intersection1,*> | 15
<top,*> | 6

Figure 8.9: Number of states in symbolic execution LTS of StreetController model with 1 intersection with different structural abstractions. i1 stands for intersection1 part and * indicates any parts with the indicated prefix.
### Abstracted parts

<table>
<thead>
<tr>
<th>Abstracted parts</th>
<th>$i = {\text{intersection1}}$</th>
<th>$i = {\text{intersection1, intersection2}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>10 587</td>
<td>10 587</td>
</tr>
<tr>
<td>$&lt;\text{top,}i,\text{NS,walk}, \text{top,}i,\text{WE,walk}&gt;$</td>
<td>9 577</td>
<td>8 353</td>
</tr>
<tr>
<td>$&lt;\text{top,}i,\text{NS,cars}, \text{top,}i,\text{WE,cars}&gt;$</td>
<td>8 265</td>
<td>4 627</td>
</tr>
<tr>
<td>$&lt;&lt;\text{top,}i,\text{NS,<em>}, \text{top,}i,\text{WE,</em>}&gt;$</td>
<td>6 947</td>
<td>3 043</td>
</tr>
<tr>
<td>$&lt;\text{top,}i,*&gt;$</td>
<td>3 117</td>
<td>337</td>
</tr>
<tr>
<td>$&lt;\text{top,*}&gt;$</td>
<td>546</td>
<td>10</td>
</tr>
</tbody>
</table>

**Figure 8.10:** Number of states in symbolic execution LTS of StreetController model with 2 intersection parts with different structural abstractions. $i$ is intersection1 part or intersection1 and intersection2 parts and * indicates any parts with the indicated prefix.
Abstracted parts  

\[
\begin{array}{|c|c|}
\hline
\text{Abstracted parts} & i = \text{all intersection parts} \\
\hline
\langle \text{top}, i, \text{NS}, \text{cars} \rangle, & 69\,022 \\
\langle \text{top}, i, \text{WE}, \text{cars} \rangle & \\
\langle \langle \text{top}, i, \text{NS}, * \rangle, & 51\,757 \\
\langle \text{top}, i, \text{WE}, * \rangle & \\
\langle \text{top}, * \rangle & 4\,791 \\
\text{top}, * & 546 \\
\hline
\end{array}
\]

Figure 8.11: Number of states in execution LTS of StreetController model with 3 intersection parts with different structural abstractions. i stands for all intersection parts and * indicates any parts with the indicated prefix.

Experiments took less than 10 minutes. Even though the initial model is too large for exploration, by removing parts of it we can substantially reduce it.

The interesting and useful application of structural abstraction is to abstract away all parts except for a single part. In such a case a state machine in a module of this left out part is executed as if all the necessary signal were external without any communication with other parts. Below we present the results of this kind of module extracting structural abstraction for modules in StreetController models and in the PBX model subsystems.

First, we extract modules common of all StreetController models. Figure 8.12 presents the sizes of LTS in symbolic and concrete execution (with values between -4 and 4). The execution times
Figure 8.12: Number of states in execution LTS of TrafficLights model with different structural abstractions used to extract a single module.

were less than 1 second. WalkLights has LTS of the same size in both executions, because it does not depend on any external signal with values. For other modules the sizes differ, but in all cases the state spaces are very small.

Module extracting structural abstraction have been applied to StreetController modules with 1, 2 and 3 intersections. The sizes of execution LTS are given in Figure 8.13 for symbolic and concrete execution. Each LTS is larger with each added intersection. The differences between symbolic and concrete executions are not as large as for instance in case of IntersectionController, this is because the external variables with input values are used more and they affect more states in the latter case.

We applied the extraction of modules on several modules of PBX subsystems to show a behavior of modules in symbolic and concrete executions. We gather the results in Table 8.2. All of the presented experiments were done in less than 3 minutes. It can be observed that most of the

<table>
<thead>
<tr>
<th>Module</th>
<th>Symbolic</th>
<th>Concrete(-4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WalkLights</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>CarLights</td>
<td>15</td>
<td>121</td>
</tr>
<tr>
<td>Controller</td>
<td>33</td>
<td>95</td>
</tr>
<tr>
<td>IntersectionController</td>
<td>20</td>
<td>284</td>
</tr>
</tbody>
</table>
### Module | Symbolic | Concrete(-4,4)
--- | --- | ---
StreetController for 1 intersection | 6 | 22
StreetController for 2 intersections | 10 | 26
StreetController for 3 intersections | 22 | 38

Figure 8.13: Number of states in execution LTS of TrafficLights model with different structural abstractions used to extract a single module.
modules have rather small execution $LTS$. The largest one is for \texttt{CellPhoneEventFilter}, because the module models the reception of pressing keys on the phone, and combination of those must be considered. In most cases there is no substantial increase between symbolic and concrete execution, with the exception of \texttt{OrigSession}

