Design and Verification of Concurrent Real-Time Systems using SDL and MSC


Yves Bourgeois
Design and Verification of Concurrent Real-Time Systems using SDL and MSC

Master Thesis

submitted in partial fulfillment of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

in

SOFTWARE ENGINEERING

by

Yves Bourgeois
born in Wilrijk, Belgium

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
Abstract

This thesis examines both the shortcomings of the Formal Description Languages SDL and MSC, when they are used to design hard real-time systems, and the lack of verification tools to verify the timing requirements of such hard real-time systems. In the first part of this document we will discuss that SDL and MSC do not have a proper model of time, or a formal way to express all aspects of real-time systems. Therefore, we propose a set of semantic rules, which make it possible to formally describe these timing requirements. In the second part, we will discuss how we can statically verify the temporal properties of the SDL and MSC model, using the UPPAAL verification tool, together with a case study that proves the correctness of the proposed method.
Acknowledgements

I wish to thank Hans-Gerard Gross, my mentor, for the opportunity to do my thesis project under his guidance. I thank him for always being ready to help me, if I had questions, and to numerosly read and valuably comment the many drafts I have sent him.

I also wish to thank my classmates. Carsten Krauter, who I could disturb any part of the day, with numerous questions and odd remarks, Dimos Mpekas, to help me understand how a path finding algorithm exactly works, and Joseph Scheuer, who fittingly reminded me about the evil of global variables.

I thank Maik Renner, who was willing to spend numerous evenings reading my final dissertation, whilst interlacing it with valuable comments.

I also wish to thank my friends and family, for the valuable support they gave me, and for not interfering too much.
# Contents

## Acknowledgements

ii

## Contents

iii

## List of Figures

vii

1  **Introduction**  

1

2  **The Specification and Description Language (SDL)**  

5  

   2.1  SDL: a short introduction  

   2.1.1  SDL: the basics  

   2.1.2  SDL and the Development of Real-Time Systems  

   2.1.3  Timed automata  

   5  

   2.2  SDL, time and clocks  

   2.2.1  Formal specification of clocks  

   8  

   2.3  Urgencies  

   10  

   2.4  Annotations in SDL  

   2.4.1  Clocks and derived counters  

   2.4.2  Urgencies  

   2.4.3  Events  

   2.4.4  Duration of events  

   2.4.5  Channels  

   2.4.6  Periodicity of external input  

   12  

   2.5  Summary  

   17  

3  **Message Sequence Charts**  

19  

   3.1  Expressing time requirements by extending MSC  

   3.1.1  How to extend MSC  

   3.1.2  Time and timers  

   3.1.3  HMSC related time requirements: instance delay  

   3.1.4  Time requirements of consecutive events: time intervals  

   21  

   3.2  Message delivery interval  

   26
<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor speed interval</td>
<td>27</td>
</tr>
<tr>
<td>3.1.5 Time requirements of not-consecutive events</td>
<td>28</td>
</tr>
<tr>
<td>Precedence edges</td>
<td>28</td>
</tr>
<tr>
<td>Absolute time</td>
<td>29</td>
</tr>
<tr>
<td>3.1.6 The synchronous hypothesis</td>
<td>29</td>
</tr>
<tr>
<td>3.2 Summary</td>
<td>32</td>
</tr>
<tr>
<td>4 Static Verification of the SDL model</td>
<td>35</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Why can we use timed automata?</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Timed automata: a definition</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 The timed automaton as a labelled transition system</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Selection of tools</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 SDL design tool: Real-Time Developer Studio</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Selection of verification tools</td>
<td>39</td>
</tr>
<tr>
<td>Real-Time Spin</td>
<td>40</td>
</tr>
<tr>
<td>Kronos</td>
<td>41</td>
</tr>
<tr>
<td>UPPAAL</td>
<td>42</td>
</tr>
<tr>
<td>UPPAAL: the tool of my choice</td>
<td>43</td>
</tr>
<tr>
<td>4.4 Parsing the SDL model to its counterpart in UPPAAL</td>
<td>44</td>
</tr>
<tr>
<td>4.5 Mapping SDL to UPPAAL</td>
<td>44</td>
</tr>
<tr>
<td>4.5.1 Timing requirements defined in RTDS and UPPAAL</td>
<td>44</td>
</tr>
<tr>
<td>The system clock</td>
<td>45</td>
</tr>
<tr>
<td>Urgencies</td>
<td>47</td>
</tr>
<tr>
<td>Event</td>
<td>48</td>
</tr>
<tr>
<td>Duration</td>
<td>48</td>
</tr>
<tr>
<td>Channel delay</td>
<td>48</td>
</tr>
<tr>
<td>Span</td>
<td>49</td>
</tr>
<tr>
<td>Periodicity of external input</td>
<td>50</td>
</tr>
<tr>
<td>4.5.2 Parsing from SDL to UPPAAL: SDL2UPPAAL</td>
<td>52</td>
</tr>
<tr>
<td>File structure of an RTDS project</td>
<td>52</td>
</tr>
<tr>
<td>RTDS vs. UPPAAL: the internal xml structure</td>
<td>52</td>
</tr>
<tr>
<td>Translating the RTDS model to a UPPAAL model: SDL2UPPAAL</td>
<td>55</td>
</tr>
<tr>
<td>Translating the timing requirements to their UPPAAL counterpart</td>
<td>60</td>
</tr>
<tr>
<td>4.6 Test results</td>
<td>67</td>
</tr>
<tr>
<td>4.6.1 The test case</td>
<td>67</td>
</tr>
<tr>
<td>4.6.2 Verification</td>
<td>69</td>
</tr>
<tr>
<td>Verifying spanAtoB.xml</td>
<td>69</td>
</tr>
<tr>
<td>Verifying spanAtoC.xml</td>
<td>70</td>
</tr>
<tr>
<td>4.7 Conclusion</td>
<td>71</td>
</tr>
<tr>
<td>5 Static Verification of the MSC Model</td>
<td>74</td>
</tr>
<tr>
<td>5.1 Time consistency in bMSC and MSC specifications</td>
<td>74</td>
</tr>
<tr>
<td>5.1.1 Consistency of temporal properties of a single bMSC</td>
<td>74</td>
</tr>
<tr>
<td>Consistency analysis using the distance graph</td>
<td>74</td>
</tr>
<tr>
<td>Consistency analysis using the UPPAAL tool</td>
<td>75</td>
</tr>
</tbody>
</table>
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td></td>
</tr>
<tr>
<td>Depth first search: how it works</td>
<td>104</td>
</tr>
<tr>
<td>Random depth first search</td>
<td>105</td>
</tr>
<tr>
<td>Pitfalls</td>
<td>106</td>
</tr>
<tr>
<td>D SDL2UPPAAL</td>
<td>109</td>
</tr>
<tr>
<td>D.1 The function parse_process()</td>
<td>109</td>
</tr>
<tr>
<td>D.1.1 Function parse_state_symbol()</td>
<td>110</td>
</tr>
<tr>
<td>D.2 Parse the decision transitions</td>
<td>111</td>
</tr>
<tr>
<td>D.2.1 Function parse_decision_transitions()</td>
<td>111</td>
</tr>
<tr>
<td>D.2.2 Function determineTops()</td>
<td>112</td>
</tr>
<tr>
<td>D.2.3 Function findPathForAllTops()</td>
<td>113</td>
</tr>
<tr>
<td>D.3 Function parse_normal.transitions()</td>
<td>115</td>
</tr>
<tr>
<td>E Extending the model parser to support timing requirements</td>
<td>116</td>
</tr>
<tr>
<td>E.1 Extending function parse_process()</td>
<td>116</td>
</tr>
<tr>
<td>E.2 Function parse_comment()</td>
<td>117</td>
</tr>
<tr>
<td>E.3 Parsing timing information and timing requirements</td>
<td>117</td>
</tr>
<tr>
<td>E.3.1 Extending function SDL2UPPAAL()</td>
<td>117</td>
</tr>
<tr>
<td>E.3.2 Function add_urgencies()</td>
<td>118</td>
</tr>
<tr>
<td>E.3.3 Function add_durations()</td>
<td>118</td>
</tr>
<tr>
<td>E.3.4 Function add_span_paths()</td>
<td>118</td>
</tr>
<tr>
<td>Function add_path()</td>
<td>119</td>
</tr>
<tr>
<td>E.4 Function add_updates()</td>
<td>120</td>
</tr>
<tr>
<td>F SDL2UPPAAL: The Test Case</td>
<td>121</td>
</tr>
<tr>
<td>G MSC2UPPAAL</td>
<td>127</td>
</tr>
<tr>
<td>G.1 The function parse_bmsc()</td>
<td>127</td>
</tr>
<tr>
<td>G.2 Creating the automaton</td>
<td>128</td>
</tr>
<tr>
<td>G.2.1 the function create_automaton()</td>
<td>128</td>
</tr>
<tr>
<td>G.2.2 the function create_automaton_recursive()</td>
<td>129</td>
</tr>
<tr>
<td>H MSC2UPPAAL: The test case</td>
<td>130</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 A simple SDL diagram: processes, blocks, signal routes and channels [3] 6  
2.2 Graphical representation of communication paths 7  
2.3 Start and stop symbol 7  
2.4 State, input and output symbol 7  
2.5 Three annotation methods for non-functional requirements 13  
2.6 A clock definition in SDL 13  
2.7 Urgencies in SDL: eager transitions 14  
2.8 Duration in SDL 16  
2.9 Channels in SDL 17  
2.10 External channels in SDL 17  
3.1 A simple bMSC 20  
3.2 Building blocks of an hMSC 20  
3.3 hMSC 21  
3.4 An unconnected timer 22  
3.5 A connected timer 22  
3.6 Example of connected and unconnected timer symbols in practice 24  
3.7 Repetitive behaviour in an hMSC 25  
3.8 How to specify instance delay in a bMSC 25  
3.9 Event associated timing interval: closing of a valve within 200 ms 26  
3.10 Message delivery interval: message passing on a physical medium 27  
3.11 Example of a processor speed interval 28  
3.12 Precedence edges 29  
3.13 Timing constraints on two unrelated events 30  
3.14 Absolute time 30  
3.15 A system with react delay 31  
3.16 A system according to the synchronous hypothesis 31  
3.17 Temporal correct and incorrect MSCs 33  
3.18 Duration of events in bMSCs 34  
4.1 Comparison of the RTDS and the UPPAAL representation of a model 45  
4.2 A clock definition example 46
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Representation of a lazy state in both RTDS and UPPAAL</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Representation of an urgent state in both RTDS and UPPAAL</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Representation of a delayable state in both RTDS and UPPAAL</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Representation of duration in both RTDS and UPPAAL</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Representation of channel delay in both RTDS and UPPAAL</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>Timing requirements in RTDS</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Timing requirements in UPPAAL</td>
<td>51</td>
</tr>
<tr>
<td>4.10</td>
<td>Parsing decision transitions</td>
<td>58</td>
</tr>
<tr>
<td>4.11</td>
<td>The graph to be explored</td>
<td>63</td>
</tr>
<tr>
<td>4.12</td>
<td>Breadth First Search and its queue after one round</td>
<td>63</td>
</tr>
<tr>
<td>4.13</td>
<td>BFS and its queue after two rounds</td>
<td>64</td>
</tr>
<tr>
<td>4.14</td>
<td>Path finding with loops</td>
<td>64</td>
</tr>
<tr>
<td>4.15</td>
<td>A problem during verification</td>
<td>66</td>
</tr>
<tr>
<td>4.16</td>
<td>The environment</td>
<td>70</td>
</tr>
<tr>
<td>4.17</td>
<td>Verification using UPPAAL</td>
<td>70</td>
</tr>
<tr>
<td>4.18</td>
<td>Verification using UPPAAL</td>
<td>70</td>
</tr>
<tr>
<td>4.19</td>
<td>Verification when $\text{correctIrisScan}$ is 1</td>
<td>71</td>
</tr>
<tr>
<td>4.20</td>
<td>Verification when $\text{correctIrisScan}$ is 0</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>From bMSC to the distance graph</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>A simple bMSC of an automatic door</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>The automaton representation of Figure G.1</td>
<td>81</td>
</tr>
<tr>
<td>5.4</td>
<td>The path from event $\text{closed}$ to event $\text{Closed}$</td>
<td>83</td>
</tr>
<tr>
<td>A.1</td>
<td>The Software Development Cycle [3]</td>
<td>95</td>
</tr>
<tr>
<td>C.1</td>
<td>Two states with a simple transition</td>
<td>98</td>
</tr>
<tr>
<td>C.2</td>
<td>Urgencies of locations</td>
<td>99</td>
</tr>
<tr>
<td>C.3</td>
<td>A location with an invariant</td>
<td>99</td>
</tr>
<tr>
<td>C.4</td>
<td>Transitions in UPPAAL</td>
<td>100</td>
</tr>
<tr>
<td>C.5</td>
<td>The UPPAAL model and its DBM</td>
<td>101</td>
</tr>
<tr>
<td>C.6</td>
<td>The graph to be searched</td>
<td>103</td>
</tr>
<tr>
<td>C.7</td>
<td>BFS: search space after one step</td>
<td>103</td>
</tr>
<tr>
<td>C.8</td>
<td>BFS: search space after two steps</td>
<td>103</td>
</tr>
<tr>
<td>C.9</td>
<td>BFS: search space after three steps</td>
<td>104</td>
</tr>
<tr>
<td>C.10</td>
<td>DFS: search space after one step</td>
<td>104</td>
</tr>
<tr>
<td>C.11</td>
<td>DFS: search space after two steps</td>
<td>105</td>
</tr>
<tr>
<td>C.12</td>
<td>DFS: search space after four steps</td>
<td>105</td>
</tr>
<tr>
<td>C.13</td>
<td>Search tree with cycles</td>
<td>107</td>
</tr>
<tr>
<td>C.14</td>
<td>DFS: search space after some steps</td>
<td>107</td>
</tr>
<tr>
<td>F.1</td>
<td>The SDL model of the secure automatic door</td>
<td>122</td>
</tr>
<tr>
<td>F.2</td>
<td>The console after we ran the SDL2UPPAAL parser</td>
<td>123</td>
</tr>
<tr>
<td>F.3</td>
<td>The path going from event A to event B</td>
<td>124</td>
</tr>
<tr>
<td>F.4</td>
<td>The path going from event A to event C</td>
<td>125</td>
</tr>
<tr>
<td>F.5</td>
<td>Example of a path for which the TCTL formula is invalid</td>
<td>126</td>
</tr>
<tr>
<td>H.1</td>
<td>A scenario: successful opening and closing of the door</td>
<td>131</td>
</tr>
</tbody>
</table>
H.2 The path from event A to event B ................................. 132
Chapter 1

Introduction

In real-time systems, the correctness of the system will not only depend on the correctness of its computations, but also on its temporal correctness. Or in other words, the time at which a computation finishes, its deadline, is of equal importance as the correctness of its result. We can identify two different types of deadlines: hard deadlines and soft deadlines. Hard deadline systems, also called safety-critical systems, are systems in which the loss of a deadline could mean the loss of money, and even the loss of lives, which is clearly unacceptable. Soft deadline systems are systems in which a deadline might be missed, but is preferred not to. Here, the system will recuperate from the loss of the deadline, without any harm done. In the course of the thesis, we will only regard hard deadline systems, although the methods presented will also be applicable on soft deadline systems. The reason for this is that hard deadline systems have stricter temporal requirements. If these requirements are verified to be correct, then surely their weaker counterparts will also be verified as correct.

It is clear now, that the verification of the temporal requirements of real-time systems is of great importance, to ensure that, for each run of the system, all deadlines are met, at all cost. It is even so that, during the design of major real-time systems in the industry, over 50% of the development time is spent on the verification of these systems[56].

The verification techniques of today mostly rely on manual verification and testing of the system, e.g. with the help of prototypes. With these techniques, the tester needs to identify all possible simulation traces of the system at hand, and all these traces need to be assessed against both the functional and the non-functional requirements. Under the regime of manual verification and testing, development can typically only be performed in small iterations, evolving an existing system and assessing the changes. However, this technique is too much of a burden to prove the correctness of the temporal requirements of a real-time system, since the complexity of timed systems is exponentially more complex than the complexity of untimed systems[56]. This brings with it, that it becomes very hard to manually find all traces of the system execution, to provide sufficient coverage of its state space[56]. Because of this, verification becomes not reliable anymore, in that some system executions might never have been tested, while these executions could violate one or more temporal requirements. Solutions for this problem have been found in e.g. evolutionary or genetic algorithms, which will automate the execution-trace generation [38].
Introduction

From the above, we can derive two important problems that come with the current verification techniques of temporal requirements, being that verification is time and cost expensive, and that it might be incomplete, and therefore unreliable. Because of this, formal verification techniques have been developed[56].

During formal verification, we will meticulously explore the correctness of the system. To be able to do this, we need an abstract, mathematical model of the system. This can be in the form of a timed automaton, a finite state machine, a Petri net, etc., which can be combined with specification formulae. With this model, we will be able to verify the temporal requirements of a system, as they are specified in the system specifications. The advantages, of formal verification, are that it will greatly reduce the time and cost factor to verify a system, since we now can make use of specialized verification tools, and it will greatly enhance the quality of the overall system design[56]. However, the main drawback of formal verification, is that it requires a formal specification language, which are not that popular in the software industry. This is mostly caused by the popularity of informal languages, as will be discussed next, and the lack of education of software developers concerning formal languages.

As stated above, formal specification languages are not commonly used and applied in the software industry[45]. UML\textsuperscript{1} is the most used modelling language in the software development community[45], and the main reasons why it is so popular are:

- UML is very easy to understand and learn
- it has several views of the system at hand, and allows a clear overview of the software architecture
- it is the common language of different kinds of people, going from developers, to marketing people, to customers
- it has multi-vendor tool support

However, UML is an informal language, since OMG\textsuperscript{2}, the organisation that specified the UML standard, never was able to come to a consensus concerning a precise semantics. This has as a result that UML has very weak semantics[47], which makes it not suitable for formal verification.