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
Module                  & Symbolic & Concrete(-4,4) \\
\hline
\texttt{CallController} & 16       & 40   \\
\texttt{Call}           & 9        & 353  \\
\texttt{OrigSession}    & 13       & 13   \\
\texttt{TermSession}    & 3        & 563  \\
\hline
\texttt{OAMSubsystem}   & 20       & 52   \\
\texttt{GMSC}           & 39       & 103  \\
\texttt{SystemAdminProxy}& 182      & 43 650  \\
\texttt{HLR}            & 41       & 89   \\
\texttt{DeviceManager}  & 3563     & 3571 \\
\texttt{ProxyManager}   & 3563     & 3571 \\
\texttt{CellPhoneEventFilter} & 3563     & 3571 \\
\hline
\end{tabular}
\caption{Number of states in execution $LTS$ of PBX subsystems model with different structural abstractions used to extract a behavior of a single module.}
\end{table}

In this section we showed some of the experiments we performed with the structural abstraction engine. We showed that the application of structural abstraction substantially reduces the size of the state space of the model. This enables easier analysis of models and it supports modular view of them. This is useful especially for large models such as the \texttt{StreetController} model for 3 intersections and PBX subsystems, that is, models for which we could not generate the entire state space. Even for models that large we can reduce the state space to the several hundred states and we can inspect their unconstrained behavior. Even though the behavior is overapproximated, the analysis is still useful, for instance, to detect the unreachable locations. The above experiments also demonstrate that symbolic execution is useful especially for models with only up to 3 capsules. Larger compositions become too complex to handle and to analyze.
8.5 State aggregation

In this section we present results of applying the state aggregation abstraction to our experimental models. We show how this abstraction can be useful to reduce the size of the state space, and we also show that this abstraction may lead to increase in the size of $LTS$.

The results of applying the state aggregation abstraction to the Controller model are given in Figure 8.14 (these experiments took less than 1 second). In the first row of the table we show the size of the state space without aggregation. In the second row we show the results of aggregation of states with the aggregation group including locations responsible for operation of the capsule. These are the states that in the original UML-RT state machine were inside a composite state Operating (see Figure 2.2). This aggregation reduces the number of states by about 40% in both concrete and symbolic execution. This happens, because during the execution many states have the same locations, that is the entire abstraction group, and we detect their similarity. In the last row of the table we see that the number of states may also increase with this type of abstraction, because additional paths are explored. These additional paths are the result of enabling all transitions outgoing from all locations in an aggregation group. Consequently, we may explore paths that are not present in the original state machine.

In Figure 8.15 we gathered the results of applying several state aggregation abstractions to the StreetController model with 1 intersection. All of the shown execution took less than 1 second. In the first row of the table we provide the number of states without any abstraction. In the second row we have a state aggregation with one aggregation group that groups all the operating locations in IntersectionController. Thanks to this abstraction we reduce the number of states by more than 70%. In the next row we also use one aggregation group, this time on initialization locations in the same module. This abstraction does not change the number of states. This happens, because we aggregate states for which triggers need to be present in a respective execution queue. Consequently, even though all outgoing transitions from all aggregated locations are possible to be fired, it is the queue contents that enables transitions actually be fired. In the last row we present an abstraction with 2 aggregation groups that have locations Yellow and YellowRed for both cars parts. This aggregation leads to some additional states with the resulting $LTS$ being larger than the original $LTS$. 
<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Symbolic</th>
<th>Concrete (-4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>89</td>
<td>183</td>
</tr>
<tr>
<td>Operating</td>
<td>55</td>
<td>108</td>
</tr>
<tr>
<td>(Start walk, Start cars, Walk, Cars, Stop walk, Stop cars)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>129</td>
<td>286</td>
</tr>
<tr>
<td>(Starting, Walk ready, Cars ready, Both ready)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.14: Number of states in execution LTS of Controller model with different state aggregation abstractions.


### Table 8.15: Number of states in execution $LTS$ of StreetController model with 1 intersection with different state aggregation abstractions.

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Symbolic</th>
<th>Concrete (-4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>188</td>
<td>862</td>
</tr>
<tr>
<td>Operating in intersection</td>
<td>62</td>
<td>196</td>
</tr>
<tr>
<td>Initialization in intersection</td>
<td>188</td>
<td>862</td>
</tr>
<tr>
<td>Yellow and YellowRed in all cars</td>
<td>222</td>
<td>970</td>
</tr>
</tbody>
</table>

Figure 8.15: Number of states in execution $LTS$ of StreetController model with 1 intersection with different state aggregation abstractions.
 CHAPTER 8. EVALUATION

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Symbolic</th>
<th>Concrete (-4,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>10 867</td>
<td>45 849</td>
</tr>
<tr>
<td>Operating in both intersections</td>
<td>2 807</td>
<td>9 417</td>
</tr>
<tr>
<td>Initialization in both intersections</td>
<td>10 867</td>
<td>45 849</td>
</tr>
<tr>
<td>Yellow and YellowRed in all cars</td>
<td>16 423</td>
<td>65 325</td>
</tr>
</tbody>
</table>

Figure 8.16: Number of states in execution LTS of StreetController model with 2 intersections with different state aggregation abstractions.
Figure 8.16 presents the results of applying state aggregations to the StreetController model with 2 intersections. Running times for all the experiments were less than 1 minute. Aggregation groups used in each case are similar to the ones presented for the StreetController model with 1 intersection shown in Figure 8.15. In the first row of the table we show the number of states in the full LTS. In the next row there are results for two aggregation groups, each of which includes locations responsible for operating an intersection. In this case the abstraction reduces the number of states in the execution. In the next row of the table we have the case in which the number of states remains the same. In the last case we present the results for aggregation with 4 aggregation groups, each of which contains locations Yellow and YellowRed. As in case of 1 intersection, this abstraction results in an execution with more states than the regular LTS.

State aggregation is a useful abstraction to group locations of a state machine, for instance, according to hierarchical compositions of locations in the original UML-RT State Machine. The abstraction may reduce the size of the resulting state space, but it is not guaranteed. It is possible that the size of state space may even increase. Therefore, it is important to group locations which have outgoing transitions triggered by external signals. The state aggregation abstraction is useful in case of complex state machines with a large number of locations.

8.6 Model checking algorithms

In this section we present results of experiments with model checking techniques introduced in Chapter 6. These techniques can be applied to the concrete execution LTS or to any abstractions as indicated in Section 6.3. However, as indicated in this section the abstractions that are overapproximations, that is structural abstraction and state aggregation, require manual checking of feasibility of counterexamples and witnesses. Consequently, this verification technique is not fully automatic and obtained results might not be easily comparable. Therefore, we present only the results of fully automatic model checking. We selected symbolic execution abstraction to perform checking, because execution LTSs are typically smaller than the concrete execution LTSs.

In the following experiments we use symbolic execution rules with subsumption. The presented results include the number of symbolic execution states in the symbolic LTS generated when checking for the satisfaction of a property. Therefore, the resulting numbers are the sizes of state space
necessary to prove or disprove the property. We compare the number of states discovered during the regular model checking and the number of execution states explored during the model checking based on limited exploration.

In Figure 8.17 we present results of checking several properties of the Controller model. All experiments took less than 1 second. In the third column of the table we provide the number of states used in the regular model checking and in the fourth limited exploration. In the first row of the table we use a property to check whether the attribute carsD ever takes a value different than 120, which is desired and should be satisfied. This property is indeed satisfied and it requires 9 states in the regular model checking and 121 during limited exploration. Because the property to check is reached quickly, regular algorithm is very efficient. In the next row we make sure that in all execution paths after both ready we reach a state Stop walk and Stop car, which is what should happen. This property is also satisfied and it requires exploration of only a few states during on-the-fly model checking. Two other properties are also desirable and check whether walk part will move from No walk location to Walk location and whether we reach location Green in all executions require more state space to explore. Finally, we have a property that checks whether we can have lights for cars to be green and lights for pedestrians to be set to walk. This property is clearly dangerous and is not satisfied in the model. However, we need to explore the entire state space to prove it. In this model, because we deal with a small state machine, the limited exploration performs worse, because we need to add extra transitions and states to take care of the pulling elements (see Section 6.4). In case of this model the limited exploration performs also worse than full exploration.

Experiments with StreetController models use the properties gathered in Table 8.3, 8.4 and 8.5. The properties from group 1 refer only to intersection1 and are in Table 8.3. Property 1 describes checking if a transition with any trigger and with the ready signal as an output will be reached on any path, so we are checking if the first intersection becomes ready. The second property (Property 2) checks whether we can reach a state in which both of the attributes of Controller hold a value different than a default. The next property (Property 3) checks whether we will reach a state in which cars in NS is in Green location and walk of WE is in Walk. Property 4 is used to determine whether in all paths cars eventually reach location Green. Finally, with Property 4 we check reachability of an execution state in which cars in both NS and WE are in the location Green. This last property is
### Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Label in chart</th>
<th>Regular</th>
<th>Limited exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EF \ (\text{carsD} \neq 120) \ &amp; \ \langle \text{top} \rangle$</td>
<td>Property 1</td>
<td>9</td>
<td>121</td>
</tr>
<tr>
<td>$AF \ [E \ \text{Both ready} \ &amp; \ \langle \text{top} \rangle \ \cup$ (Stop walk $\ &amp; \ \langle \text{top} \rangle$ OR Stop cars $\ &amp; \ \langle \text{top} \rangle$)]</td>
<td>Property 2</td>
<td>10</td>
<td>136</td>
</tr>
<tr>
<td>$AF \ [E \ \text{No Walk} \ &amp; \ \langle \text{top,walk} \rangle \ \cup$ (Walk $\ &amp; \ \langle \text{top,walk} \rangle$)]</td>
<td>Property 3</td>
<td>54</td>
<td>136</td>
</tr>
<tr>
<td>$AF \ (\text{Green} \ &amp; \ \langle \text{top,cars} \rangle)$</td>
<td>Property 4</td>
<td>54</td>
<td>131</td>
</tr>
<tr>
<td>$EF \ \text{Green} \ &amp; \ \langle \text{top,cars} \rangle$ AND Walk $\ &amp; \ \langle \text{top,walk} \rangle$</td>
<td>Property 5</td>
<td>89</td>
<td>445</td>
</tr>
</tbody>
</table>