It should be clear by now, that formal verification is important for the verification of real-time systems, but that these systems are, mostly, designed in an informal language, being UML. Since UML is so widely used in the software development sector, it is a difficult task to change these work methods. This brings us to the main motivation of the research I have performed, in that we have searched for a formal language, for which a UML profile exists, so the industry is more likely to adopt it, and which does allow formal verification. A UML profile allows us to generically extent UML to build models in particular domains, such as the real-time systems sector. We have found a languages that encapsulate these two requirements, in the form of the Specification and Description Language (SDL) and the Message Sequence Chart language (MSC). For these languages, there already exists a UML profile, which is described in the Z.109 Recommendations[47]. This will make the transition from the informal UML

\begin{footnotesize}
\textsuperscript{1}UML: Unified Modelling Language
\textsuperscript{2}OMG: Object Management Group
\end{footnotesize}
language to the formal SDL and MSC languages smoother, which will in turn make it more likely that the industry will adapt itself to these formal languages.

SDL is a formal description language, first developed by ITU-T, and is described in the last installment of the Z.100 Recommendations[29]. SDL was first used in the telecommunication sector[51][37], but because of its powerful development environments, that allow verification, simulation, test generation and even code generation[40], it has spread to the software sector, in general, and the real-time software sector, in particular. SDL will describe the static structure of a system. For very large, complex systems, engineers will use MSC alongside SDL. MSC, also developed by ITU-T, and described in the Z.120 Recommendations[30], is used to describe the dynamic behaviour of the system, more specifically the internal flow of control type behaviour, using scenarios and anti-scenarios[30].

SDL and MSC have two characteristics, which represent the main reasons why we suggest these languages. First, ITU-T has developed the Z.109 Recommendations, titled "SDL combined with UML"[47]. In these recommendations, it is described how SDL and UML can be used next to each other, and how a subset of the UML language can be mapped one-to-one to a subset of the SDL language[47]. Furthermore, the MSC language is very similar to the sequence diagram of UML. Because of this, the gap, between the informal UML language and the formal SDL and MSC languages, has been reduced to a minimum, which makes it very likely for the industry to adopt these formal languages. Second, as these languages are formal, they have an underlying mathematical structure. In this case, the underlying structure is an abstract or finite state machine. The advantage of this is that, now, we have the possibility to add clocks to this finite state machine, which will transform it to a timed automaton, which, in turn, allows us to perform an automatic formal verification of the temporal requirements of the system. This will be further addressed in the course of the thesis.

However, there are two main problems that will need to be addressed. First, since both SDL and MSC originate out of the telecommunication sector, for which time, together with time progression, is not that great of an importance, these languages have a very weak model of time, and they lack the ability to fully express all real-time requirements in the design. Obviously, this is a vast deficiency of the SDL and MSC language, when used in the real-time systems sector. Therefore, we will have to create a strict model of time, and find a formal way, on how we can describe the temporal requirements, inside the SDL and MSC models. Second, there is, still, very little tool support to verify the temporal requirements of the SDL and MSC models. Therefore, in this thesis, we will examine different ways on how we can perform a static verification of these temporal requirements. A static verification is a verification technique in which we will only deal with static time information, and discard all dynamic time information, such as, for example, extra delay caused by pre-emption during the scheduling of processes.

The rest of this paper is organized as follows. In chapter 2, we will discuss the SDL language, together with its deficiencies surrounding a proper model of time, and its deficiencies concerning the expressivity of real-time constraints. In this chapter, we will define the semantic rules, with which we can formally describe time information and timing requirements in the SDL model. In chapter 3, we discuss the MSC language, also with its deficiencies surrounding the expressivity of real-time constraints. Here, we will define semantic rules, as well, with which we also can formally describe time
information and timing requirements. In chapter 4, I describe how we can translate the time information and timing requirements from the SDL model, to its equivalent counterpart in the verification tool UPPAAL. We will conclude this chapter with a test case, which will prove the correctness of the proposed technique. In chapter 5, I will describe which different techniques exist to verify the timing consistency of the MSC model, and how the timing requirements, as described in the MSC model, can be transformed to their counterpart in the verification tool UPPAAL. This chapter will also conclude with a test case, to prove the correctness of the proposed techniques. In chapter 6, I will shortly describe what still needs to be addressed in the future, and which open issues remain, before I conclude in chapter 7.
Chapter 2

The Specification and Description Language (SDL)

2.1 SDL: a short introduction

The Specification and Description Language SDL is, as mentioned in the introduction, a Formal Description Technique (FDT), first developed by ITU-T in 1976. Currently, SDL is described in the Z.100 Recommendations [29], which had its most recent update in 1999, and is generally referred to as SDL-2000. The main benefit of SDL is that it allows us to define the specifications of complex, discrete and reactive systems in a precise and unambiguous way, and this independent from any particular application domain [46].

SDL was initially developed for the telecommunication sector, but has also proven useful in other sectors, among which is the real-time system development sector, our main field of interest. SDL, combined with MSC, which will be discussed in chapter 3, has proven successful for several reasons. First, SDL provides both a graphical and a textual representation. The graphical representation allows an engineer to intuitively develop and understand the system structure and architecture, while the textual representation allows third party tools to validate and simulate the actual model, and this even in very early stages of design. Second, there exist tools that allow the synthesis of the model into C or C++ code. Because of these two benefits, SDL will allow rapid development and future extension of a system [39].

SDL can be used in the entire software development process, going from the analysis of the requirements to the design and implementation of a system. After obtaining the functional requirements, we can create an initial SDL and MSC model, which will describe the static structure of the system, together with its general behaviour. In later stages of system development, the SDL model will be refined, details will be added, and eventually it will be synthesised to, for example, C code. It is clear now that SDL is both a specification language, in which system requirements are defined, and a design or implementation language, from which source code is eventually synthesised. A clear distinction between these two different aspects of SDL is important, as we will only treat SDL as a specification language. This choice originates from the initial

\footnote{For abbreviation purposes, we will refer to SDL-2000 as SDL, unless mentioned otherwise}
description of the problem, in the introduction, in which we wish to validate timing requirements of real-time systems in early stages of system development.

In SDL, the structure and behaviour of the system will be described by processes which can be grouped into blocks. These blocks can, in turn, be grouped into nested blocks, etc. These processes and blocks will communicate with each other through asynchronous communication paths via signal exchanges. These paths are called channels, when they connect two different blocks, or signal routes, if they connect two different processes. The difference between a channel and a signal route is that the first has an arbitrary delay in message passing, while the latter does not [3]. An example of a simple SDL model is depicted by Figure 2.1.

![Figure 2.1: A simple SDL diagram: processes, blocks, signal routes and channels [3]](image)

2.1.1 SDL: the basics

I will now shortly describe the basic components of the SDL language. I will only treat the very basics of SDL needed to understand the examples depicted in this document. For more information about the SDL syntax, I refer to [29][58].

The overall SDL model is referred to as the system. Everything outside the system is referred to as the environment. There is no graphical notation to separate the system from the environment, but one might use the notion of a block to capture the entire system [58]. A block has no real physical meaning, it just allows a hierarchical decomposition of the system in inner blocks and/or processes. A process can be regarded as code that will be executed.

A block is represented by a rectangular box, while a process is represented by a octagonal box, as we can see in Figure 2.2(a) and Figure 2.2(b), respectively [29][58].

As stated above, processes and blocks communicate using signal exchanges. The connection paths used can be connected to either the environment, processes or other communication paths. A path can in turn be unidirectional or bidirectional. The graphical representation of both a channel or a signal route is the same. A unidirectional and bidirectional communication path is depicted in Figure 2.3(a) and Figure 2.3(b), respectively.
2.1 SDL: a short introduction

Other important graphical representations of SDL are the start and stop symbol for the execution of a process, as depicted in Figure 2.4(a) and Figure 2.4(b), respectively.

A process can also be in a certain state, receive messages and send messages. This is depicted in Figure 2.5(a), Figure 2.5(b) and Figure 2.5(c).

2.1.2 SDL and the Development of Real-Time Systems

As mentioned in the beginning of this chapter, the use of SDL has expanded over more domains than the telecommunication sector, for which SDL was originally developed.
The sector of real-time system development is one of these sectors. The main reason for this is the same reason why SDL is popular in the telecommunication world, being the intuitive graphical representation of the system, verification tools that are even usable in early stages of development, and the possibility to synthesise the model to C or C++ and map it to numerous physical configurations [39]. Besides that, SDL also fully supports asynchronous communication channels, as is needed in real-time systems.

In real-time systems, time itself is of great importance. Together with the functional requirements, which tell what the system should do, we now also have non-functional requirements, in terms of time constraints. There is a permanent relationship between functional and non-functional requirements, since the correctness of the system now not only depends on the correctness of the events, but also on when in time these events take place. These time constraints are a burden for engineers, especially in asynchronous systems with heterogeneous communication lines, lines with different speeds, delays, losses, etc., as they have to be maintained in all phases of system development.

There are, however, some drawbacks concerning SDL and real-time systems. The main problem of SDL is that in its Recommendations [29] it is not defined how we can formally describe these requirements when designing a real-time system. When we use SDL as a specification language, we need to be able to specify our assumptions of durations of events, delays caused by sending messages over channels, etc. How this information can be formally introduced in SDL will be discussed later on in this chapter. When we use SDL as an implementation language, we need primitives to ensure our timing constraints during code generation. However, as mentioned above, this document will not deal with the SDL as a programming language, since this is out of the scope of this work.

Another problem we will encounter while developing and verifying real-time systems modelled in SDL, is the progress of time itself during the execution of the system. The Z.100 recommendations of SDL [29] only give a very weak time model. How we can improve the notion of time, will be discussed in section 2.2 of this chapter.

### 2.1.3 Timed automata

As will be discussed in chapter 4, the SDL model will be transformed into a timed automaton for verification purposes. This will make it possible to validate its requirements. Thus, the reader should keep in mind right now that with every model in SDL, there is a certain timed automaton equivalent. Events and processes will have a representation in this automaton as states with transitions, and the timing requirements that will be included in the SDL model, as discussed in this chapter, will also be present in this timed automaton. I will elaborate about this more in chapter 4.

### 2.2 SDL, time and clocks

SDL, as it is described in its Z.100 Recommendations [29], has a now variable and a timer to deal with time. The now variable will be used to simulate the progress of time in the system, as it represents the system clock, and the timer construct will be used to
define timing requirements. We will now discuss the drawbacks of the now variable. The disadvantages of the timer construct will be handled in section 2.3 and section 2.4.

The now variable has three drawbacks. First, it must be assumed that time only progresses when the system is in a stable, inactive state, or in other words, when no signals are sent or consumed [29]. This means that if there is no timer set, time will not progress [40]. Second, as stated in [46], "whether two occurrences of now in the same transition give the same value is system dependent, however it holds that now$_{n+1} \geq$ now$_n$". This implies that the now variable might or might not be different for two occurrences of it in two different, concurrent processes somewhere in the system. Because of this undefined propagation of the now variable, together with the fact that it can only be used via timers, it is impossible to use it for validation of real-time systems. This is so, since now can take on any value, which would cause a state space explosion during validation. A third drawback of the now variable is that its physical meaning is unclear. It is not defined whether it expresses seconds, nanoseconds, minutes. Thus, we will need to create a formal clock model that overcomes these deficiencies. How this can be done, is suggested by ITU-T [46], and this will be discussed next. A more complete elaboration, on how we get to this specific clock model, can be found in appendix A.

2.2.1 Formal specification of clocks

With the knowledge of theoretical clocks, as described in appendix A, we can formally describe a usable clock. A discrete real clock has been formalized by ITU-T[46] as followed:

\[
\text{ClockDefinition:} \\
\text{clock} \text{ Identifier} [\text{ClockParameterList}] \\
\text{ClockParameterList:} \\
\text{Clockparameter} \\
\text{ClockParameter:} \\
\text{start} \text{ IntegerValue TimeUnit} \\
\text{drift} \text{ RealValue} \\
\text{offset} \text{ IntegerValue TimeUnit} \\
\text{granularity} \text{ IntegerValue TimeUnit} \\
\text{range} \text{ IntegerValue TimeUnit}
\]

An example of a clock declaration would thus be e.g.

\[
\text{clock} \text{ clock1} [\text{drift} \ 10e-6, \ \text{offset} \ 1 \text{ ms}, \ \text{granularity} \ 100 \text{ us}, \ \text{range} \ 3000 \text{ us}]
\]

where the clock has a drift of 10 ppm, a maximum offset of 1 millisecond, a granularity of 100 microseconds and a range of values going from 0 to 2900 microseconds.

A derived counter has been formalized by ITU-T[46] as followed.
counter Identifier [CounterParameterList]

CounterParameterList:
   Counterparameter, CounterParameter

CounterParameter:
   Reference identifier
ticksrefclock IntegerValue
granularity IntegerValue
range IntegerValue
displacement IntegerValue TimeUnit

An example of a counter declaration would be e.g.

counter counter1 [reference clock1, ticksrefclock 1000, granularity 1,
range 24]

where the reference clock clock1 will have to do 1000 ticks before this counter will increase by one.

How one can use these formal declarations of clocks and counter in SDL will be discussed in section 2.4.

In the next two sections we will discuss the shortcomings of the timer construct as defined in the Z.100 Recommendations, and how non-functional requirements can be formally included in the SDL model.

2.3 Urgencies

As mentioned in the beginning of the previous section, SDL, as described in the Z.100 Recommendations, has besides the now variable also a timer construct to include a notion of time in the SDL model. The problem with this timer construct, however, is that it is only capable to express minimal duration of events [21] [40]. To express maximal duration, we could add more timers together with invalid states that are defined as unreachable in the state machine, which is the equivalence of the SDL model as discussed in the beginning of this chapter. This, however, would mean that we need programming features in the SDL model, which is undesired. Another method to overcome this deficiency of the timer construct, as used in several validation and verification tools[21], solves the problem by adding time guards, that impose waiting, to the state machine, by making all transitions instantaneous and by letting time only progress if specified by a timer. This way of working, however, will require timing constraints everywhere in the model, which will make the model cumbersome. Also, using explicit time guards will make the line between functional requirements and non-functional requirements very vague, which will, in turn, make the model hard to read.

The problem at hand is thus how to formally define maximal duration of events in SDL, or in other words, how to define when a transition in the timed automaton equivalent should be taken. In general, there are three factors that characterize a transition. First, there is a non-time dependent enabling condition which comes from the functional requirements. Second, there is a time dependent enabling condition, which
The Specification and Description Language (SDL) 2.3 Urgencies

depicts at which point in time the transition is possible. Therefore, this will define the minimal time to wait before a transition can be taken. The third, last and new factor that characterizes a transition is the urgency of the transition. This will bound the upper time the system can stay in a certain state. There are three different kinds of urgency, as specified by [21][40][39]:

- eager: the transition should be taken as soon as possible, and no time shall pass
- lazy: an arbitrary amount of time might pass before the transition is taken
- delayable: a combination of the eager and lazy urgency types. The transition will be lazy to a certain point in time. Afterwards, the transition becomes eager.

With this new knowledge of urgencies, we can restate the problem described above. In the Z.100 Recommendations [29], all transitions are considered lazy. This is of no importance for code generation, but it is for validation of real-time systems. Some verification tools, however, make all transitions instantaneous, or eager. These are two extremes that are not usable in reality. For example, we might want an external input to be lazy when the time of arrival is unknown, or we might want an input to be regarded as eager for critical events. This is now possible using urgencies.

During validation, we might, however, come to the conclusion that we did not specify urgency everywhere in the system model, so it is not clear whether to treat these transitions as eager or lazy. A solution suggested by [21] proposes different validation modes. We can either treat all undefined urgencies of the transitions as eager or lazy, or we can let the user define the urgencies during validation. The latter is not a great option when the system is large and a lot of transitions are not specified as eager, lazy or delayable. We will deal with this problem in chapter 4.

Working with urgencies has one more danger that needs to be considered during validation. Should there always be one eager path available, then time would not progress at all. This is also called the zeno behaviour [39]. This would make the validation obsolete. The problem, however, is the model itself, as created by the engineer, and is of no concern for us.

**Representation of urgencies in a formal way**

As was the case for clocks and derived counters, we need a formal way on how we can describe the urgency of an event in SDL. I have, therefore, created the following rules:

```
UrgencyDefinition:
  urgency urgencyType
  urgencyType
  eager
  ||
  lazy
  ||
  delayable[IntegerValue TimeUnit, IntegerValue TimeUnit]
```
where for the delayable urgency, the first integer value is the lower bound of the delay and the second integer value is the upper bound of the delay.

An example of a delayable event can be

\textbf{urgency delayable} [5 ms, 1 s]

which implies that the delay of the event should be between five milliseconds and one second. How we can specify these urgencies in the actual SDL model will be discussed next.

2.4 Annotations in SDL

Now that we have established a formal time model using clocks and derived counters, and a formal way of expressing urgencies of events, we still need to formalize the way to actually express timing requirements.

We can divide the timing requirements into two groups\cite{39}. On one side we have assumptions. These assumptions tell something about the system, e.g. its scheduling, the system architecture, etc. Assumptions will also tell something about the environment, such as the periodicity of incoming signals, etc. On the other side, we have assertions, requirements of which we are sure about, mostly local properties of the system. It can be so, for example, that it is known which communication means will be used in the system, so that we e.g. know for sure what delay a certain communication channel will have.

There are several options on how we can add these temporal requirements \cite{39} \cite{46} \cite{50}. First, we can try to use already existing semantics and use them in such a way that we can express non-functional requirements. This seems not to be a good idea, since this would mix functional and non-functional requirements, which is not desirable. A second option is to actually change the SDL language. This would have a strong impact for all sectors SDL is used in, which is also undesirable. A third and most promising option is the use of annotations, as the title of this section indicated. An annotation is a comment in SDL. Our goal will be to formalize the way we express non-functional requirements in these annotations. The benefit of this technique is that we do not influence the actual SDL language, which means that if the model is used for other purposes than validation of real-time properties, the non-functional requirements will just be comments which can easily be ignored. Another benefit is that we keep a clean separation between functional and non-functional requirements in the model itself, which makes the model more readable for the engineer and does not complicate code generation afterwards. Furthermore, an advantage of using annotations is that we can now take benefit of its hierarchical structure. This implies that an annotation which is defined in a high level of the SDL model also has effect on parts of the model at a lower level of abstraction, unless it is refined or overwritten. Again, this will greatly improve the usability of annotations to express non-functional requirements.