---

**Figure 8.17:** Number of states in symbolic execution LTS of Controller used for checking properties.
CHAPTER 8. EVALUATION

<table>
<thead>
<tr>
<th>Properties - group 1</th>
</tr>
</thead>
</table>
| Property 1 (should be satisfied) | AF (any.any()[ready()] @ top.intersection1)  
| Property 2 (should be satisfied) | EF ((carsD != 120 @ top.intersection1.WE AND walkD !=20 @ top.intersection1.NS))  
| Property 3 (should be satisfied) | AF ('Green' @ top.intersection1.NS.cars)  
| Property 4 (should be satisfied) | EF ('Walk' @ top.intersection1.WE AND 'Cars' @ top.intersection1.NS))  
| Property 5 (should not be satisfied) | EF('Green' @ top.intersection1.NS.cars AND 'Green' @ top.intersection1.WE.cars))  

Table 8.3: Properties from group 1 referring to intersection1 used in experiments with the StreetController models.

<table>
<thead>
<tr>
<th>Properties - group 2</th>
</tr>
</thead>
</table>
| Property 1 (should be satisfied) | AF (any.any()[ready()] @ top.intersection1 OR any.any()[ready()] @ top.intersection2)  
| Property 2 (should be satisfied) | EF ((carsD != 120 @ top.intersection1.WE) AND (walkD !=20 @ top.intersection1.NS) AND (carsD != 120 @ top.intersection2.WE) AND (walkD !=20 @ top.intersection2.NS))  
| Property 3 (should be satisfied) | AF ('Green' @ top.intersection1.NS.cars OR 'Green' @ top.intersection2.NS.cars))  
| Property 4 (should be satisfied) | EF ('Walk' @ top.intersection1.WE AND 'Cars' @ top.intersection1.NS AND 'Walk' @ top.intersection2.WE AND 'Cars' @ top.intersection2.NS))  
| Property 5 (should not be satisfied) | EF('Green' @ top.intersection1.NS.cars AND 'Green' @ top.intersection1.WE.cars) OR ('Green' @ top.intersection2.NS.cars AND 'Green' @ top.intersection2.WE.cars))  

Table 8.4: Properties in group 2 referring to intersection1 and intersection2 used in experiments with the StreetController models.
CHAPTER 8. EVALUATION

Table 8.5: Properties in group 3 referring to intersection1, intersection2 and intersection3 used in experiments with the StreetController models.

<table>
<thead>
<tr>
<th>Properties - group 3</th>
<th>AF ((\text{any.any()[ready()] @ top.intersection1 OR any.any() [ready()] @ top.intersection2 OR any.any() [ready()] @ top.intersection3}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1 (should be satisfied)</td>
<td>(AF) (\left('Green' @ top.intersection1.NS.cars OR 'Green' @ top.intersection2.NS.cars OR 'Green' @ top.intersection3.NS.cars\right))</td>
</tr>
<tr>
<td>Property 3 (should be satisfied)</td>
<td>(EF) (\left('Walk' @ top.intersection1.WE AND 'Cars' @ top.intersection1.NS AND 'Walk' @ top.intersection2.WE AND 'Cars' @ top.intersection2.NS AND 'Walk' @ top.intersection3.WE AND 'Cars' @ top.intersection3.NS\right))</td>
</tr>
<tr>
<td>Property 5 (should be satisfied)</td>
<td>(EF) (\left('Green' @ top.intersection1.NS.cars AND 'Green' @ top.intersection1.WE.cars\right) OR (\left('Green' @ top.intersection2.NS.cars AND 'Green' @ top.intersection2.WE.cars\right) OR (\left('Green' @ top.intersection3.NS.cars AND 'Green' @ top.intersection3.WE.cars\right))</td>
</tr>
</tbody>
</table>

what we want to avoid and it should not be satisfied. Properties in group 2 and 3 additionally refer to intersection2 and intersection3, respectively. The exception is that for all 3 intersection there is no Property 2, since with 3 intersections controller does not change the attributes.