There are three techniques on how we can express non-functional requirements in annotations, as suggested by \cite{39}. First, we can put the annotation inside the SDL model. This will make the model very easy to understand, but if many of these annotations exist, it might make the model cumbersome. A second technique is to put the annotation in the SDL document, and third we can include the annotation from another
document inside the SDL document. The advantage of the latter is that we now can exchange and reuse already existing information about the system, e.g. information of the used communication means. Example annotations can be seen in Figure 2.6.

Since we need to make a clear distinction between a regular comment and a non-functional requirement, we will use from now on the following rule to mark a temporal requirement[39]:

$$\{\text{keyword} \ \text{identifier} \ _{constant} \ _{value}\}$$

Now I will describe the various annotations that are needed to express all possible temporal information and requirements in SDL.

### 2.4.1 Clocks and derived counters

We already stated how we can formally describe clocks and counters in section 2.2.1. We can now use this together with the annotation technique as described here, to actually put this information inside the model. This can be seen in Figure 2.7, where we define a clock that has a drift of 10 ppm$^2$, an offset of 1 millisecond, a granularity of 100 microseconds and a range of values going from 0 to 2900 microseconds. Since a timer or a counter is usually defined once for the entire model, we will use the second technique of annotations, by putting the information about the clock inside the SDL document.

Figure 2.7: A clock definition in SDL

---

$^2$parts per minute
2.4.2 Urgencies

We also need to be able to express urgencies, as was described in section 2.3. We will also use the annotation technique for this, and since urgencies are closely related to events, we will use the first annotation technique, as can be seen in Figure 2.8. Here, we state that if a message is received, the transition should be taken immediately, without losing time, or in other words, that the transition should be eager.

![Figure 2.8: Urgencies in SDL: eager transitions](image)

2.4.3 Events

Events are important entities in the model, which can be used to specify timing requirements, as described in [50] [46], e.g. before and after the sending of a message. Therefore, we need to be able to formally specify these events in the SDL model. Because of this, I have defined the following semantic rule:

```
event: event eventName
```

An example of an event in a SDL model can be seen in Figure 2.9. These events can later on be used to specify timing requirements.

In the next sections, I will define formal rules to express requirements considering duration or processing time, communication channels and the periodicity of external input. These are the three most important timing requirements needed in real-time systems, as described in [39][46][50].

2.4.4 Duration of events

The duration of events has an ambiguous meaning. From the user point of view, the duration of an event includes not only the execution time of the event itself, but also the possible blocking time due to scheduling and the queue time of the process in the processor of the system. Thus, it needs to be clear what is actually meant with the
duration of an event. Since scheduling and queuing of processes are system dependent, and the fact that they have nothing really to do with the process itself, we will assume that when we are speaking of durations of events or processes, we will only speak about its execution time, the time the processor needs to execute the event or process/processes.

Therefore, we need to be able to formally specify three separate things, being how long a certain process takes to execute, where we want an event reference and how to describe the timing requirements considering duration of different events.

To indicate how long a certain process takes, I have defined the following semantic rule.

\[
\text{duration of process:} \\
\text{duration IntegerValue TimeUnit}
\]

To indicate where we want an event reference, I have defined the following semantic rule.

\[
\text{event reference:} \\
\text{event String}
\]

To indicate the non-functional timing requirement, I have defined the following semantic rule.

\[
\text{durationBounds:} \\
\text{span (event1, event2) < timeInterval} \\
\text{timeInterval:} \\
\text{IntegerValue TimeUnit} \\
\| \\
[ \text{IntegerValue TimeUnit, IntegerValue TimeUnit} ] \\
\| \\
[ \text{IntegerValue ± IntegerValue% TimeUnit} ]
\]

The above implies that the time between two events can be either fixed number or an interval.

An example of how events are declared, how we can give a certain duration to a process and how we can define the non-functional requirements concerning duration are expressed can be seen in Figure 2.9. Here, we say that there are two important events A and B, and that there should not be more than 20 milliseconds between these two events. The algorithm itself will take between 15 to 20 milliseconds. We can therefore conclude in this case that the non-functional requirement would hold during verification.
2.4.5 Channels

Channels, as described in the Z.100 Recommendations, are FIFO\(^3\), lossless, and have an arbitrary delay. It is clear that this is not according to reality and that this is too vague. Signals sent over a channel can be lost and can be received in another order than the one they have been sent with. Also the fact that channels might have an arbitrary delay is not sufficient for the verification of real-time systems. Therefore, I created a semantic rule to define the non-functional requirements of channels in the SDL model.

\[
\text{channelInformation:}
\begin{align*}
\text{channel} & \quad \text{delay} = \text{timeInterval}, \\
& \quad \text{lossRate} = \text{RealValue}, \\
& \quad \text{ordered} = [\text{TRUE} \parallel \text{FALSE}]
\end{align*}
\]

Above we can see that a channel can have a fixed delay or a delay between certain boundaries. A channel can also have a certain lossRate, in which 0 percent means no loss of signals, and ordered means whether or not the channel keeps the order of the signals or not.

An example of such a channel specification can be seen in Figure 2.10. Here it is defined that the channel has a delay between three and five milliseconds, on which message losses do not occur and the messages received might not be in the same order as they were sent.

2.4.6 Periodicity of external input

As processes within SDL communicate by means of signals, will the system also communicate with the environment via signals. These signals can have certain non-functional characteristics such as response time and inter arrival time. These too should be expressible in SDL. The formal semantic rule for this can be seen below.

\(^3\)FIFO = First In, First Out
In this chapter, we have discussed what is missing in the SDL language, so that it can be used in the development of real-time systems. First, we have discussed that the model of time, defined in SDL using the `now` variable and timers, is insufficient to use for verification purposes of non-functional requirements. Therefore, we needed to be able to define a discrete real clock in SDL with its characteristics, such as granularity, range, etc. For this, the clock model, as was suggested by [46], was used.

Secondly, we have stated that timers are insufficient to specify all characteristics of a real-time system. Therefore, we first discussed the different aspects that require specification in the SDL model, as they were described in [39][46][50]. It turned out that we need to be able to specify

- urgencies
- events, which represent important places in the model
The Specification and Description Language (SDL) 2.5 Summary

- duration of events, e.g., the duration of a process
- deadlines, which define non-functional requirements of the system
- channels, since channels can have a certain loss rate, messages might arrive in different orders, etc., and this might influence the timing property of a system
- periodicity of external input, which is needed to be specified, as will be seen in chapter 4

For each of these temporal specifications, I have created a semantic rule, so that we can express them in a formal way.

After this, we discussed how we could actually represent these requirements in the SDL model. Three ways of doing so were available, as described in [39] [46] [50]:

- use already existing semantics
- change the SDL language
- use annotations, or comments, from the SDL language to express the time-related aspects

Using annotations seemed to be the best way to go, since this would have the least impact on other sectors that use SDL, besides the real-time sector. Important here was that we needed to find a uniform way on how to express these annotations. This was done by using \$\{ ... \}\[39], to separate the annotations from the regular comments.

As one might have noticed, we have only described very basic timing requirements. This is so, since most timing requirements tell something about the behaviour of the system, and this should be dealt with by MSC, as will be explained in the next chapter [46].
Chapter 3

Message Sequence Charts

Message Sequence Charts (MSC), is a graphical specification language created by ITU-T [30]. It is a formal language which had its first appearance in 1993 and 1996, when it was formally described in the z.120 recommendations [27][28]. ITU-T gave MSC an update in 1999 [30], generally known as MSC-2000\(^1\). MSC is in fact a programming language in its own right, but for complex systems we will also require the Specification and Description Language (SDL).

MSC can be used to specify and document system requirements in terms of sequences of interactions that guide the system design in early system development. With MSC one can describe test cases and scenarios which the eventual system should adhere to, and with MSC one can also verify system properties that are defined in SDL specifications. Important to mention here is that, with MSC, one can not only create scenarios that the system should follow, but also scenarios that the system should never follow, which are called anti-scenarios [12]. One can also perform system tests with MSC, by visualizing sample behaviour of simulated system specifications using scenarios. Even after system deployment one can still use MSC to express legacy specifications of a system to help in software maintenance and software reengineering [23].

There are two different types of MSCs: bMSC or basic MSCs and hMSCs or high-level MSCs [30]. Basic MSCs will depict one scenario or anti-scenario that the final system should obey to. A scenario describes different patterns of interaction between processes within the system. To define all requirements of the requirement specifications, it is clear that many scenarios, and possible anti-scenarios, should be created and combined with each other. The combination of these MSCs is called the MSC specifications.

Figure 3.1 depicts a simple bMSC. As one can see in Figure 3.1, a bMSC consists of different process instances, depicted by vertical life lines. A process can have different events that take place during its existence. In MSC, these events are instantaneous, in that they do not consume any time. The processes communicate with each other through messages, which are depicted by the horizontal arrows. A message arrives in the message queue of the receiving process, and is only considered to be read when it is fetched from this queue. A bMSC has certain conditions it should abide [12]. First of all, every process is a sequential agent. This implies that all events have a

\(^1\)For abbreviation purposes, we will refer to MSC-2000 as MSC, unless mentioned otherwise.
certain linear order that should be respected. In Figure 3.1, the scenario depicts that the receiver process should not send the “readyMsg” message before it receives the “initMsg” message, or at least not in this scenario. Second, a message should be sent before it is received, which guarantees causality in message passing. Third, all messages that are sent, should also be received. Forth, the causality relationship between events should be respected. This means that the order of events totally depends on 1) the order of events within one process and 2) the messages that are sent and received. Fifth and last, a scenario depicted by bMSCs can be complete or incomplete [30]. This has as advantage that in early design stages one can omit trivial details, which will be added in later stages of the system design process.

While bMSCs are sufficient in small projects, one would like to work with different levels of abstraction in larger projects, thus one would want a certain structure in the MSCs. This can be achieved by using high level MSCs or hMSCs. An hMSC is a directed graph that can contain references to both hMSCs and bMSCs. It will graphically describe the compositions of the different MSCs, and will describe parallel, iterative, sequential and non-deterministic execution of these MSCs [25]. The graphical symbols that make up an hMSC are shown in Figure 3.2.

An example of an hMSC is depicted in Figure 3.3(a) and 3.3(b), where in the first
3.1 Expressing time requirements by extending MSC

3.1.1 How to extend MSC

The academic community is somewhat divided in how MSC should be extended to support timing requirements. On one side, there are people who wish to actually change MSC to fit the new functionalities [54] [30], while on the other side there are people who wish to extend MSC using annotations [48]. This is very similar to what we have seen in section 2.4 of the previous chapter. As was the case for SDL, we will now also prefer the use of annotations in MSC, instead of the extension of it, to represent timing information and timing requirements. The main reason for this
is that, as was described in section 2.4, MSC will not only be used in the real-time systems sector, but in many other sectors. Thus the advantage of the approach using annotations is that we do not put such a burden on these other sectors, as for these sectors, the annotations can be treated as regular comments.

We will not divert from the annotation syntax as described in chapter 2. Thus an annotation for a non-functional requirement still looks like

$$\{\text{keyword identifier constant value}\}$$

### 3.1.2 Time and timers

As MSC is closely related to the SDL model, we can use the model of clocks as described in section 2.2 to represent time. An already existing construct in MSC to deal with time is available in the form of timers. As stated before, timers have been used in MSC since its early days [27]. A timer can be set, reset and can "time-out". A timer can graphically be connected or unconnected, and thus one has two different sets of graphical representations of timers, as can be seen in Figure 3.4 and 3.5 below. The two representations exist only to make the graph more readable. One should use the connected representation when one wants to depict time requirements between two related events, and the not-connected representation when one wants to depict time requirements between two events that are not related.

![Figure 3.4: An unconnected timer](image)

![Figure 3.5: A connected timer](image)
In practice, these timers can be used as in the examples seen in Figure 3.6(a) and 3.6(b) [58]. Here, we have two scenarios that describe the successful and failed establishment of a client to a server, depending on the time spent between the request by the client and the answer from the server. In Figure 3.6(a), not-connected timers are used since there is no relationship between setting the timer and stopping the timer. In Figure 3.6(b), connected timers are used since now there is an important connection between setting the timer and its time-out, and, by using connected timer symbols, this is represented visually in a much clearer way. Note that the hexagonal boxes in both pictures depict states, as described in z.120 recommendations [30].

Timers can be used to express both minimal and maximal time bounds of two or more consecutive events, but only for events in a single process. This is the biggest drawback of timers, since often one needs to specify time constraints between processes. This will be solved by introducing time or delay intervals, as described next.

### 3.1.3 HMSC related time requirements: instance delay

The z.120 recommendation of 1999 [30] was found to be incomplete concerning time constraints in repetitive behaviour, as described in [54]. Repetitive behaviour is created by loops in an hMSC, as seen in Figure 3.7. We are able to specify that a bMSC should only take between one and two time units, but we are unable to specify the amount of time between two executions of the same bMSC in a loop. How to specify the duration of a bMSC or the time spent in a certain state of a bMSC will be discussed in section 3.1.6.

The problem of instance delay is solved by [54], by allowing the specification of the instance delay in the bMSC, as shown in Figure 3.8, using the following semantic rule I created:

\[
\text{instance delay:}\quad \text{instance delay timeInterval}
\]

The specification of the instance delay will not have any effect when the bMSC is run for the first time, but will make sure that there is a certain delay, be it specific or an interval, if the same bMSC is run consecutively. In Figure 3.8 for example, should the Transaction bMSC be run for the first time, it will take between 1 and 2 seconds to finish, as specified by the duration annotation in the hMSC of Figure 3.7. At this point, there is no need to take the instance delay into account. Should the bMSC require another run because of the loop, there will need to be an extra delay of 2 seconds before the next execution of that bMSC, as defined by the instance delay in Figure 3.8.

---

2Note that loops are only possible in hMSCs, and not in bMSCs, since bMSCs are purely sequential, as stated in the introduction of this chapter
Message Sequence Charts

3.1 Expressing time requirements by extending MSC

(a) Successful connection of the client to the server

(b) Failed connection of the client to the server due to a time-out

Figure 3.6: Example of connected and unconnected timer symbols in practice
3.1 Expressing time requirements by extending MSC

**Figure 3.7: Repetitive behaviour in an hMSC**

```
Transaction

$$\{\text{duration } [1s, 2s]\}$$
```

**Figure 3.8: How to specify instance delay in a bMSC**

```
msc Transaction

\$$\{\text{instance delay } 2s\}$$
```

### 3.1.4 Time requirements of consecutive events: time intervals

Time intervals or delay intervals can be split up in three different parts. First, there is a time interval that can be associated with an event. This is a global time constraint interval in which an event must occur. Second, there is a message delivery time interval, in which one can specify when a message should be delivered. Third, we have the processor speed time interval, which depicts the time bounds between two consecutive events. All these will be discussed in more detail next.

To express the actual time constraints, one should follow the following rules, as they are described in [30] and [58]:

- For absolute time information: put a `@` in front, e.g. @30s: 30 seconds after system start
- For relative time information: put nothing in front, e.g. 30s: after 30 seconds, from this moment on
- A time interval is expressed between square brackets: `"[" and "]"`, e.g. [5s, 10s]: after 5 seconds but before 10 seconds, from now on
- time units are RTOS specific tick counts, unless otherwise specified using s, ms or µs for seconds, milliseconds and microseconds, respectively
Event associated timing interval

The event associated timing interval depicts the minimal and maximal time between a certain event and any event previous in its trace, as described in [25]. With this, it will be possible to describe temporal behavior such as an event that closes a valve to stop the supply of acid within a specified amount of time, no matter what event was executed before. This timing requirement can be formally described by the semantic rule I created:

\[ \text{event interval:} \]
\[ \text{event interval} \quad \text{timeInterval} \]

This can, for example, be represented as depicted in Figure 3.9. This scenario describes that the valve needs to be closed after 200 milliseconds, no matter what other event was executed before.

![Figure 3.9: Event associated timing interval: closing of a valve within 200 ms](image)

Message delivery interval

As stated in the beginning of this chapter, message passing does not consume any time in MSC, but a message is considered to be received only if the receiving process has fetched it out of its queue. One can see that the first assumption is not realistic, in that in fact it will take some time to transmit a message. It is clear now that we have to include a time interval that depicts the time needed to send the message over the physical link. It is impossible in this case to solve this problem with the already existing timer constructs, since we need time information between two processes. On the other hand, the second characteristic of message passing, being the time it takes before the message is fetched from the input queue, depends on the scheduling algorithm. If the process has a high priority, the message will be received faster than when the process has a low priority. However, since we will concentrate on static verification, schedulability analysis is out of the scope of the thesis.

We will only define the actual time needed to send a message over a certain medium, as described by [25]. Besides execution time, we also have to define two
other characteristics of channels, as discussed in section 2.4.5 of the previous chapter, being message loss and the fact that messages might or might not be received in a different order than the order they were sent in. A formal way of expressing these three characteristics of a channel thus remains the same.

\[
\text{channelInformation:}
\begin{align*}
\text{channel delay} &= \text{timeInterval} , \\
\text{lossRate} &= \text{RealValue} , \\
\text{ordered} &= [\text{TRUE} \lor \text{FALSE}]
\end{align*}
\]

An example of a graphical way to depict the time interval of a physical medium over which messages will be sent is shown in Figure 3.10[25]. Here, we see that it will take between 100 and 200 milliseconds to send a message over a certain channel, that there is no message loss and that messages might arrive out of order.

**Processor speed interval**

The processor speed interval, as described in [30], depicts the minimal and maximal allowed time bounds between two events in a single processor. The processor speed interval can only be used for two successive, visually ordered events in a MSC, as was the case for the message delivery interval. We can formally express the processor speed interval using the following semantic rule.

\[
\text{processor speed interval:}
\begin{align*}
\text{processor speed interval timeInterval}
\end{align*}
\]

An example representation of the processor speed interval is shown in Figure 3.11. Here, we can see that the maximal time interval between the calculation state and the
actual sending of the result should be between 200 and 500 milliseconds [30].

It can be argued that this problem can also be solved using timers. This is true, but we would need two timers to depict the lower and upper bounds of the interval, which will make the chart cumbersome. On the other hand, one can not use processor speed intervals to replace timers, in that the first can only specify delays between two consecutive events, while the latter can do this between two or more consecutive events.