Figure 8.18 presents the results of checking the properties in the StreetController model with 1 intersection. The running time for all experiments is less than 30 seconds. In the case of Property 1 the positive result is obtained very quickly in both regular and limited exploration checking. The second property, i.e., Property 2, is also checked quickly in regular checking but it takes more states in the limited checking, because of extra transitions and states generated to perform pulling operations. The same happens for all other properties. For regular checking of properties we need to explore more states with each property, up until the full state space is checked for the final property. The Limited exploration is different as it depends more on the initial parts set, for instance we need to explore more states in checking of Property 2 than in checking of Property 3. This becomes even more apparent in the case of checking StreetController with 2 intersections.
Properties (group 1 in Table 8.3)

<table>
<thead>
<tr>
<th>Property</th>
<th>Regular</th>
<th>Limited exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>14</td>
<td>29</td>
</tr>
<tr>
<td>Property 2</td>
<td>27</td>
<td>164</td>
</tr>
<tr>
<td>Property 3</td>
<td>41</td>
<td>133</td>
</tr>
<tr>
<td>Property 4</td>
<td>120</td>
<td>570</td>
</tr>
<tr>
<td>Property 5</td>
<td>188</td>
<td>1 136</td>
</tr>
</tbody>
</table>

Figure 8.18: Number of symbolic states explored during checking properties from Table 8.3 of StreetController with 1 intersection.
Figure 8.19 presents the results of experiments with the StreetController model with 2 intersections. All experiments were performed in less than 10 minutes. As in case of the previous model both Property 1 and Property 2 from both groups are checked quickly and only a small number of states are explored. In case of Property 3 from group 1 (i.e., referring to 1 intersection) we explore more states than when checking Property 4. This relation is reversed for properties in group 2 (i.e. when \texttt{intersection1} and \texttt{intersection2} are used). This is because the version of Property 3 in group 2 checks for states that are in \texttt{Green} location in the first or in the second intersection, whereas the group 2 version of Property 4 looks for states that satisfy its subformula in the \texttt{intersection1} part and \texttt{intersection2}. Obviously, using conjugation as in Property 3 is more constraining than using disjunction as in Property 2. For the StreetController with 2 intersections the limited exploration is more efficient, especially when checking for the last three properties in group 1. This is because the checking is limited only to one or two parts in the model that contains a total of 8 parts. Therefore, even with the extra pulling states we are able to limit the exploration compared to the exploration involving all 8 parts. This holds also for the properties that require exploration of the substantial portion of state space such as Property 4 or 5. Even though Property 5 in group 1 requires the full exploration in regular checking, we can avoid that in the limited exploration. Interestingly, we do not reduce the state space that much for properties in group 2, that is if properties refer to larger number of parts. This effect is especially evident for Property 4 and Property 5 (group 2). To check them need to explore significantly more states than during the regular exploration.

In case of the StreetController model with 3 intersections we present the results of checking based on limited exploration only, since the model is too large to perform regular on-the-fly model checking, except for the first property. The experiments took up to 24 hours. In case of checking properties from group 1 all properties are checked and the largest state space is around 4000 states. This demonstrates that limited exploration is very efficient if number of referred parts is small compared to all parts in the model. When checking properties from group 3, that is, if all 3 intersections are used, the results are much worse. Property 4 and Property 5 could not be checked, because we encountered \texttt{OutOfMemoryException}. Property 3 was checked and the results were obtained after exploration of a small part of the state space.

Finally, we present the results of checking properties in the PBX model subsystems. These
CHAPTER 8. EVALUATION

<table>
<thead>
<tr>
<th>Properties</th>
<th>group 1 (Table 8.3)</th>
<th>group 2 (Table 8.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regular</td>
<td>Limited exploration</td>
</tr>
<tr>
<td>Property 1</td>
<td>14</td>
<td>29</td>
</tr>
<tr>
<td>Property 2</td>
<td>136</td>
<td>212</td>
</tr>
<tr>
<td>Property 3</td>
<td>6 886</td>
<td>626</td>
</tr>
<tr>
<td>Property 4</td>
<td>4 589</td>
<td>1 098</td>
</tr>
<tr>
<td>Property 5</td>
<td>10 587</td>
<td>2 726</td>
</tr>
</tbody>
</table>

Figure 8.19: Number of symbolic states explored during checking properties from group 1 and group 2 (Table 8.3 and 8.4) of StreetController with 2 intersections (Y-axis is logarithmic with base 10).
Figure 8.20: Number of symbolic states explored during checking properties from group 1 and group 3 (Table 8.3 and Table 8.5) in the StreetController with 3 intersections (Y-axis is logarithmic with base 10).
results, i.e., the numbers of execution states are shown in Table 8.6, with > indicating that the checking stopped after an `OutOfMemoryException`. In case of limited exploration the execution times of experiments were less than 20 minutes, the regular exploration has took up to 24 hours. In most of the cases we obtained results only for the limited exploration technique. This is because we selected properties that mention limited number of parts. Otherwise, that is, with more execution parts the checking could not be finished for either technique. In case of two properties

\[ AF(\text{WaitingForDevice} @ \text{top.gMSC}) \]

and

\[ AF((\text{activeSession} > 0 @ \text{top.device1 OR activeSession} > 0 @ \text{top.device2})) \]

the result of checking is negative, that is, we discover a cycle in which both subformulas are not satisfied. Since the cycles are detected early, both checking techniques perform well. The property of `ProxyManager`

\[ EF((\text{dialed1} >= 0 @ \text{top.proxy1 AND dialed2} >= 0 @ \text{top.proxy1})) \]

is satisfied early and get the result quickly. In case of other properties limited exploration is much more beneficial, because we avoid exploring interactions with execution parts that are not necessary. The results of limited exploration vary, and the necessary exploration is between 500 and 15000 states.