### 3.1.5 Time requirements of not-consecutive events

#### Precedence edges

With timers, event intervals, message delivery intervals and processor speed intervals, there was always the prerequisite that the events are consecutive, one after the other. One might however want to specify timing constraints between events that are not consecutive, e.g. events that do not reside in the same process and that are not connected by message passing arrows. For this, one can use precedence edges [54], as can be seen in Figure 3.12. The drawback of this method is, however, that the message sequence chart can become cluttered and unreadable, which is addressed in [54].

However, since we stated in section 3.1 that we would not change MSC, which would be required to implement the above graphical notation, but that we would use annotations, we have to be able to 1) depict the start and end of a precedence edge using annotations, and 2) depict the timing requirements that go with the precedence edge. The first problem, I solved using the same technique as in section 2.4.3 of the previous chapter. We will define the start and stop of a precedence edge as two events that are important to us. This can formally be done using the following semantic rule that I created:
To depict the actual timing requirements, I have created the following semantic rule:

\[
\text{durationBounds: } \quad \text{span} (\text{event1}, \text{event2}) < \text{timeInterval}
\]

An example of the graphical representation of these two semantic rules can be seen in Figure 3.13. Here, we can see that there should not be less than 400 milliseconds and not more than 600 milliseconds between events A and B and that these events are unrelated, i.e. they are not consecutive, in that they do not reside on the same processor and they are not directly connected by a channel.

**Absolute time**

One can also define an absolute moment in time at which a certain state should be reached or a certain event should occur [58]. The formal semantic rule that should be used to depict this is the following.

\[
\text{absolute point in time: } \quad \text{absolute} @\text{pointInTime}
\]

A graphical representation of this is shown in Figure 3.14. Here, it is depicted that the system should reach the connected state 30 time units after system start.

### 3.1.6 The synchronous hypothesis

In the z.120 recommendations [30] it is stated that "all events are instantaneous, i.e. atomic and do not consume time". According to the synchronous hypothesis, this gives
3.1 Expressing time requirements by extending MSC

Figure 3.13: Timing constraints on two unrelated events

Figure 3.14: Absolute time

no problem during verification of functional system requirements of the system [2] [11] [57]. In this section, I will first shortly describe what the synchronous hypothesis stands for and what its advantages are. Then I will make my case that in order to verify temporal requirements, one can not rely on this hypothesis. I will not discuss why the synchronous hypothesis is actually useable, since this is out of the scope of this document. I refer to the following sources for more information [2][11][57].

According to [57], the main question is "to which extent must the temporal aspect of the system be modelled?". We discussed in section 2.1.3 that every SDL model has an equivalent state machine. This is also true for every MSC model. An example state based system is shown in Figure 3.15. After an arbitrary physical delay, the system will sense a state change, defined here as the "new observable state", and it will take the system an arbitrary amount of time to change to the control state. This delay is called the react delay.

The synchronous hypothesis will now assume that the system is infinitely fast concerning reaction on state changes. Because of this, one can set the react delay from Figure 3.15 to zero. In other words, the system will be able to produce the output
synchronously with its input, which will thus take no time at all [2]. Furthermore, the hypothesis will assume that the physical delay only takes one (abstract) clock tick. The system as described above can thus be transformed to the one shown in Figure 3.16.

The advantages of the synchronous hypothesis are manifold. First, synchronous systems are easier to describe and to analyze than asynchronous systems because, while using the synchronous hypothesis, the system becomes highly deterministic. Second, sophisticated algorithms can be used to produce highly efficient code, which will be more predictable than asynchronous code, and thus be more readable. Third and last, and of most interest for us, the synchronous hypothesis, and the fact that it makes the system more deterministic and thus predictable, will allow automatic verification of the system, and it will avoid the state space explosion problem[2].

Thus, the hypothesis is of great help during verification of the system, and it is used in [12], [54], [25] and [23] to verify temporal aspects of the system. The goal of this verification is whether or not the temporal specifications stated in the MSCs can
be met. For example, Figure 3.17(a) is temporally incorrect since the timing requirements are not always met. This is so, because we state that the time between event A and B, which is the sending of message $a$ and the reception of message $b$ respectively, should take at least 400 milliseconds. However, if we look at Figure 3.17(a) we can see that sending a message over the channel takes a minimum of 100 milliseconds. Thus, the minimal time between events A and B will be 200 milliseconds, being two times the time needed for sending two messages, while regarding the computation event, in accordance with the hypothesis, as immediate. Therefore we can say that the temporal aspects of the system are in conflict. On the other hand, in Figure 3.17(b), the temporal requirements are met, and the verification will be successful. This is so since the minimal time between events A and B, being 200 milliseconds as discovered above, will always be larger than 150 milliseconds. The same goes for the maximal time bound: the maximal time spent for sending and receiving a message is 400 milliseconds, being the addition of the maximal time needed to send a message over the channel, while the maximal time between the two events should be smaller than 500 milliseconds, which is thus always the case.

As stated in the introduction, we want to find the global configuration of the system, which consists of many different parts from possibly many different suppliers, each part with its different temporal aspects. These temporal aspects mostly are related to events issued by one part. But, as stated above, current temporal verification techniques will always consider events to be instantaneous, and thus, when we compare two similar parts from two different vendors, these temporal verification techniques will give exactly the same results. This is clearly undesirable. Therefore I conclude that while verifying temporal aspects, one should not use the synchronous hypothesis. What the effect will be in the verification process will be discussed in chapter 5. It is clear now that we will need information about the time spent in a certain state of the bMSC. This can be formally described using the following semantic rule.

\[
\text{duration:} \\
\text{duration timeInterval}
\]

An event can have a fixed duration or a duration interval, as can be seen in Figure 3.18(a) and Figure 3.18(b) respectively.

### 3.2 Summary

In this chapter, we have discussed the necessary extensions of the MSC language, so that MSC can be used in the development of real-time systems.

First, we have shortly discussed what the difference between a basic MSC and a high-level MSC is. Second, we have discussed that we will use annotations to express timing requirements in MSC, similar to what we did in chapter 2.

After this, it became clear that MSC uses the same model of time, by means of the now variable and timers, as was the case for SDL. Because of this, we decided to incorporate the same time model as was described in section 2.2 of chapter 2.
We continued by identifying the different aspects of real-time systems that needed to be expressed in MSC. The different timing aspects themselves, which needed representation in the MSC model, were subtracted from different papers. These aspects can be divided into three main categories, and all of these need a formal way to describe them in a bMSC:

1. HMSC related timing constraints: instance delay[54]. Instance delay is the delay between two consecutive runs of the same bMSC.

2. Time requirements of consecutive events: time intervals[30][58]. We have identified three different time intervals, being:
   - Event associated time interval[25]: depicts the minimum and maximum delay between an event and any event previous in its trace
   - Message delivery interval[25]: depicts how long it takes to send a message
   - Processor speed interval[30]: depicts the minimal and maximal time bound between two processes on the same processor
3. Time requirements of not-consecutive events[54]. Here, we first discussed how precedence edges can solve this problem. However, the use of precedence edges would mean for us to change the MSC language, which was not desirable. Therefore, we developed an annotated version to represent the precedence edges.

For all these timing requirements, I had to create a formal semantic rule, so that all of these aspects are expressible in the MSC model, using annotations.

We concluded this chapter with a discussion about the synchronous hypothesis[2][11][57]. The synchronous hypothesis will regard all events as instantaneous, to make verification easier. However, I made it my case that, to do a correct verification of the timing requirements, as specified in the MSC model, we can not use the synchronous hypothesis. Because of this, it will be required that events are not treated simultaneously, and this requires the possibility to define a certain duration for an event. Therefore, we specified a semantic rule on how to describe the duration of an event in a bMSC.

Figure 3.18: Duration of events in bMSCs
Chapter 4

Static Verification of the SDL model

4.1 Introduction

In chapters 2 and 3, we have studied what needed to be added in the design and programming languages SDL and MSC, concerning timing requirements. We have seen that SDL and MSC needed to be expanded with, among others, a clear mechanism to denote the progression of time within the system, and formal ways on how to express timing requirements. However, our goal was not to merely express timing requirements, but also to validate them. Proving that these timing requirements are respected is of great importance in hard real-time or safety-critical systems, since in these systems, not making a deadline could mean a loss of money and maybe even a loss of lives [31][22]. For example, the airbag system of a car should always respect its timing requirements, since an early or late deployment of the airbag could cost the lives of the people inside the car. However, it turns out that nowadays, engineers only take notice of timing requirements rather late in the design phase [1]. This lead to systems that generally fail to meet these requirements, and projects that are not cost-effective and not manageable. Also, it turns out that more than 50% of the entire project time is spent on the verification of the system[56]. Therefore, there is need of an automatic verification of the timing requirements, even early in the design phase, as was discussed in the introduction. Due to fact that we want to move the industry from informal languages, such as UML, to formal languages, such as SDL and MSC, we can use mathematical reasoning about the system to do the actual formal verification [31].

The verification of timing requirements will deal with very basic safety properties, such as the absence of deadlocks, which occurs for example when two processes try to access shared resources which are mutually locked, and invariances [31]. Invariances are properties that should hold during all possible executions of the system. Verification will also have to deal with more complex safety and liveness properties, such as linear properties [31]. These properties can represent deadlines of tasks, a maximal amount of time between two events, etc.

In this chapter, we will discuss how we can validate a real-time concurrent system, which was designed in SDL, using timed automata. This will be achieved using the verification tool UPPAAL[34]. This chapter is structured as followed. I will first
present a formal definition of a timed automata, together with its labelled transition system. Next, I will shortly discuss the different tools that are out there, and I will explain why I picked the tools of my choice, being the real-time developer studio for the development of the SDL models, and UPPAAL for the verification of these models. After this, I will describe how the translation from the functional and non-functional elements of the SDL model to the UPPAAL model happens, using a self-created SDL2UPPAAL parser. I will conclude this chapter with some examples that demonstrate the correctness of both the parser and the proposed techniques.

### 4.2 Why can we use timed automata?

As discussed before, SDL is based on extended finite state machines, also known as abstract state machines [29][33], which are related to timed automata [31]. The main drawbacks of SDL are, as described in chapter 2, the fact that SDL is loose about the progress of time, represented by the *now* variable, whose values cannot be compared with each other, and its timers and enabling conditions based on this *now* variable. The first drawback implies that the system can stay an undefined amount of time in a certain state, and that the system might take an undefined amount of time to execute a certain task. This obviously makes verification impossible [31]. The timers and enabling conditions on the other hand make the verification process complex and difficult. Therefore, we will map the SDL model to a timed automaton, on which verification will be possible, without adding too much restrictions on the original model.

As stated before, the SDL model can easily be mapped to the timed automaton. The timed automaton will lay a heavy constraint on the progress of time, so that now, we can actually specify actions that occur at a specific moment in time, or in a certain interval of time [31], something that was impossible to do in SDL. Furthermore, time conditions can now be expressed in a simple, straightforward form. The only way to measure time will be by means of clocks. There are certain conditions that should be obeyed when using clocks:

- several clocks can be used at the same time
- all clocks move at the same rate
- all clocks can be initialized, reset and tested separately
- only conditions of the form \(X \sim C \) or \(X - Y \sim C\) are allowed

where \(X\) and \(Y\) are clocks, \(C \in \mathbb{Z}^+\) and \(\sim\) is one of the following equation signs: \(\leq, <, =, >\) or \(\geq\).

Important to note here is that these conditions for clocks will put a certain restriction on the SDL model. If we would want to use clocks with various speeds, we could e.g. use multi-rate automata[8]. If we would want to use clocks with various clock speeds depending on the state of the system, we could use integrator automata[49]. However these automata will lay an even greater burden on the SDL model: in multi rate automata, clocks cannot be compared to each other, and in integrator automata, strong structural restrictions should be obeyed. Therefore, we can conclude that timed automata have a complexity that allows us to model the most general time models [31].
4.2 Why can we use timed automata?

We will now give a general definition of a timed automaton, as defined in [31].

4.2.1 Timed automata: a definition

A timed automaton is a tuple \( A = (\Sigma, \chi, Q, q_0, E, \text{inv}) \) where

1. \( \Sigma \) is a finite set of transition labels
2. \( \chi \) is a finite set of clocks
3. \( Q \) is a finite set of discrete states
4. \( q_0 \) is the initial state, and part of \( Q \)
5. \( E \) is a set of transition edges between the states of \( Q \)
   - each \( e \in E: e = (q, \varsigma, u, a, X, q') \) is an edge for which
     - \( q, q' \in Q \): source and destination states
     - \( \varsigma \): clock guard of the transition, in conjunction with \( \chi \), following the conditions of clocks, being only of the form \( X \sim C \) or \( X - Y \sim C \). Let \( \text{CP}(\chi) \) be set of conjunctions of clock conditions over the clocks of \( \chi \).
     - \( u \): urgency of the transition, which will be discussed later on
     - \( X \subseteq \chi \): set of clocks reset during transition \( e \)
6. \( \text{inv} : Q \longrightarrow \text{CP}(\chi) \): invariant of state \( q \)

The semantics of a timed automaton is given by associating the automaton by an infinite labeled transition system (LTS), depicted by \( G_A \), of each timed automaton \( A \). This LTS is also called the semantic graph.

4.2.2 The timed automaton as a labelled transition system

- nodes, also called zones, of \( G_A \): configurations or dynamic states of \( A \), depicted by tuples \( (q, v) \)
  - \( q \in Q \)
  - \( v \) = valuation of the clocks of the automaton.
  - for a configuration to be consistent: \( v \) must satisfy the invariant \( \text{inv} \) of that state
- edges of \( G_A \): transitions of \( A \) from one configuration to another. In a state \( (q, v) \) there are two transitions allowed
  - Discrete transitions: occur when \( e \in E \) is taken. The system moves from state \( (q, v) \) to state \( (q', v') \) where
    * \( v'(x) = v(x) \): no time has passed
    * \( \forall x \in \chi \) and \( v'(x) = 0, \forall x \in X \)
    The transition is denoted by \( (q, v) \xrightarrow{e} (q', v') \)
4.2 Why can we use timed automata?

– Time transitions happen when an amount $\delta \in \mathbb{R}$ has passed without any
discrete transition being fired. The time transition moves the system from
$(q,v)$ to $(q, v+\delta)$. The transition is denoted by $(q, v) \xrightarrow{\delta} (q, v + \delta)$

According to [31], we can now convert an infinite labelled transition system to a sim-
ulation graph. This simulation graph has the following characteristics:

- it is finite, in contrast to the labelled transition system
- it contains all reachable states of the original timed automaton
- every run of A is contained in a path of the simulation graph
- verification of temporal logic now becomes a reachability problem, which is
decidable in the simulation graph [31]

We can now verify the different temporal properties of the system. This can be
done as followed [31]:

**Absence of deadlock** This occurs if we arrive in state $(q,v)$ and there is no discrete
transition possible, even after arbitrary delay. Thus, the following transition is
not possible any more: $(q, v) \xrightarrow{\delta \leftarrow} (q', v')$

**Non-zenoness of runs** A zeno run happens when, after infinite system execution, the
global clock of the system does not advance above a certain limit. This hap-
pens, for example, when there is always a transition available with the urgency
*eager*. Urgencies will be explained later on in this chapter.

**Invariance properties** Only some variants of invariance properties can be checked
using the simulation graph. We can, for example, define a propositional logic
formula that must hold on a certain state $q$ of the automaton. A propositional
logic formula is a formula with which we can define conditions that must hold
in (a) certain state(s), e.g. 'the range a certain clock should be in when entering
a certain state'.

**Timed linear properties** These are more complex properties, that can, for example,
define deadlines of tasks. These properties can be defined using propositional
logic formulas, but can also be defined using automata [31]. To express timed
linear properties in regard to a finite execution of the system, one can use finite
automata to express these timing requirements. To express properties regarding
infinite timing requirements, one can use Büchi automata. A Büchi automaton
is an extended finite automaton which accepts infinite inputs.

Now that we know that the temporal requirements of an SDL model can be verified
using timed automata, we need a to map the SDL model to the timed automaton. But
first, I will shortly introduce the various tools that are out there to either design SDL
models, or to verify timed automata.
4.3 Selection of tools

4.3.1 SDL design tool: Real-Time Developer Studio

The tool, that was chosen to design the actual SDL and MSC models, is the Real-Time Developer Studio, in short RTDS. RTDS is created by PragmaDev\(^1\), and was first introduced in 2001. PragmaDev, located in Paris, France, is currently the leading provider of graphical modelling tools based on SDL and MSC.

RTDS provides a graphical user interface which give the developer the opportunity to manage the project, to design the models, to simulate them and to generate code from them. Or, in other words, RTDS can be used through the entire development cycle, which was the main goal of SDL and MSC in the first place, as described in chapters 2 and 3.

With RTDS, the developer can choose to either use the SDL standard as described by ITU\(^2\), or he can choose to use the SDL-RT standard\(^3\). This is so, because there were some problems that arose using the original SDL standard during the design of real-time (embedded) systems. First, there was no real SDL compiler, which made it hard to actually use the designed model on the target hard- and software. Second, it was not described in the standard how we could use external C APIs, real-time operating systems libraries, protocols, etc. Third, some common and useful constructs, such as semaphores, were not available in the original SDL standard.

For these reasons, it was preferred to extend the SDL standard, forming SDL-RT. SDL-RT is an extension of the SDL standard, and therefore it has all the benefits of the original standard. Besides this, SDL-RT has a more C-like behaviour, which makes it possible to use C code inside the model. It also adds the use of semaphores to the original standards. Furthermore, it supports classical real-time concepts, such as timers.

As mentioned before, PragmaDev allows the developer to also actually compile the SDL-RT model to C code. For this, different compilers, for different target operating systems, are available, e.g. POSIX\(^2\), which produces C code that is optimised for, and makes use of the libraries of the specific target operating system.

Since SDL-RT is closely related to the original SDL standard, and it basically just adds some features, as described above, all timing information and requirement extensions as described in chapters 2 and 3 are still applicable, and it makes no difference for the eventual verification using timed automata.

4.3.2 Selection of verification tools

There exists a variety of specialized tools with which we are able to verify temporal and non-temporal properties of timed automata. I chose not to create a verification technique by myself, as these tools were created by people with many years of experience in the field, and the tools underwent years of development and fine-tuning. Furthermore, the goal of the thesis is to be able to verify concurrent, asynchronous real-time systems, and not to understand how the actual verification works in all its

\(^{1}\)PragmaDev: www.pragmadev.com

\(^{2}\)POSIX, or Portable Operating System Interface, is a widely used standardized interface between the user/software and its operating system
details. However, you might be interested in different approaches of verification. For this, I refer to the TLA+ verification method, as described in [36].

I will shortly discuss several possible verification tools, after which I will conclude this section with my preferred choice. I will only discuss verification tools that use timed automata as the system representation. There are other verification tools, such as HyTech[5][53], that use linear hybrid automata. These automata allow the specification of clocks that can have varying clock rates, and clock rates between certain intervals. However, [5] states that a linear hybrid automaton can be transformed automatically to a timed automaton. Therefore, it is sufficient to only examine verification tools that use timed automata. Next, we will discuss the tools Real-Time Spin, Kronos and UPPAAL.

**Real-Time Spin**

**Spin**  Spin is a verification tool which can verify asynchronous, concurrent software systems[26][52]. With Spin, we can perform a random simulation, a partial correctness verification, or an exhaustive, total correctness verification upon the system at hand[55]. A system is found to be correct, when there are no violations against assertions, there are no deadlocks, and all linear-temporal-logic formulae (LTL) are satisfied.

To describe the entire system specification, Spin uses the Promela language[52]. The system specification is made up out of two parts. One part describes the system itself, and is made up out of processes, channels and variables. These processes act asynchronously with each other, and communicate via the channels. The second part is the so-called never-claim, or the property specification part[52]. It acts as a normal process, but the main difference is that there can only be one never-claim per system specification, it does not participate in the actual system execution, and it interacts with other processes in a synchronous manner. The never-claim is used to assert that the system does not violate certain conditions[26][52].

Spin also uses Linear Temporal Logic, or LTL, to specify claims about the system, which need to be checked for correctness. LTL formulae allow for more detailed claims about the system. The semantics of LTL is summarized in appendix B.1. These formulae allow nesting. These formulae also permit us to specify qualitative conditions over time, which might state that something has to hold until something else eventually holds.

To verify a system, Spin will have to do the following steps[26]. First, it needs to translate every process into a finite automaton. Second, it will asynchronously multiply these automata, to get the state space of the entire system, also known as the simulation graph. Third, it needs to translate each LTL formula in a separate Buchi automaton. After these three steps, Spin will synchronously multiply the state space with each LTL Buchi automaton, which results in another Buchi automaton. It now turns out that the LTL formula is satisfied, if and only if the language of the Buchi automaton is empty[26].

**Real-Time Spin**  As was mentioned in the previous section, it was only possible to specify qualitative conditions over time, which is clearly insufficient to verify real-
Static Verification of the SDL model 4.3 Selection of tools

time systems, as these require much more detail. For this reason, Verimag\(^3\) developed Real-Time spin, or rt-Spin\(^2\)[52][55].

For rt-Spin, the Promela language was extended to rt-Promela, to allow the specification of quantitative time conditions, such as, for example, event B should occur 3 time units after event A. This was made possible due to the addition of clocks to the Promela language\(^2\)[52]. Rt-Spin only uses abstract time, so there exists no notion of milliseconds, minutes, hours, etc.

The transformation of the processes described in the rt-Promela language will lead to a timed (Buchi) automaton.

Unfortunately, there is no real-time variant of LTL that can be used in rt-Spin, since the real-time version of LTL is not always decidable, which results in the fact that it is not always possible to translate rt-LTL into a timed automaton, as was possible for regular LTL formulae. This leads to the fact that all time related conditions have to be put in the never-claim, so they can be translated to a timed automaton, which can be used for verification purposes, as described in the previous paragraph.

Kronos

Kronos is a verification tool, also developed by Verimag\(^7\)[59]. Kronos uses timed automata to represent the system at hand, and therefore it is a possible candidate to verify systems designed in SDL.

Kronos offers two types of verification methods, being logical verification and behaviour verification\(^59\). These two methods will be discussed now.

Logical verification Logical verification will be achieved using TCTL\(^4\) formulae. Kronos is able to use the entire spectrum of the TCTL language, including nesting of TCTL formulae. A description of the TCTL language can be seen in appendix B.2.

With the TCTL formulae, we can state properties of the system concerning reachability of a certain set of states, e.g. to check safety properties, to check for non-zenoness of runs, and to check for bounded response\(^59\).

For the logical verification method we can use three different symbolic verification techniques.

1. **Forward symbolic analysis** For this, we will do a forward exploration of the simulation graph to find all possible successors of a given set of states\(^7\)[59]. After this, we again can use the result to verify whether a certain TCTL formula is correct.

2. **Backwards symbolic analysis** For this, we will use the simulation graph to find all possible predecessors for a given set of states\(^7\)[59]. After this, we can use the result of the backwards analysis to check whether a certain TCTL formula is correct.

3. **On-the-fly** With this technique, we will not first build up a graph on which we can perform the verification afterwards, as was the case for the forward and

\(^3\)Verimag is a French academic research laboratory, website: http://www-verimag.imag.fr
\(^4\)TCTL\(^{[17]}\): Time-Computational Tree Logic)
backward analysis\cite{5}. In stead, we will build up the required graph while verification moves forward. Therefore, this technique is called the on-the-fly method.

These three techniques allow us to verify the temporal aspects of our system. Either the TCTL formula is verified to be correct, or not correct. In the latter case, we can view a diagnostics trace, which shows an example of a certain system execution for which the formula was found to be invalidated.

**Behavioural verification**  For behavioural verification, the tool will verify whether or not the behaviour of the system is what was expected\cite{59}.

The verification process will happen in three phases. First, we need to build up an abstract model of the system, which represent its very basic behaviour. Second, all transitions in the implemented automaton that originate from certain implementation issues are marked. Since these transition have nothing to do with the behaviour of the system itself, they are considered *non-observable*, and can therefore be hidden. Third, all exact timing information will be discarded as well, so that only causal relationships between states remain.

During behavioural verification, we will now have to check whether the abstract automaton we obtained from the first phase, and the stripped down automaton we obtained from the second and third phase, are *bisimilar*. This means that their behaviour is equivalent, and thus as expected. It is clear that this method is not sufficient for quantitative verification of time requirements.

**UPPAAL**

UPPAAL is a simulation and verification tool created by the Department of Computer Systems of the Uppsala university and the BRICS university of the Aalborg\cite{32}\cite{34}\cite{35}, hence the acronym UPPAAL.

UPPAAL uses timed automata to represent the system, as was the case in the Kro-nos tool, and it has extended these automata with more general data types, such as booleans and integers. The timed automata also have clocks which will all move at the same rate, and which represent an abstract notion of time. The latter implies that UPPAAL does not use time unit, such as e.g. seconds, milliseconds, etc.

The state space of the system, which is the combination of all the timed automata, is symbolic. This means that we will never work with the actual representation of the state space, but only its symbolic equivalent. Therefore, a state in the state space is made up out of the collection of all the clocks and data variables of all timed automata of the system\cite{34}.

In earlier versions of UPPAAL, verification was done by either a backward or forward reachability analysis\cite{32}, techniques that are still used in the Kronos tool, as described in section 4.3.2. However, UPPAAL now only uses an on-the-fly verification technique\cite{5}. This means that the state space will be build up during verification. Should verification fail, then the current state space can be viewed in the simulator.

The verifier of UPPAAL uses temporal formulae, which are a restricted subset of TCTL formulae\cite{5}\cite{32}\cite{34}. They are restricted in such a way that nesting of formulae is not allowed. The syntax of the TCTL formulae can be seen in appendix B.2.
UPPAAL: the tool of my choice

I have examined the pros and cons of each tool, which were described in the previous section of this chapter, and I have chosen to work with UPPAAL. My choice is based on different factors, which will shortly be discussed next.

Translation from SDL to UPPAAL/Kronos and rt-Spin  As was shown in the previous section, it became clear that rt-Spin uses a system representation which is written in the rt-Promela language. This means that a parser is required to transform the timed automata, obtained from the SDL model, to RT-Promela programs. Such a parser already exists, created by the IF project of Verimag. IF[42], or the Intermediate Format, is a universal, high-level language, based on extended timed automata[44], to represent asynchronous timed systems. The main goal of IF is to be an intermediate language between design tools and verification tools. This would mean that any design language could be translated to the IF format, and this IF model could be translated into any language used by a certain verification tool. The main goal of the IF project is thus clearly to make it possible to validate any real-time system, designed in a certain language, using every possible verification tool available[41].

There exists a parser to translate IF code to Promela code, but this solves only half of our problem, as we also need to translate our SDL model to IF. Since the design program we use to create the SDL model, being the Real-Time Developer Studio, is not (yet) supported by any IF parser, I have decided not to use rt-Spin as the verification tool of my choice.

Furthermore, should we use rt-Spin, we would have to translate all time related properties of the SDL model to never-claims, which would be a burden.

On the other hand, since UPPAAL and Kronos use actual timed automata as the presentation of the overall system, it will be easier to translate the timed automata model of the SDL model to its counterpart in UPPAAL or Kronos.

UPPAAL vs. Kronos: performance  Also important for future use, is how fast the tools can verify the time requirements of the system, if they can verify them at all, due to, e.g., a state space explosion. As [5] shows, it turns out that UPPAAL is much quicker and can verify larger systems than Kronos, using the on-the-fly method, as described above. However, Kronos performs much better than UPPAAL using the forward and backward analysis methods, methods that are not available in UPPAAL.

UPPAAL vs. Kronos: updates  Although it seems now that Kronos is much faster than UPPAAL, and thus seems to be the better pick, one also must know that these benchmarks were taken in the year 2000, more than seven years ago. Since then, Kronos was updated only in 2002, while UPPAAL had its last update march 2007. According to the release notes, big improvements were made in runtime and memory consumption. This fact, together with the actuality that I was already familiar with the use of UPPAAL, and that it turned out to be moderately easy to translate the SDL model to a UPPAAL representation of the timed automaton, I chose UPPAAL as the verification tool to be used.
4.4 Parsing the SDL model to its counterpart in UPPAAL

As mentioned in section 4.3.1, we will use the Real-Time developer studio, in short RTDS, to design the SDL and MSC models of our real-time systems. In chapter 2, we already were introduced to the different components that are used in SDL, and now I will shortly describe how these components can be used to build up the first, preliminary model of the system. Next to this, I will also describe how the counterpart of the SDL system looks like in the UPPAAL tool. I will then continue to describe how the timing information in the SDL mode, as specified in chapter 2, can be translated to the UPPAAL model. I will conclude this section by discussing my own SDL2UPPAAL parser, which will translate the SDL model, together with its timing information and requirements, to its UPPAAL counterpart.

A lengthy description of the different elements of UPPAAL, together with the different options that are available during verification, and the consequences of these options, can be found in appendix C.1 and C.2, respectively.

4.5 Mapping SDL to UPPAAL

The SDL model will consist out of five main parts, being the start symbol, the state symbol, the input symbol, the update symbol and the decision symbol. An example of a model designed in RTDS can be seen in figure 4.1(a), and its UPPAAL counterpart can be seen in figure 4.1(b).

As we can see, the initial state of the UPPAAL model is defined by the start symbol that is connected to the state labelled \textit{start}. RTDS requires that there can only be an input symbol behind a state symbol, because a system can only change states after a new input is received. Thus, we can see in figure 4.1(a) that we have an input \textit{message} after the input state. This can also be found in the UPPAAL model. Next, we have a decision symbol. This represents a guard in the UPPAAL model. The decision symbol splits the path in two ways, one for the expression to be true, and one for it to be false, thus resulting in two separate transitions in the UPPAAL model. Should the expression of the decision symbol be true, then in this case we will update the \textit{i} variable, and proceed to the \textit{stop} state. Otherwise we will return to the \textit{start} state. Two things are important here. First, it is not possible in RTDS to create loops via linkage, as is possible in UPPAAL. Therefore, we define the same state multiple times, as can be seen in figure 4.1(a). Second, to make the translation to UPPAAL more smoothly, we will use the UPPAAL notation to describe an update, for which multiple updates of multiple variables and clocks are possible in the same input symbol, as long as they are separated by use of a comma[34][9].

As mentioned before, it is obligated for a state to be followed by an input symbol, if any. Otherwise task and decision symbols can be mixed freely.

4.5.1 Timing requirements defined in RTDS and UPPAAL

The goal of this thesis is to verify real-time systems, in early stages of the design phase, which are represented in SDL and MSC. Chapter 2 already described how we can specify temporal requirements, and now we have to examine how these timing requirements can be used, or can be represented, in the UPPAAL model, and how we
can translate these requirements from their specification in RTDS to its representation in UPPAAL.

**The system clock**

Since SDL had a weak model to represent time, being the now variable, it was decided that we had to introduce a new concept of time. We did this by means of a clock definition, as described in section 2.2, using the annotation technique. An example of such a clock definition in an SDL model, designed using RTDS, can be seen in figure 4.2.
The clock has different characteristics, of which some will be used in the UPPAAL model, and some will not. The clock itself is not actually represented in the UPPAAL model, for the simple reason that the purpose of it was to create a solid base of time progression in the SDL model, and UPPAAL already has a strict model of time and time progression. Why it is then, that we need the clock, will be discussed next.

**Drift** Drift defines whether or not a clock diverges from the ideal clock, as discussed in section 2.2.1, and is expressed in parts per minute. There are two questions to be asked here. First, is it really important to take drift of a system clock into account, when we are in the beginning of the design phase? This depends on the system at hand, since for systems that will run for e.g. a couple of years, drift might become an important factor. On the other hand, for systems that run for only a few minutes, or even a couple of hours, drift might be negligible. Second, how could we incorporate drift into the UPPAAL model? Drift means a divergence from the ideal clock, and therefore all deadlines and durations will be affected by it. A solution could be to change all deadline and duration variables every minute. This however would make the model very complex, while its influence can be regarded as negligible. Therefore it is chosen not to deal with drift of the system clock.

**Offset** Offset is the starting value of the clock, as described in section 2.2.1. Since we do not use the actual value of the clock anywhere, especially since we only use the clock to define a strict notion of time progression, in the design phase, and all references to it are relative, we can ignore this factor. However, offset might become important during the programming phase, but that is out of the scope of the thesis.

**Granularity** Granularity of the system clock will define the actual speed of the processor. For example, in figure 4.2, we have defined a system for which the processor runs at 50 MHz, or 20 ns per clock cycle. We will use the granularity of the clock to scale all time values up or down, so that they are relatively equivalent, and they can be used in the UPPAAL model.

**Range** is considered not important, since the actual value of the clock is never used, in the design phase. In UPPAAL, clocks can go on until infinity, theoretically speaking, and thus we do not need a representation of the range of a clock in UPPAAL.

---

5UPPAAL has an abstract notion of time, so this value will need to be scaled up or down to match exactly one minute, according to the granularity of the clock, which will be discussed later on in this section.
Static Verification of the SDL model 4.5 Mapping SDL to UPPAAL

Urgencies

As discussed in section 2.3 of chapter 2, it was also necessary to be able to specify urgencies of states in the SDL model. Three kinds of urgencies were proposed, being lazy, where time might pass indefinitely, eager, when no time is allowed to pass, and delayable, which represents a combination of a lazy and an urgent transition.

In section C.1.1, we have discussed that UPPAAL has a sense of urgency, meaning that a state can either be lazy, urgent or committed. This simplifies the representation of urgencies, as defined in RTDS, in an UPPAAL equivalent model.

The representation of a lazy state in RTDS, and its equivalence in the UPPAAL model can be seen in Figure 4.3(a) and 4.3(b) respectively.

![lazy state in RTDS and UPPAAL](image)

(a) A lazy state in RTDS  (b) A lazy state in UPPAAL

Figure 4.3: Representation of a lazy state in both RTDS and UPPAAL

The representation of an eager state in RTDS, and its equivalence in the UPPAAL model, as an urgent state, can be seen in Figure 4.4(a) and 4.4(b) respectively.

![eager state in RTDS and UPPAAL](image)

(a) An eager state in RTDS  (b) An urgent state in UPPAAL

Figure 4.4: Representation of an urgent state in both RTDS and UPPAAL

To represent a state that is delayable, we have to follow a slightly different approach, since there is no direct representation of such a type of state in UPPAAL. We can represent a delayable state as depicted in Figures 4.5(a) and 4.5(b) in the RTDS and UPPAAL tool respectively.

With this method, we make sure that the state cannot be left before 5 time units, using the guard of the transition in UPPAAL, and that the state will be left before 20 time units, using the invariant of the state. This way we can simulate a delayable state in UPPAAL.
Event

In section 2.4.3 of chapter 2, it was also deemed necessary to be able to specify events in the system. As events will only be used to represent a certain point in the system, which then can be used to specify a maximal and minimal duration between two events, there will not be an explicit representation of events in the UPPAAL equivalent model. Events will only be handled internally.

Duration

We must also be able to specify duration, or, in other words, how long we can stay in a certain state. How this can be expressed using RTDS can be seen in Figure 4.6(a). Expressing duration of an event will be similar to how we expressed delayable states. This can be seen in Figure 4.6(b).

Channel delay

As described in section 2.4.5 of chapter 2, channels can also have a certain channel delay, which is the time it takes to send a message over the channel, a loss rate and whether the channel is ordered or not.

Delay can be represented in UPPAAL by adding an extra state, which represents the channel, and using the same techniques as for delayable states/transitions and duration of events.
Loss rate can be represented by UPPAAL by increasing the channel delay by the loss rate percentage. This way we will also use the fact that messages can be lost in our verification.

Ordered might increase the overall delay of sending messages over a channel that does not assure a first-in first-out delivery of messages, since a message should only be delivered when all previous messages sent are delivered. However, this delay is undeterminable by just saying whether a channel is ordered or not, and it will not be accounted for in the UPPAAL model.

An example of channel delay, together with its loss rate, can be seen in figures 4.7(a) and 4.7(b), for the RTDS and UPPAAL model respectively.