In this section we provide some evaluation of verification methods by comparing on-the-fly model checking and model checking with limited exploration. The benefits of this second technique are especially apparent in case of large models and when properties mention a small number parts in these models. We could observe this especially for `StreetController` model with 2 and 3 intersections and for PBX subsystems. In this last model on-the-fly model checking could not give us results, whereas limited exploration performed well. However, if more parts are mention in the formula we might not gain much with limited exploration. This has been observed in case of checking properties that mentioned all intersections in `StreetController` with 2 intersections. In case of checking properties involving more parts verification does not complete with either method.
### Modules and properties

<table>
<thead>
<tr>
<th>Modules and properties</th>
<th>Regular</th>
<th>Limited exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CallController</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| $\text{EF}('\text{Connected}'@\text{top.call1}.\text{termSession}$ \&
  $'\text{Connected}'@\text{top.call2}.\text{termSession}$) | $>71\ 406$ | $6\ 640$ |
| can we reach a state with calls having terminal session connected? | | |
| $\text{EF}('\text{CallProceeding}'@\text{top.call1})$ | $>249\ 917$ | $505$ |
| is it possible that call1 has a call in progress? | | |
| **OAMSubsystem**       |         |                     |
| $\text{EF}($extensionNumbers== 4 @ top.hlr$)$ | $>45\ 708$ | $936$ |
| can we have 4 extensions in HLR? | | |
| $\text{EF}($any[extensionLookup.sessionOk()] @ top.hlr.phoneExtension1$)$ | $>46\ 727$ | $2\ 693$ |
| can the extension 1 an session ok signal? | | |
| $\text{AF}('\text{WaitingForDevice}'@\text{top.gMSC})$ | $233$ | $11$ |
| can GMSC always wait for the device? | | |
| **DeviceManager**      |         |                     |
| $\text{EF}($numPhones == 4 @ top$)$ | $>108\ 353$ | $2\ 932$ |
| can we have 4 phones? | | |
| $\text{AF}($(activeSession > 0 @ top.device1 OR
  activeSession > 0 @ top.device2$)$) | $70$ | $201$ |
| can we always reach a state with active sessions on device 1 and 2? | | |
| **ProxyManager**       |         |                     |
| $\text{EF}($dialed1 >= 0 @ top.proxy1 AND dialed2 >= 0 @ top.proxy1$)$ | $109$ | $13$ |
| can we reach a state in which the first 2 dialed numbers are selected? | | |
| $\text{EF}($‘Connected’ @ top.proxy2 AND ‘Connected’ @ top.proxy1$)$ | $>59\ 637$ | $15\ 467$ |
8.7 Evaluation conclusions

In order to conclude the evaluation we are going to recall the questions we started this section and give some answers to them.

1. Can the proposed abstractions improve the analysis and understanding of UML-RT models? On which types of models and analyses do the abstractions provide the largest benefit?

   We show that abstractions typically reduce the sizes of the state space of the analyzed model. This is true in case of symbolic execution, structural abstraction and, in some cases, state aggregation. Reducing the size of the state space enables visualization of executions and we can inspect execution paths. Also, we can improve understanding of models by providing different views of executions based on abstractions, for instance, to check the interactions between selected execution parts. Therefore, the scenarios which we see the improvement in understandings include executions which can be visualized.

2. Are the proposed verification methods usable, that is, can we check for important and interesting properties of models?

   We propose verification with CTL formulas and we show that we can check for important safety and for liveness properties in CFFSMs models. Especially for larger models, for which we cannot visualize the execution LTS, checking for selected properties is an important tool to help understanding. And because checking is exhaustive, the properties are verified.

3. What is the scalability of the proposed methods?

   We compare scalability of most of the proposed methods. Symbolic execution is best suited for models with only few execution parts, and beyond that executions are not manageable. Structural abstraction is very powerful in reducing the size of the state space and it may reduce the model to only few parts, for which symbolic execution works. State aggregation is not as powerful but it may also simplify executions. When it comes to verification we show that especially limited exploration model checking is useful for large models, but only if the checked properties refer only to small number of execution parts.