![Figure 4.7: Representation of channel delay in both RTDS and UPPAAL](image)

As we can see, the channel delay needs one extra state between the start state and the end state. Going to the channel state, we reset the channel delay clock. The invariant of the channel state will make sure that it simulates a delay, in this example, of 20 time units + 5%, or 21 time units.

Span

In section 2.4.4, we have said that we can only indicate timing requirements using the span construct. This span construct states that, in between the occurrence of two events, there should not be more than a specified amount of time. The fact that this is an temporal requirements, it should also be verifiable, which is the main purpose of this thesis assignment, and the reason why we are discussing the UPPAAL tool in the first place.

However, verifying the span is not a trivial task, as there arise two problems. The first problem that comes up, are states that have no defined urgency. It is not directly
clear whether to treat these states as lazy or urgent, since there does not exist a guideline about this. There are two possible solutions. Either we let the user of the parser set a flag, whether to treat the states with unspecified urgency as lazy or urgent, or we consider all states with unspecified urgency as either lazy or urgent. I have chosen to follow the same method as used in UPPAAL, being that we will treat all states as lazy, unless specified otherwise.

The second problem is how we can create a TCTL formula, to verify the actual span. A first attempt was made using a clock per span construct, which gets reset for every transition, except for transitions that are lying on the path between the two events. This turned out to be a bad solution, since we could not specify a minimal time bound, and, furthermore, for states with a duration larger than the span between the two events, the verification would not work, since the span clock would exceed the maximal span.

Therefore, the following, working solution, was found. First, we will mark all possible paths between two events by setting a span variable. Second, on all other paths, the variable will be reset. We now can verify the timing requirement using the following example TCTL formula:

\[
\text{spanVar} == 1 \rightarrow (\text{spanVar} == 1 \text{ and } \text{spanClk} \geq 5 \text{ and } \text{spanClk} \leq 10)
\]

This formula will verify whether, if spanVar becomes set, eventually spanvar will still be set, and the spanClk will have a minimum value not lower than 5, and a maximum value not higher than 10.

An example of this can be seen in Figures 4.8 and 4.9, for the RTDS and the UPPAAL model, respectively.

As we can see, we have marked the span from the RTDS model with a span variable \( \text{spanVar} \) in the UPPAAL model, which is set everywhere on the path between the two events. The span can be verified with the following TCTL formula:

\[
(P.\text{spanVar} == 1) \rightarrow (P.\text{spanVar} == 1 \text{ and } \text{globalClk} \geq 15 \text{ and } \text{globalClk} \leq 17)
\]

This TCTL formula will be verified as correct, since the time between the two events is indeed between 15 and 17 time units. Should we try to verify the following TCTL formula, UPPAAL will find this formula to be invalid, since the minimal time span is not respected:

\[
(P.\text{spanVar} == 1) \rightarrow (P.\text{spanVar} == 1 \text{ and } \text{globalClk} \geq 16 \text{ and } \text{globalClk} \leq 17)
\]

Should we try to verify the following TCTL formula, again UPPAAL will find this formula to be invalid, since the maximal time span is not respected:

\[
(P.\text{spanVar} == 1) \rightarrow (P.\text{spanVar} == 1 \text{ and } \text{globalClk} \geq 15 \text{ and } \text{globalClk} \leq 16)
\]

**Periodicity of external input**

We will not model this in UPPAAL, for the simple reason that only the system itself will be modelled in RTDS. All information about the environment is not present in this model. Therefore, before we can actually verify the model in UPPAAL, we will have to create a model for the environment by hand, and we will thus have to include information about periodicity of external input by ourselves.
Figure 4.8: timing requirements in RTDS

Figure 4.9: timing requirements in UPPAAL
4.5.2 Parsing from SDL to UPPAAL: SDL2UPPAAL

In this section, I will describe the self-written tool that translates the SDL model, created in RTDS, to a UPPAAL model.

First, I will shortly describe the file structure of an RTDS project. Second, I will briefly discuss the xml-structure of both a RTDS file and a UPPAAL file. Third and last, I will describe how the actual parsing of a model is done, and I will highlight and explain the most important pieces of source code of the SDL2UPPAAL parser.

File structure of an RTDS project

In an RTDS project, there is one project file, with the extension .rdp. From this project file, we can obtain the system file of the project, and the names of all processes in the system, together with their filenames. From the system file, we can obtain the channel names, together with the global variables.

RTDS vs. UPPAAL: the internal xml structure

As became apparent from the title, both RTDS and UPPAAL\textsuperscript{6} use xml to represent the models internally. This greatly improves the ability to parse the RTDS model to an equivalent model in UPPAAL. This was one of the reasons why UPPAAL was preferred, instead of tools such as Kronos, as discussed in section 4.3.2.

RTDS: the internal xml structure

RTDS is able to build up an entire model using only two separate structures in xml. These two structures will be described shortly next.

Symbol structure

With the first structure, RTDS can define every symbol that can be drawn inside a model. The structure looks as follows:

\begin{verbatim}
<Symbol symbolId="" type="" xCenter="1077" yCenter="728"
fixedDimensions="TRUE" width="2110" height="1391"
color="#000000" fillColor="#ffffff">
<Description></Description>
<Text></Text>
</Symbol>
\end{verbatim}

As we can see, the structure will first of all define the symbol id, which will look e.g. like symb001, for the first symbol drawn, hence ending in 001. All symbol ids are unique for each process. Next, the structure will define the type of symbol that it represents. The different types of structures that are available can be seen in table 4.1. Furthermore, the structure defines the location of the graphical symbol in the model, using x and y coordinates, together with its dimensions. Last, but not the least important, is the text of the symbol. This field will contain e.g. the expression when

\textsuperscript{6}Xml was introduced in version 3.2 of UPPAAL. The old file syntax, xta, can still be used, but for this thesis we will only work with the xml syntax
Table 4.1: Different types of structures

<table>
<thead>
<tr>
<th>Symbol type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>stlStart</td>
<td>The start symbol</td>
</tr>
<tr>
<td>sdlStop</td>
<td>A stop symbol</td>
</tr>
<tr>
<td>sdlState</td>
<td>A state symbol</td>
</tr>
<tr>
<td>sdlInputSig</td>
<td>An input symbol, representing a guard</td>
</tr>
<tr>
<td>sdlDecision</td>
<td>A decision symbol, containing the declaration of global or local variables.</td>
</tr>
<tr>
<td>sdlText</td>
<td>A text symbol, containing an update of a variable, or the reset of a clock</td>
</tr>
<tr>
<td>sdlTask</td>
<td>A task symbol, which contains an update of a variable, or the reset of a clock</td>
</tr>
</tbody>
</table>

The symbol is a decision symbol, the state name when the symbol is a state symbol, etc.

**Transition structure** With the second structure, RTDS will define the different transitions between the symbols. An excerpt of the xml code shows the form of the structure, as can be seen below.

```
<Link linkId="" type="" textSegmentNum="0"
   color="#000000" reverseRead="FALSE">
   <Text> </Text>
   <Connector attachedSymbolId="" type="void"
      isOutside="TRUE" side="x" position="0.0"
      endType="voidend">
      <Text id="1"></Text>
      <Text id="2"></Text>
   </Connector>
   <LinkSegment orientation="v" length="66"/>
</Link>
```

Like the first structure, this link structure also has a unique link id, which has as form e.g. LINK001, for the first link drawn. There are two different types of links that are important right now, as can be seen in table 4.2. Only for a decision transition the text-field of the structure will be important. In this case the text of the transitions will either be true or false.

The structure also has two connector lines. These declare which two symbols the transition connects with each other. The attached symbol id field will contain the unique symbol ids of the connected symbols.
Table 4.2: Different types of structures

<table>
<thead>
<tr>
<th>Link type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbVoid</td>
<td>For a normal transition</td>
</tr>
<tr>
<td>dec</td>
<td>For a decision transition between a decision symbol and either a task symbol, decision symbol or state symbol</td>
</tr>
</tbody>
</table>

**UPPAAL: the internal xml structure** The xml structure of UPPAAL follows the same concepts of the xml structure used in RTDS. There are, however, some crucial differences. First of all, UPPAAL uses three different xml structures, instead of two for RTDS. Second, only one symbol is used in UPPAAL, i.e. the state symbol. This has as a consequence that all information about variable updates, synchronisation via message input, etc., has to be stored in either the structure that represents a state location or a transition.

**State structure** The first structure, as already mentioned, represents a state location, as can be seen below.

```xml
<location id="id1" x="-136" y="-88">  
  <name x="-128" y="-136">stateName</name>  
  <label kind="invariant" x="-72" y="-32">inv</label>  
  <urgent/>  
  <committed/>  
</location>
```

We can see that each location has its unique id, as was the case for all symbol structures in RTDS. Second, the location of the state is defined by its x and y coordinates. Each state can have a state name, here represented by `stateName`, and each state can have an invariant, here e.g. `inv`. The meaning of an invariant is described in section C.1.1. Furthermore, if a state is urgent, the line `<urgent/>` is included. Should a state be committed, the line `<committed/>` is included. A lazy state is a state where these lines are missing. Note that a state is either urgent or committed, and cannot be both. The xml excerpt contains both, only to show how urgent and committed locations are defined.

**Initial structure** The second structure is the structure which defines the initial state of the system. Where, for RTDS, the initial state is the state that is connected to the start symbol, UPPAAL will use a separate structure for this, as can be seen below. The `ref` field will contain the unique id of the location which is the initial location of the model.

```xml
<init ref="id1"/>
```
Transition structure  The third structure defines the transitions between all state locations of the UPPAAL model. In contrast with RTDS, there is only one kind of transition. The transition structure can contain the guard of the transition, the system input or synchronisation, and possible updates of variables and resets of clocks. An excerpt of such a transition can be seen below.

```
<transition>
  <source ref="id1"/>
  <target ref="id0"/>
  <label kind="guard" x="-64" y="-120">i < 0</label>
  <label kind="synchronisation" x="-72" y="-88">message?</label>
  <label kind="assignment" x="-64" y="-56">i = i + 1</label>
</transition>
```

Here, we can see that the transition connects the state location with id equal to \( id1 \) to the state with id equal to \( id2 \). The transition has a guard \( i < 0 \), requires an input \( message \), and will perform the update \( i = i + 1 \).

We now have a better understanding on how the internal structure of the models are stored in both RTDS and UPPAAL. In the next section, I will describe how the actual parsing will be done, which will translate the RTDS model to the UPPAAL counterpart. I also will highlight the most important, non-trivial parts of the source code. After this, we will discuss how the SDL2UPPAAL parser can be expanded, so it also incorporates the timing information from the SDL model, as was described in chapter 2.

Translating the RTDS model to a UPPAAL model: SDL2UPPAAL

The parsing process occurs in different steps, as can be seen below.
The parser will start by reading information out of the project file and the system file. By doing so, it will obtain all global variables, which can in turn be written to the output file, and all process file names, as discussed in section 4.5.2. Then, for each process, the parser will read the process file, where it will obtain all information of all symbols and transitions within the process, as will be discussed in more detail later on. With this information, the parser will continue parsing, in that first the decision transitions will be parsed, after which the normal transitions will be parsed. Now that we have all information that we need stored in memory, the locations of the process and all transitions between these locations are written to the destination file, using the structures which are comprehensible for UPPAAL, as discussed in the beginning of this section.

**Parsing a process file** The function `parse_process()` will read all the information that is contained in a certain process file. From the beginning of this section, we know that there are two kinds of structures that can be found in a single process file, being the definition of a type of symbol, or the definition of a type of transition. Every type of symbol and transition needs to be parsed differently, and all this data will need to be stored in memory. The outline of the function that does this, can be seen in appendix...
The parsing of these symbols is, as mentioned before, quite trivial, in that data is
read from the process file, and it is stored in memory, using the correct structure that
corresponds with a certain type of symbol or transition. The only parse function that
should be highlighted, is the parsing of a state symbol.

The following things happen during the parsing of a state symbol. As with the
other parse functions, we first obtain the information out of the symbol xml structure.
Then we have to make sure that every state location preserves its uniqueness in the
UPPAAL model. This is so, since in the RTDS model, we cannot create loops in
the model using transitions, as was explained in section 4.4, and which can be seen
in figure 4.1(a). Therefore, we need to check whether we have already encountered
a state with the same state name. If this is not the case, we will add this state to the
uniqueLocations hash table, which contains all unique locations that eventually will be
written to the output file, and we have to add a link from this location id, represented by
the symbol id, to itself to the linkLocations hash. The linkLocations hash will contain
the symbol id of every state encountered in the model, and it links this id to the symbol
id of the unique state location, which will be used in the UPPAAL model.

If it turns out that we have already encountered a state with a similar state name, we
will check whether that unique id was smaller than the id of the current state symbol.
If this is not the case, we will replace the unique id with the id of the current state
symbol. We do this for the simple reason that the first occurrence of a certain state a)
would have the smallest id, and b) would be drawn on the same location in UPPAAL,
should the model have been designed with the UPPAAL tool. This should improve
the aesthetics of the converted model. If we do replace the existing unique id with
the current id, we have to update all the links in the linkLocations hash, to make them
point to the new unique id. Last, we have to create a link between this id and the unique
id. The pseudo code for the parse_state_symbol() function can be found in appendix
D.1.1.

**Parsing the decision transitions**  After we have parsed the process file, we have to
parse the decision transitions. Why this is necessary and not trivial, will be explained
using a small example. Consider the following small excerpt of an SDL model, as
shown in figure 4.10(a).

Here, we can see that each decision symbol will split the path in two, since we
only accept decisions which are either true or false.

In memory, we already have all decision transitions, from the parse_process func-
tion, and we will have to transform these transitions into transitions which start at the
top decision symbol, and end in each leaf. For this, we need to do two things. First, we
need to determine the top decision symbol, which connects the decision tree with the
rest of the model above. Second, we will have to find all paths from the top decision
symbol to each leaf of the decision tree. This leads to the piece of pseudo code as can
be seen in appendix D.2.1.

To determine the tops of the decision tree(s), we will use the following observation. As can be seen in figure 4.10(a), there are two decision transitions connected to the top, there are three decision transitions connected to an intermediate decision symbol, and
for each leaf, there is only one incoming decision transition. Knowing this, we just have to count the connected transitions to a node in the decision tree, to know whether it is a top, an intermediate decision symbol, or a leaf. This is depicted in the pseudo code of appendix D.2.2.

The next step in the parsing of decision transitions, is to find all paths going from each top of the decision tree to each leaf. This will be done recursively. Important to know here is that a leaf will either consist out of a state symbol, or a task symbol. This
is depicted by the pseudo code of appendix D.2.3.

In short, the following is happening here. Starting from each top, we will examine all decision links that start in this top. Should such a decision link end in a state, than we have found a leaf, and we can store this (partial) transition, together with its guard obtained from the decision symbol. The same goes when we encounter a task symbol. If, on the other hand we encounter a decision symbol, we will recursively repeat our actions, in that we will examine each outgoing decision transition, excluding the one we came from, until we find either a state or a task symbol. The beginning of such a transition will be the id of the top decision symbol, the end will be the id of the leaf, and the guard will be a concatenation of all decision symbols that we have encountered on our path, with \textit{and} as the glue between them.

We now have a transition between each top of each decision tree, and all of its leaves. We will now continue parsing the normal transitions

\textbf{Parsing the normal transitions}  Now that we have dealt with the problem that occurred using decision symbols, we will parse the rest of the transitions in the model. The goal will be to only have transitions between states, since these are the only kind of transitions that exist in UPPAAL. This is described in the pseudo code excerpt of appendix D.3

We can have transitions between start and state symbols, state and input symbols, input and decision symbols, input and state symbols, state and stop symbols, input and task symbols, task and state symbols, and task and decision symbols. When we encounter a task symbol, we have to store the update of variables or the reset of clocks together with the transition. The same goes for input messages, as can be seen in the pseudo code.

There are three problems that we can encounter.

- It might be so that one symbol is already part of a transition. In this case we will always expand that transition, to include the other symbol.

- It might also be so that both elements of a normal transition are already part of a transition created earlier. In this case, all information of one transition will need to be copied to the other transition, and the first should be erased.

- When we have to deal with a decision symbol, it is for certain that there exist one or more transitions already going from the top of the decision tree to all of its leaves. This is so, since these transitions have been created by the \texttt{parse\_decision\_transitions} function. Therefore, we will need to either expand all these transition to include the new symbol, or copy all information from the transition, of which the new symbol is part of, to all the transitions from the decision tree.

\textbf{Parser: write locations and transitions}  After everything has been parsed, as described above, we will need to write all the unique locations together with their transitions, to the output file. For this, we will need to adhere to the structure that UPPAAL understands, as it was described in section 4.5.2.
Translating the timing requirements to their UPPAAL counterpart

In the previous section, we have discussed how we can represent the timing requirements that are defined in the SDL model, created with RTDS, inside UPPAAL. Therefore, we will need to expand the SDL2UPPAAL parser, so that it will also parse the timing requirements of the SDL model, and transform them into their UPPAAL counterparts. How this is done, will be described in this section.

Expanding the parsing of a process file  As we have seen in the above section, we first need to parse all process files to obtain all information that is stored in these process files. This was done using the function `parse_process()`. Since we now need to extract even more information out of the process file, being the extra functional timing requirements, we will first need to expand this function. The resulting pseudo code can be seen in appendix E.1.

Since it was decided in section 2.4 of chapter 2, that we would use annotations, or comment boxes, to add timing information and timing requirements, we now need to parse these symbols too. These annotations can be linked to state symbols, when they add information about that state, thus we also need to parse these links.

Parsing a comment symbol is a non-trivial task, as can be seen in the pseudo code of appendix E.2. A comment symbol can contain one or more lines, and each line represents one of two types of information. Either it represents a regular comment, which can be ignored by the parser, or it contains timing information or a timing requirement. The distinction between the two types of information is obtained using the technique as described in section 2.4 of chapter 2, where it is defined that extra functional timing requirements have to be of the form \(\{\text{timing requirement}\}\).

An important point has to be addressed here. Since we already know from section 4.3.2 that UPPAAL uses abstract time, in contrast to the SDL model, we will need to make all timing information abstract. This is done by the `makeTimeAbstract()` function, of which a part can be seen below. All other conversions can easily be made using similar calculations.
As can be seen in the small code excerpt above, we will convert all timing information in the SDL model to abstract time, using the granularity and time unit of the global system clock, which was described in section 4.5.1. Using this technique, we ensure that the relative proportion between all time units in the model is maintained. This will be illustrated by a small example.