4. Can the proposed approach be applied on industrial models?

   We performed the experiments on the customized industrial models. Our tool supports most
of the features, but some customization is still necessary, especially to deal with action code. We are able perform structural abstractions on the models and we also are able to perform verification of some properties. However, we cannot claim that the implementation and the proposed methods can be applied to any UML-RT model.

8.8 Summary

In this section we provide the evaluation of the proposed approach. We identify the questions that the evaluation should answer and we show the steps to answer the questions. The questions included usability of the methods in the context of verification and analysis as well as scalability. In our experiments we use the number of the execution states to quantify the complexity of the analyses.

The first step of the evaluation is to implement models to perform experiments with. We used three sets of such models. The first one, was simply the example model of traffic lights we used in the earlier chapter. The second set builds on this simple model and includes the models with increasingly complex structure. This is achieved by adding extra parts. Finally, we use the model that we obtained from our industrial partner and which models a Private Branch Exchange System (PBX). In this model there are 4 subsystems distinguished and these are used for evaluation.

In order to evaluate abstractions we compare the sizes of abstract executions measured with the number of execution states. We compared the symbolic execution and the concrete execution with domains reduced to only several values. We show that for open models, that is, models with external signals, we can use symbolic execution and substantially reduce the size of the state space. Structural abstraction can also reduce the size of the state space and it allows to focus on parts that are important during analysis. Finally, we check state aggregation and we note that it may not always simplify the state space. However, it is very beneficial for complex state machines.

We also performed the evaluation of model checking algorithms. In case of models with limited number of states the regular model checking performs reasonably well. This changes for larger models. In case of execution state spaces with more than 100K od states, we cannot get any results. In such cases we can use limited exploration, but only for properties that mention small number of parts. When the property refers to many parts, we have similar state space explosion problems as
in case of the regular on-the-fly algorithm.

Finally, in the conclusions of the evaluation part we recall the questions and we provide the answers to them.
Chapter 9

Conclusions and future work

In this thesis we approach the problem of analysis and verification of MDD models in a language (i.e., UML-RT) specific way. The majority of works existing general purpose in this area provide methods that rely on transformation of models to input languages of existing tools, e.g., SPIN. The disadvantage of these approaches is that the proposed transformations become complex and do not include important features of models. Unlike translational approaches we propose language-specific analysis and verifications, which brings them closer to the MDD language.

As an example of a complex modeling language we use the UML-RT modeling language as one of the languages supporting development of embedded and reactive systems. The language is modular with hierarchical components having behavior expressed using state machines with action code. For this language we provide a formal representation in the form of Communicating Functional Finite State Machines (CFFSMs). This representation supports important features of UML-RT, such as hierarchical structures, asynchronous communication and basic action code.

Based on the CFFSMs we define several abstractions, that is, alternative rules for execution that can provide a more compact view of an execution of a model. Such a view, may improve understanding of a model and enable verification. We identify three kinds abstractions for CFFSMs. The first one is symbolic execution, which deals with the values of variables used in a model. If necessary such values are represented with symbols. We use constraints on those symbols to distinguish between paths of execution. The second type of abstraction is structural abstraction, which allows the executions of models with missing parts. The third type of abstraction is state aggregation. In this abstraction we group locations in a state machine and we treat them as if it were one location. For
CHAPTER 9. CONCLUSIONS AND FUTURE WORK

each abstraction we define rules that govern the abstract execution. We also demonstrate relations between abstractions and concrete execution. We demonstrate that symbolic execution is precise, which means that there is a one-to-one correspondence between abstract and concrete paths. Structural abstraction as well as state aggregation are overapproximations, meaning that we have some abstract paths that cannot be mapped to concrete ones.

Besides abstractions, we also provide verification techniques for temporal properties. These properties are Computation Tree Logic formulas with atomic propositions describing states of execution: being in a location, having constraints on attributes satisfied, having certain signals in queues and having certain signals as input or output on a transition. We provide two algorithms that can check CTL formulas. The first one is on-the-fly, which takes advantage of step by step execution and labels states during the exploration. The second algorithm is based on the modularity of models and limits the exploration to the modules that can affect the satisfaction of the formula that is being checked.

The approach proposed in this work is implemented in the toolset called TUMLAV. The toolset consists of a transformation and rule based engine. The transformation has two phases: ATL transformation of structural elements and functions generation. The latter step uses symbolic execution and produces the constraint-based representation of action code present on transitions and as entry or exit actions in states. The output from the transformation phases is a CFFSM model, which is an input to the rule based engine. The rule based engine is responsible for executing CFFSMs using a set of rules. This set of rules defines the execution and we have a different set for the concrete execution and for each type of abstraction.