Suppose that we have two durations of two states, 400 milliseconds and 1 second, respectively. Also suppose that we have a system clock with a granularity of 200 milliseconds. If we now wish to make the durations of both states abstract, whilst keeping the relative proportions among them, the `makeTimeAbstract` function will perform the following steps:

1. Since the first duration has the same time unit as the system clock, the `makeAbstractTime` function will return 400/200, or simply put the value 2.
2. Since the second duration does not have the same time unit as the system clock, the function will first make sure that the time units are the same. This can be done by multiply the duration by 1000, which makes that 1 second becomes 1000 milliseconds.
3. Next, the `makeTimeAbstract` function will return 1000/200, or the value 5

As one can see, we now have two abstract durations, being 2 and 5 respectively, which have the same relative proportion as their concrete counterparts.

One last remark must be made. As we can see in the definition of the `timeAbstractFunction`, there is an option, so that we can round the abstract time value up or down, while doing the division. Whether to round up or down depends totally on the purpose of the time information.

**Duration of events** For the duration of events, we will always round up. E.g. if we have a duration of 21 milliseconds, and a clock speed of 20 ms, we will have an abstract time of 2. This has as a result that our verification will be pessimistic. But on the other hand, if we have a duration of 39 milliseconds with the same

```plaintext
makeTimeAbstract(timeString, round) {
    If (timeString in ms) {
        If (systemClock in ns) {
            If (round == roundup)
                return ceil(timeString * 1000000 / sysClock.granularity)
            Else
                return floor(timeString * 1000000 / sysClock.granularity)
        }
    }
    ...
}
```
system clock, and we would round down, we would get an abstract time of 1, which is too optimistic. For verification purposes, it is always better to have false-negatives instead of false-positives. Therefore we will always round up.

Span For the same reason as above, being that it is better to have a pessimistic model that passes the verification, than to have an optimistic model that might contain false-positives, we will round the minimal span bound down, and the maximal span bound up.

Parsing the comment links is a trivial task, as the symbol id for events, durations and urgencies, that currently have the id of the comment symbol, have to be changed to the ids of the actual state symbol. This way we can resolve the duration and urgency of state location, and we can identify which event is related to which state.

Parse the non-functional timing information and requirements Now that we have acquired the timing information and timing requirements, we have to add these to the UPPAAL model. For this, we have added three functions to the SDL2UPPAAL parser, as can be seen in appendix E.3.1.

Add urgencies For the add_urgencies method, we will set the urgency Boolean to true, for each unique state location that has an eager urgency. This can be seen in the pseudo code in appendix E.3.2.

Add durations For the add_durations function, we will have to set the invariant of that state to an equation of the form \( \text{durClk} \leq \text{duration} \), and we have to reset the duration clock \( \text{durClk} \) for each incoming transition to that state. How this is done can be seen in the pseudo code of appendix E.3.3.

Add span paths The most interesting function that needs to be added to the parser, is the add_span_paths function. This function will add a span variable to all paths between two events, as was described in section 4.5.1. To do this, we need to find all possible paths between two events, which is similar to the reachability problem, which is how UPPAAL verifies the timing requirements, as is described in appendix C.2. As is explained in appendix C.2, this can be done using the breadth first search and depth first search algorithms. I have chosen to implement the breadth first search algorithm to find all paths between two location states, with some limitations, as will be discussed later on. The pseudo code for the add_span_paths, can be seen in appendix E.3.4.

We will, for all timing requirements defined in the SDL model in the form of span statements, add a path from its begin event to its end event. Since there can be more than one timing requirement in the model, we also need to differentiate the different span variables along the path from each other. This will be done by adding the counter
value at the end of the span variable name. For example, the path of the third span statement in the model will be indicated by the \textit{span3} span variable.

We now will try to find all paths between the two events. The function that does this, will use the breadth first search algorithm, as mentioned before, to find these paths. The pseudo code for this can be seen in appendix E.3.4.

As can be seen in appendix E.3.4, we have a queue of all traces to be examined. This queue is altered a bit in comparison to how it was used in appendix C.2.2. Where the queue in the latter was only used to hold the states that need exploration, the queue in the \textit{add\_path()} function will hold all traces that need exploration. This can be seen in the following example.

Let us reuse the tree we used in appendix C.2.2. Suppose we are looking for a path from the top of the tree to node \textit{F}.

![Figure 4.11: The graph to be explored](image)

In the beginning of the algorithm, the trace queue will hold only one trace, which consists out of the top node. This trace is popped from the queue, and is further explored. This gives us the progression through the search space as can be seen in figure 4.12(a). Since none of the new nodes are the node we were search for, both traces will be pushed on the traces queue, as can be seen in figure 4.12(b).

![Figure 4.12: Breadth First Search and its queue after one round](image)
Since we are still working with the breadth first search algorithm, the first element added to the queue is the next to be examined. This gives us the following search in the tree, as can be seen in figure 4.13(a), with a traces queue as can be seen in figure 4.13(b).

Figure 4.13: BFS and its queue after two rounds

This mechanism will continue. In the next round, it will find a successful trace from the top of the tree to node $F$ which will be put into the successful traces vector. It will examine the other traces as well, but the algorithm will conclude that no more traces can be found to node $F$.

Another important point in the algorithm, which puts a huge restriction on it, is the fact that the algorithm will stop examining a trace when it encounters a state location that it has already encountered before. This means that loops will never be part of the path, which makes the path finding algorithm very restrictive. However, this was done because of the following reason. In the appendix, it is said that the reachability problem is PSPACE-complete. This means that it needs a polynomial amount of memory, and an unlimited amount of time to solve the entire problem, which is to find all paths. This can be shown by the following example, as can be seen in figure 4.14. To find a path from the beginning to the end, the breadth-first algorithm should take the loop at least eight times.

Figure 4.14: Path finding with loops

In the above example, we will have at least seven traces with an amount of elements going from one to seven, that will never reach the end state. The real problem arises when we encounter many loops in a single model, which might involve big parts of the model, and which have to be traversed an unknown amount of time. It is not hard to understand now, that the memory consumption for the reachability problem, using a breadth first algorithm, will require polynomial space, and that it might take an infinite amount of time to find all paths of a big model.
Because of this fact, we will always have to use a restricted form of the algorithm, as e.g. I described it in the pseudo code. Here, all loops will be ignored. Furthermore, more optimal solutions for the path finding problem is out of the scope of this thesis.

A last important point of the pseudo code, is that we restrict ourselves to only expand traces we can actually take. This means that we will only take transitions for which the guards are not validated by any of the variable assignments we have encountered previously on that trace. To manage this, we have two important functions, being possible_transition and add_updates, which will be discussed next.

**Add updates** The add updates function is a function that will parse the updated variables of a transition taken, and will either store the values of new variables, or it will renew the value of an already known variable. There is however a problem that arises here.

Since we do not necessarily search a path beginning at the first state of the model, it might be so that variables have been given a value before the first element of our trace. This is a problem, since this variable might be used during our trace examination, or another variable might be using this variable in an assignment, e.g. \( a = b + 1 \), where the value of \( b \) is defined before the beginning of our trace.

There are three possible solutions:

1. We determine all possible values of all variables, beginning from the first state of our model
2. We give the variables, that were previously defined, all possible values
3. We discard all use of variables that were previously defined.

We have chosen for the third solution, since the first and second solution will be costly in time and space, as they might cause a state space explosion, and are therefore not attractive to use.

With the third solution, another problem arises, in that now we might consider paths that would be impossible to take, if we would consider previously defined variables. This is so, since a certain path might have a guard that should prevent us to take that transition, but since the guard uses a variable which was defined somewhere before our trace, the guard is discarded. This problem will also return when we write the automata to the UPPAAL files, and we will address this problem in more detail later on in this section. The pseudo code of the add updates function can be seen in appendix E.4.

As can be seen in the pseudo code, we will have to check whether or not a transition can be taken. This depends on two factors. Either the guard of the transition uses variables that are not known to the current trace, or in other words, variables that were defined before the trace. These guards will always be validated as true. On the other hand, guards that use known variables have to be validated true. If the latter would validate false, then the transition cannot be taken.

After all successful paths have been found, we only need to mark these successful paths, using the span variable, concatenated with the span number, to make the span variable unique for each span requirement in the model.
**Expand the output of the parser** Although we have added a lot of extra information to the model, the representation of this extra timing information only uses the basic elements of UPPAAL, which makes that the output part of the parser needs very little modification. Two things were added.

First, we were not able yet to write invariants of states. Now we need to be able to do that, since we need invariants to represent, among others, duration of events. This needs to be in conformance with the xml structure of UPPAAL, as described in section 4.5.2. The same goes for urgencies.

Second, we need to build up the TCTL formulae, one for each span requirement in the SDL model. These will need to be written to a TCTL formula file, which must conform to the UPPAAL structure.

However, during tests, we ran into a problem, which made the verification unreliable. This can best be explained using the following small example. Consider an automaton as shown in figure 4.15.

![Figure 4.15: A problem during verification](image)

Here, we can see that we have two traces, of which only one is part of the path, indicated by the span1 variable. If we now want to verify whether the duration of the path is 60 time units, we will use the following TCTL formula:

\[(\text{span1} = 1) \rightarrow (\text{span1} = 1 \text{ and spanClk} >= 60 \text{ and spanClk} <= 60)\]

On first sight, this formula should be validated as correct, since, in the automaton, we have two states on the path with each a duration of 30 time units, or 60 time units in total. However, UPPAAL will also consider the trace going to the urgent state, for which the clock spanClk will only be 30 time units. This results into the fact that the above TCTL formula is regarded as invalid.

To solve this problem, two solutions are possible. The first solution would be to simply block all traces that are not part of the span path. This would be ok if no other span paths are defined. However, should there be more than one span path, this...
solution will not work, as one span path might then block traces that belong to other span paths. This is undesirable.

A second solution is to simply lift the path from out of the automaton, and write each span path to a different UPPAAL file. This will not have the drawback of the first solution, but it will bring with another problem, as already described above, in that it is not clear what to do with variables that were declared somewhere before the beginning of the span path. Up until this point, no conclusive solution was found for this problem, and it remains an open question. A possible solution would be to examine all possible values of all possible variables at the beginning of each path. However, this will be computationally expensive, and a clear and user friendly implemention of this solution in UPPAAL is yet to be found.

I have chosen to implement the second solution. The values of all variables, that are declared before the beginning of the span path, will simply be equal to their intial values, unless they are changed by hand.

4.6 Test results

In the course of this chapter, we have seen how we can translate the timing require-ments, which we have specified in the SDL model, according to the semantic rules denoted in chapter 2, to their UPPAAL counterparts. For this, I have created the SDL2UPPAAL parser, which was discussed in section 4.5.2 of this chapter. In this section, we will demonstrate both the applicability of the methods I have proposed in this chapter, as well as the correctness of the SDL2UPPAAL parser itself, using a small case study, which will be described next.

4.6.1 The test case

Let us consider the following system. We need to create a controller for a secure, automatic door. The door in question has four main components:

- A numeric keyboard, together with a display. This will allow a person to enter a passkey to open the door.
- An iris scanner, to identify the person who wants to open the door
- The door engine, which will open and close the door automatically
- The door controller, which will control the everything. This, we will need to design by ourself.

The controller has the following functions:

- When somebody wants to open the door, he or she will press the open door button, which needs to be caught by the door controller
- The controller then needs to update the display, so the person in question can see that an iris scan is required
- After the iris scan, the controller needs to update the display again, so the person using the door knows that he needs to enter the passkey
- After the passkey has been entered, the controller has to check both the iris scan and the passkey for correctness
- If one of the two is incorrect, the door stays closed
- Otherwise, the controller has to send a signal to the door engine, so that the door starts to open
- When the door is fully opened, the engine will send a signal back to the controller, and the controller needs to pass this signal along to the person who wants to traverse through the door.
- When the close door button is pressed, the controller must send a message to the door engine, so the door starts to close. No passkey is required now.
- When the door is closed, the engine will send a signal to the controller, so the controller knows that the door is fully closed.

The following timing information is supposed:
- To check whether or not the passkey and the iris scan are correct, will require 30 milliseconds
- The door engine will take 900 ms to open the door
- The door engine will take 100 ms to close the door

Furthermore, there are two timing requirements that come with this case, being:
- When the iris scan and the passkey are correct, the door should be open within an interval of 500 milliseconds and 2 seconds
- In case of emergency, the door needs to be closed again between 1 and 2 seconds, after the passkey has been entered

Now that we have the specifications of the system to be built, we can start to design the system in RTDS. The first draft of the SDL model looks like the model depicted in Figure F.1.

As we can see, we assume that the starting state is closed. Therefore, that state is connected to start symbol. After we have received a signal from the open door button, we will wait for both the iris scan and the passkey. After receiving both, we will check whether or not they are correct. This will take 30 milliseconds, as described above. This is indicated by the duration annotation attached to the state. When one of the security mechanisms fails, the door will stay closed. This is achieved by the decision symbol. On the other hand, when both the iris scan and the passkey are determined to be correct, the door controller will send a signal to the engine, and we are in the openingDoor state. This will last 900 milliseconds, as was described in the specifications, and this is denoted by the duration annotation. When the door is fully opened, the door engine will send a signal to the controller, which puts the controller in the open state, as can be seen in figure F.1. When the close door button is pressed, the controller will send yet another signal to the engine, so the door starts closing. This will take 100 milliseconds. After the door is fully closed, a signal from the engine will be received by the controller, and we are back to our start state.
4.6.2 Verification

Now that we have the SDL model, we will use the SDL2UPPAAL parser so we can validate the two timing requirements, which are described in the specifications. After we have let the parser run, we get output in the console, as can be seen in figure F.2.

Here, we can see that the parser will first read the system file, out of which it obtains the filename to the process file. It then parses the process file, of which it parses the decision transitions, the normal transitions, after which it adds the span paths, urgencies and the durations. After this, it will write all span paths to their separate files, together with the TCTL formula, as was described in section 4.5.2. This gives use four files, begin spanAtoB.xml, which will contain the span path from event A to event B of Figure F.1, spanAtoB.q, which will contain the TCTL formula of this path, spanAtoC.xml and spanAtoC.q. These will be shortly discussed next.

Verifying spanAtoB.xml

As mentioned before, this file will contain the marked path from event A to event B. With this path, we will be able to verify whether or not the door will be fully open within a time interval going from 500 milliseconds to 2 seconds, after both the iris scan and the passkey have been validated to be correct. The resulting path, in UPPAAL, can be seen in Figure F.3.

As we can see, the locations waitingForPassword and open are urgent locations, denoted by the letter $U$ inside the state symbol. This is so, since, in the SDL model, as depicted in Figure F.1, these locations have an annotation that says that these are eager locations, in which no time shall pass. Furthermore, locations checkForCorrectness and openingDoor both have a duration of 30 and 900 time units respectively. This also is in congruence with the SDL model. Last, we can see that the span path is depicted by its span variable, in this case $span1$, as was explained in section 4.5.2 of this chapter.

The UPPAAL model relies on the reception of messages, as can be seen by the light green message names, which end with a question mark, at each transition. Therefore, before we can do the actual verification, we need to create the environment of the system, which will simulate the sending of the messages. The environment of our span path can be seen in Figure 4.16. The sending of a message, in UPPAAL, is depicted by the message name, ending in an exclamation mark.

Now that we have the environment, together with the span path, for which we want to verify a certain timing requirement, we can continue with the actual verification. According to the specifications of the secure, automatic door, we have to verify whether, after both the iris scan and the passkey are found to be valid, the door will open within an interval of 500 milliseconds and 2 seconds. According to section 4.5.2, this can be done using the following TCTL formula:

$$\text{Span1} == 1 \rightarrow (\text{span1} == 1 \text{ and spanClk1} >= 500 \text{ and spanClk1} <= 2000)$$

If we now use UPPAAL to validate this automatically generated TCTL formula, we get the following result, as can be seen in Figure 4.17.

---

7 This is a cleaned-up version of the resulting UPPAAL model of the SDL2UPPAAL parser.
8 Note that UPPAAL uses an abstract notion of time. Since the system clock has a granularity of 1 millisecond, 2 seconds become 2000 abstract time units.
We can see that the verification was successful, as was expected.

To show the correctness of the methods, as described in this chapter, even more, consider the following. We can see, in Figure F.1, that the time it will take for the door to fully open is exactly 930 time units, of which 30 time units are from the calculation of the correctness of both the iris scan and the passkey, and 900 time units are from the actual opening of the door. Thus, we can now alter the TCTL formula, to verify whether the span clock $spanClk1$ will eventually be exactly 930 time units, and not more, not less. That this is so, can be seen in figure 4.18.

\[
\begin{align*}
P0.span1 &= 1 \rightarrow (P0.span1 == 1) \land P0.spanClk1 >= 930 \land P0.spanClk1 <= 930) \\
\text{Property is satisfied.} \\
P0.span1 &= 1 \rightarrow (P0.span1 == 1) \land P0.spanClk1 >= 930 \land P0.spanClk1 <= 930) \\
\text{Property is satisfied.} \\
P0.span1 &= 1 \rightarrow (P0.span1 == 1) \land P0.spanClk1 >= 930 \land P0.spanClk1 <= 930) \\
\text{Property is satisfied.} \\
P0.span1 &= 1 \rightarrow (P0.span1 == 1) \land P0.spanClk1 >= 930 \land P0.spanClk1 <= 930) \\
\text{Property is satisfied.}
\end{align*}
\]

Figure 4.17: Verification using UPPAAL

Verifying spanAtoC.xml

Now that we have verified that the first temporal requirement is correct, we will now try to verify the second time requirements, which states that, in case of emergency, the door should close within 1 and 2 seconds, after a passkey has been entered. The resulting path in UPPAAL can be seen in Figure F.4.
Since we start the span path after the iris scan, the parser will ignore all use of the variable which contains whether or not the iris scan was correct, as explained in section 4.5.2. This results into the fact that all paths after the guard symbol that reach the *closed* state, will be considered as valid, and will, as a result, be represented in the UPPAAL model. The TCTL formula used to verify the temporal requirement is:

\[
\text{Span3} == 3 \rightarrow (\text{span3} == 3 \text{ and spanClk3} >= 1000 \text{ and spanClk3} <= 2000)^9
\]

To do a complete verification, we will need to let the variable `correctIrisScan` represent all its possible values, which in our case are only two: 0 and 1. If the variable is 1, it is easy to see that the verifier will verify the TCTL formula as correct, which can be seen in Figure 4.19.