We performed experiments with TUMLAV using our running example and two sets of models. The first set is developed to provide models with increasing complexity, which we achieved by adding more internal parts. The second set of models is an adaptation of a model that we received from our industrial partner. We use 4 subsystems of this model in our experiments. In the experiments we measure the sizes of explored state spaces. First we compare concrete and symbolic execution and, as expected, symbolic execution can reduce the size of the state space in models that receive signals with data from the environment. However, for models with more than several parts, symbolic execution may produce state spaces that are too large to handle. In order to minimize the number of parts we use structural abstraction. Thanks to this abstraction we can reduce the number of explored
states from more than 100K to several hundreds, which makes executions possible to visualize. State aggregation abstraction is not as effective for state space reduction, but is useful in case of complex, hierarchical state machines. We also perform experiments with model checking in both versions of algorithms. On-the-fly model checking is useful for models with up to 100K states in the execution. For larger models and properties with only several parts mentioned limited exploration checking is effective.

There are several future directions of the work:

1. The first and most obvious direction of is to extend the support for more advanced features of UML-RT models. These features include structural elements, such as plugin parts, dynamic binding between ports or multiple parts and ports. Another beneficial way of extending the support is to include more advanced features of action code, such as more complex data types, method calls and more comprehensive support for loops including loop invariants. Extending the support for other features of UML-RT, will allow to analyze a larger set of UML-RT models, also more complex ones. The support for other modeling languages similar to UML-RT may also be possible.

2. The direction inspired by pragmatic considerations is to support our abstractions and verification techniques in the context of the original UML-RT models, not in the context to CFFSMs. For instance, if we want to abstract an execution part, we want to be able to do that in for parts in the original UML-RT models. This direction requires mostly effort in the implementation.

3. Another direction to extend symbolic execution is to combine it with concrete execution, similar to concolic testing and DART [53]. In the context of our approach we can achieve it with different generation of values for variables on input, external signals. Some of the signals would have variables generated with concrete values (e.g., randomly selected) and some of the signals would get symbolic values and use those values later on during the execution. By having some of the variables concrete, we can minimize the use of constraints and simplify checks for satisfiability with SAT solvers, which are computationally expensive. Similarly to DART we can explore all branches created with guards and case conditions. This would allow to perform reachability checking, i.e., whether certain locations are reachable or not.
4. Future research could also consider other types abstractions. For instance, we can define abstractions that simplify the contents of queues by providing more complex ways to describe their similarity. In this way we can abstract out some of the paths that are the result of the different ordering of elements in queues or the presence of multiple elements. Another type of abstraction to consider is to make some input signals always available. Similarly to what we assumed in structural abstraction and signals received from abstracted parts, we can extend it to other signals. In this way we can fire more transitions without waiting for synchronization with other parts.

5. In the current state of the art of model checking there exists a lot of optimization possibilities, such as BDDs or partial order reduction. We wish to use these techniques in the presented method. There are two ways to achieve it. The first one is to adapt them appropriately and then implement them directly in the model checker and made them suited for our consideration. The second approach is to translate an execution LTS to an input language of one of existing model checkers. Note that we propose to transform the execution LTS, not the original UML-RT models. In this way we could still leverage the presented languagespecific approach to analysis and deal only with transformation of the resulting execution LTS.

6. Another interesting technique to optimize model checking is counter example guided abstraction refinement [33], which uses counter examples to decide which elements should be refined to provide a more detailed view of the execution of the model. In case of CFFSMs we can start checking with the limited number of parts and use the feasibility of counter examples to check which parts, previously abstracted, should be added to the execution. In this way each check would use only the necessary execution parts. A similar strategy can be used to refine aggregated states, that is, counter-examples could be used to decide which aggregation groups to split.

7. Finally, the proposed approach relies very much on modularity of models. An interesting avenue for future research would be to apply modular model checking techniques such as assume guarantee methods [37]. The most challenging part of these methods is to find a succinct way to represent properties of modules and to select the ones which are appropriate
to do the analysis. In the context of state machines this might be difficult task, since state machines are possibly complex themselves. However, checking properties of each machine separately is much more effective than checking the entire model. Consequently, we believe that this direction of future research is also very promising.
Bibliography


[61] IBM. *IBM Rational Rhapsody*. 


Appendix A

Ecore meta-model of CFFSMs

This appendix presents the Ecore meta-model of CFFSMs. It is shown in two parts. The first one for the structural elements and the second for State machines.
Figure A.1: Ecore meta-model of structural elements in CFFSMSs.
Figure A.2: Ecore meta-model of state machine elements in CFFSMs.
Appendix B

TUMLA

In this appendix we present several screenshots from our tool TUMLA. Figure B.1 shows an example of a symbolic execution LTS with couple of its states visible. In the outline tab (left bottom) we can observe the general shape of the LTS. There is also a navigator through CFFSMs models (left top) and a view with properties of the selected states (bottom). Figure B.2 shows a screenshot when execution parts are selected to be abstracted out and Figure B.3 is a screenshot when we select locations to be aggregated.
Figure B.1: A screenshot from TULMAV- a symbolic LTS.
Figure B.2: A screenshot from TUMLAV- selection of execution parts to abstract.
Figure B.3: A screenshot from TULMAV- selection of locations to be aggregated.