Figure 4.19: Verification when `correctIrisScan` is 1

However, if we let the iris scan variable be 0, we get the following result, as can be seen in Figure 4.20.

![Figure 4.20: Verification when `correctIrisScan` is 0](image)

Why this is, can be seen in the simulator of UPPAAL, as depicted in Figure F.5. As we can see here that, instead of taking the path downwards, the verifier will take the path to the right. This is so, since we stated, by hand, that the variable, which depicts whether or not the iris scan was successful, is 0, or in other words, that the iris scan failed. This results in the fact that we now reach the *closed* state after 30 time units, which has as result that the TCTL formula, which requires a minimal time of 1000 time units, is regarded as invalid.

It is clear now, that we will not be able to use SDL to validate every requirement. How we will be able to verify such specific requirements will be discussed in the next chapter, using MSC.

### 4.7 Conclusion

In this chapter, we have discussed how we can perform a static verification of a SDL model using the UPPAAL tool. For this, we used the fact that SDL is based upon extended finite state machines, which are related to timed automata, which are, in turn, used in the UPPAAL tool. For this reason, I have started this chapter by giving a formal definition of a timed automaton.

---

*Note that UPPAAL uses an abstract notion of time. Since the system clock has a granularity of 1 millisecond, 2 seconds become 2000 abstract time units.*
Then, we have discussed the various tools that we could use, for both designing
the SDL model itself, and to do actual the verification.

To design the SDL model, we only had the Real-Time Developer Studio available.
Its most important characteristics are:

- Usable in the entire design cycle, going from the beginning of the analysis phase,
to the implementation phase, and eventually to code generations

- Two standards can be used, being SDL and SDL-RT. SDL-RT is an expanded
version of SDL, with which we are able to use C-like behavior, external C APIs,
semaphores, etc.

- Different code generators are available, for different target operating systems

To do the actual verification, I have discussed three tools, which use timed aut-
oma as the representation of a model. These tools were RT-Spin, Kronos and UP-
PAAL, of which the most important characteristics are

- RT-Spin
  - Uses the RT-Promela language to represent the model
  - Uses LTL-formulae to verify qualitative timing requirements
  - Uses the never-claim to verify quantitative timing requirements

- Kronos
  - Uses timed automata to represent the model
  - Uses the entire spectrum of TCTL to verify qualitative and quantitative
    timing requirements
  - Three types of verification techniques
    1. Forward symbolic analysis
    2. Backward symbolic analysis
    3. On-the-fly

- UPPAAL
  - Uses timed automata as representation
  - Uses only a part of the TCTL spectrum: nesting is not allowed
  - Only one type of verification: on-the-fly

Since UPPAAL has been updated the most, and because I already was familiar
with it, I chose to do the verification of the SDL model with the UPPAAL tool.

After I made it clear why I chose these particular tools, I continued to describe
how the different types of timing information and timing requirements, as they are
represented in the SDL model using the semantic rules described in chapter 2, can be
translated to their UPPAAL counterparts. For this, we discussed the following:

- Clocks, and how they will be used to make all time information in the SDL
  model abstract
- urgencies: how lazy and eager urgencies can be represented in UPPAAL, using
  the lazy and urgent options of a state in UPPAAL

- events, and how they will only be treated internally

- Duration and channel delay, and how they can be simulated in UPPAAL using
  invariants of states and guards of transitions

- spans, which represent timing requirements:
  - how span paths are marked in the UPPAAL model, using the span variable
  - how they can be verified using TCTL formulae

- periodicity of external input, and why this will be represented, in the UPPAAL
  model, by the environment

After this, we have discussed the most important points of the SDL2UPPAAL
parser, which will translate the SDL model, together with all of its temporal informa-
tion, to its UPPAAL counterpart. Here, we also discussed the problem I encountered
during verification, in that it is unclear how to deal with variables that were declared
before the beginning of the span path. This remains an open question.

I have concluded this chapter with a small test case, which shows the correctness
of the SDL2UPPAAL parser, as well as the correctness of the proposed verification
method. During this test case, it became clear that we will not be able to verify all
temporal requirements in SDL, and it revealed the need for MSC models, alongside
the SDL model, if we want to verify every possible temporal requirement.
Chapter 5

Static Verification of the MSC Model

In the previous chapter, we have discussed how we can verify temporal aspects of the SDL model, using timed automata and the UPPAAL tool. However, as is was stated in chapters 2 and 3, for larger systems, it is also required to use MSC, next to SDL, to be able to fully describe the behaviour of the entire system in all its details. As this behaviour, as specified in MSC, incorporates timing information, we also need to be able to verify these timing requirements. How we might go about this, will be discussed during the course of this chapter. First, we will describe two possible techniques, with which we can verify the time consistency of both basic MSCs and the MSC specification. Second, I will shortly describe the implementation of one of these techniques, after which I will conclude with some test results, which will demonstrate the correctness of the proposed method. MSCs, as described in chapter 3, have a partial ordering of events. So, before we should check the time consistency of MSCs, we first need to check if the partial ordering of events is respected. How this is done is described in [15], and is beyond the scope of this document.

5.1 Time consistency in bMSC and MSC specifications

5.1.1 Consistency of temporal properties of a single bMSC

To check the consistency of a single bMSC, we will shortly examine two very different techniques. The first technique is based on temporal constraint networks[24]. The second technique is a technique that I toughed up by myself. Here, we will verify the time consistency of a bMSC using the UPPAAL tool, which was also used to verify SDL models, as described in chapter 4.

Consistency analysis using the distance graph

BMSCs that are extended with temporal requirements must also be consistent regarding these temporal requirements, as explained in the beginning of this chapter. This means that, for example, if there is a requirement that states ‘a message must be received within 3 time units’, then the delay of the channel where the message is sent
Static Verification of the MSC Model  5.1 Time consistency in bMSC and MSC specifications

over, should not be more than 3 time units, to ensure temporal consistency. How this
can done is described in [24][15], and will be discussed shortly in this section.

To validate a bMSC, we will need to build the distance graph $T_g(M)$ of this bMSC. The
distance graph is a directed, weighted graph, and can be constructed using two
simple rules:

1. each node in the bMSC is represented by a node in the distance graph $T_g(M)$
2. each event edge, signal edge and temporal edge, of the bMSC, that goes from
   node $e$ to $e'$, labelled with a BCET and WCET time, is represented by two
   labelled edges in $T_g(M)$
   - an edge going from $e \rightarrow e'$, which gets as weight the negative value of the
     BCET of the corresponding edge in the bMSC
   - an edge going from $e' \leftarrow e$, which gets as weight the WCET of the corre-
     sponding edge in the bMSC.

An example of how a bMSC is transformed to a distance graph can be seen in
Figure 5.1 [16]. Here, the e0 node is added to make the graph suitable for absolute
timing requirements.

![Figure 5.1: From bMSC to the distance graph](image)

The bMSC is regarded as time consistent, if there are no cycles with negative cost
[24][15]. This can be verified, for example, using the Floyd-Warshall algorithm, which
will calculate the all-pairs shortest-paths of the distance graph. The drawback of this
method is that it is computation expensive, since it has a complexity of $O(n^3)$, $n$ being
the number of nodes in the graph.

**Consistency analysis using the UPPAAL tool**

It seems that not a lot of time has been spent, by the academic world, on the verification
of the time consistency of message sequence charts, as the only technique that can be
found in literature is the one described in the previous sub-section. Therefore, I chose
to challenge myself to examine other possibilities of verification, which I found in
the UPPAAL tool. I will explain my approach using a small example. Consider the
following bMSC, which can be seen in Figure 5.2.
In this bMSC, we can see the simple behavior of an automatic door. When someone wants to open the door, he or she can press the button. The button controller will then send a signal, in the form of a beep, to the person in question, to let him know that the door is opening, and it also will send a message to the door engine, so that the door actually starts to open. When the door is fully opened, the engine will send a message to the button controller, who, in turn, will send a signal to the user, to indicate that the door is open, and that he or she can traverse through it.

We can identify two separate traces in this simple bMSC, as it is depicted in Figure 5.2. The first trace starts at the person who presses the button, and ends with a signal to that person, to indicate that the door is opening. The second trace also starts at the person in question, and ends with the signal that indicates that the door is open. The idea of the verification technique will thus be to a) represent all traces of the bMSC in the UPPAAL tool, and b) to add the necessary timing information, which is described in the bMSC, to the UPPAAL model, so that we can verify whether all timing requirements hold for all traces.

Since it was more interesting for me to test my own verification technique, I have chosen not to implement the technique as described in section 5.1.1. Instead, we will use a technique similar as the one described in chapter 4, in that a parser is needed that will translate the bMSC model to its UPPAAL counterpart. How the actual implementation is done, will be described in section 5.2, but first, I will discuss how we can verify an entire MSC specification.
5.1.2 Verifying the time consistency of an entire MSC specification

As was discussed in chapter 3, it is clear that, for huge and complex systems, one bMSC is not sufficient. Therefore, such systems will have many bMSCs, which can be structured using HMSCs. This is also called the MSC specification. Thus, we will need to be able to verify the entire MSC specification for time consistency.

The hierarchy of the MSC specification can be unwound to a basic MSC, which can then be verified on temporal consistency, using one of the two techniques described above. It is stated in [24] that a MSC specification can be

- time consistent, if all paths that start from the start node of the MSC specification are time consistent
- partially time consistent, if some paths are time consistent, but others are not
- time inconsistent, if all paths are time inconsistent

There are two problems that can arise during the unwinding of such an MSC specification:

1. The MSC specification must present a regular language\(^1\)[13]. If the MSC specification is not regular, then we cannot check the temporal consistency. A MSC specification is definitely regular, if the specification is bounded. We say that a MSC specification is bounded if there exists no process in a cycle that sends messages, but does not receive messages from other processes in the cycle. How this is verified is beyond the scope of this thesis.

2. Loops and iterations in HMSC, as can be seen in Figure 3.2 of chapter 3. These loops can lead to infinite paths, which makes it seemingly impossible to check on time consistency. This problem will be addressed next.

The solution to the above problem, being how to handle the unwinding of an HMSC into its bMSC equivalent, when there are loops in the MSC specification, is to limit the number of iterations taken during the unfolding of the loop[24]. At first sight, this might imply that we do not verify the entire behaviour of the system, but only a part of it. However, as proven by [24], an MSC specification will be timing consistent if:

- each finite and each closed infinite sequential component of the MSC specification is timing consistent
- each infinite sequential component in the MSC specification has matched timer events

A finite sequential component is a string of bMSCs that is finite, or in other words, which contains no loops. A closed infinite sequential component has a path of the form \(n_1, n_2, \ldots, n_j, \ldots, n_k, n_j, \ldots, n_k\), where \(n_1, n_2, \ldots, n_j, \ldots, n_k\) is an infinite component, in that this path is created with the use of a loop in the HMSC [24], \(n\) being a bMSC.

\(^1\)A regular language is a language that represents a possible infinite set of finite sequences of symbols from a finite alphabet
5.2 Verifying the timing consistency of a single bMSC

As was described in section 5.1.1, I have chosen to implement my own technique to verify bMSCs, using the UPPAAL tool. I will only concentrate on single bMSCs, since the verification of an entire MSC specification is trivial, as described in section 5.1.2. Furthermore, I have chosen to implement only the basic subset of timing information and requirements, which will be discussed next.

5.2.1 Timing requirements

As mentioned before, I will only implement the basic timing requirements. This is so, since it turns out that, while using only these basic timing requirements, we are able to describe the timing behavior of most basic systems.

The system clock

As was the case for the SDL model, we will use the system clock to replace the now variable, as the latter represented a very weak model of time. As was the case for the SDL model, as described in section 4.5.1 of chapter 4, we will only use the system clock to scale all exact time information to its abstract counterpart, so this information can be used in the UPPAAL model.

The representation of the system clock in the bMSC model is the same as can be seen in figure 4.2 of chapter 4.

Events

As was described in section 3.1.5 of chapter 3, it is useful to be able to specify timing requirements between events that do not necessarily reside on the same lifeline, and that are not visually ordered, one after the other. For this, it was deemed necessary to be able to specify events. As events only specify important locations in the bMSC, which are needed for internal purposes, they will not be explicitly represented in the UPPAAL counterpart of the bMSC model.

Channel delay

Chapter 3 also identifies the need to be able to specify channel delay. The representation of this in the UPPAAL model will be the same as described in section 4.5.1 of chapter 4.

Duration

As was the case in SDL, we also need to be able to specify duration in the bMSC model, as described in section 3.1.6 of chapter 3. This was so, since I made my case that applying the synchronous hypothesis is not a good idea, when verifying for timing requirements.

The representation of duration, in UPPAAL, is the same as was the case for the SDL, as described in section 4.5.1 of the previous chapter.
Span
As it was described in section 3.1.5 of chapter 3, we will use the span construct in the bMSC model, to specify all timing requirements. The translation of this span construct to its UPPAAL equivalent TCTL formula will be the same as described in section 4.5.1. The only exception is, that we will not have to worry about the urgencies of the locations in the traces of the bMSC, since all locations will be regarded as lazy, with the fact that staying in a location might be limited by an invariant, caused by a channel delay or duration specification.

Timing constructs that are not implemented
As mentioned before, I have chosen to only implement the very basic timing constructs, that were described in chapter 3. But for the sake of completeness, I will shortly describe how the other timing constructs could have been implemented.

Event interval An event interval, as described in section 3.1.4 of chapter 3, is a delay between two consecutive occurrences of a certain bMSC. As came apparent from section 5.1.2 of this chapter, this will only be important during the verification of an entire MSC specification, when we will unfold the HMSC to its bMSC representation. To include the timing behavior described by the event interval, we can simply add an extra duration between the current bMSC and all of its previous bMSCs. This can be achieved in UPPAAL, by adding an extra state between the consecutive bMSCs, with an invariant and a guard to specify the extra duration, similar on how a channel delay or a duration was expressed, as discussed in section 5.2.1.

Processor speed interval The processor speed interval depicts the minimal and maximal duration between two consecutive events in the same processor. It is quite clear that a processor speed interval can be modeled, in UPPAAL, the exact same way as a duration of an event is modeled, as described in section 5.2.1.

Instance delay Instance delay is the delay between two consecutive executions of the same bMSC. We can fairly easily see that this is very similar to the event interval, only now we have a repeated execution of the same bMSC, instead of the successive execution of different bMSCs. Therefore, we can use the same technique, as described above, to represent instance delay in the UPPAAL model.

5.2.2 Parsing from the bMSC to UPPAAL: the MSC2UPPAAL parser
As came apparent from the previous discourse, we will need to translate the bMSC model, together with all its timing information and requirements, to its UPPAAL counterpart. This will be done with the MSC2UPPAAL parser. This is a lot of similarities to what happened in chapter 4, where we discussed the SDL2UPPAAL parser, as will come apparent in the remainder of this section. The main advantage was that we could reuse a lot of code from the SDL2UPPAAL parser, to build the MSC2UPPAAL parser.
The internal structure

The internal structure of the bMSC, designed in RTDS, is exactly the same as for the SDL model, as was described in section 4.5.2 of chapter 3. The only difference is that we will now also use the position field. This is necessary to determine the exact location of the incoming and outgoing messages of a single lifeline, which will, in turn, allow us to determine the partial order of the traces.

Translating the bMSC to its UPPAAL counterpart: the MSC2UPPAAL parser

The parsing will happen in different steps, as can be seen in the pseudo code below.

```plaintext
MSC2UPPAAL(bMSCfile) {

    //write head of output file
    write_head_output_file();
    //parse the process file
    parse_bmsc(processFileName);
    //create the automaton
    create_automaton();
    //mark the paths of the span construct
    add_span_paths();
    forEach(spanPath)
        //write the template header
        write_template_header();
        //write the state location structures
        write_span_path_locations();
        //write the transition structures
        write_span_path_transitions();
        //write the end of the UPPAAL file
        write_system_end();
}
```

As we can see, this is rather similar to what happened in the SDL2UPPAAL parser. First, we still need to prepare the UPPAAL xml file. After this, we will have to parse the bMSC file, which will be discussed later on. This is followed by the new create_automaton function, which will also be discussed in more detail, later on in this chapter. After this, we have to mark the paths that deal with the span construct in the bMSC model. This will happen with the same breadth first algorithm introduced in section 4.5.2 of chapter 4. After all the parsing is done, we will write every span path to its corresponding output file.

Parsing the bMSC file  Since we will deal with very basic bMSCs, we will only need to parse a few symbols, as can be seen in the pseudo code of appendix G.1. In short, the following things happen. If the parser find a lifeline, it will initialize a list, where all incoming and outgoing messages will be stored. Therefore, when the parser find a message link, it will store the message in both lifelines, as a message is always
sent between two lifelines, together with its orientation, incoming or outgoing, and its position on the lifelines. The latter is important, as discussed before, since we will use that later on to determine the partial order of the bMSC. Parsing a comment symbol or a comment link is the same as we did in the SDL2UPPAAL parser. The only difference is that we now will store which lifeline the comment is attached to, together with its position.

Creating the automaton  After we have all information from the bMSC file, we will create the automaton representation of the bMSC, which will contain all traces of that bMSC, together with the timing information, as described in section 5.2.1. The pseudo code can be seen in appendix G.2.

Logically, the creation of the automaton will work as followed. Let us recap the bMSC of figure 5.2. Every point on every lifeline in which a message is sent or received is represented by a state in the automaton. Furthermore, we can see that a trace is split up into two traces, everywhere there are multiple messages sent. We can also see that a trace ends, there were there is an incoming message node on the lifeline, but no more nodes that represent an outgoing message. A second case where a trace ends is when the next node on the life line is also an incoming message.

Following this logic, we get the following automaton in UPPAAL, which represents all traces of the bMSC, depicted by Figure 5.2. As we can see, I have chosen to end all traces in a special end state. The reason for this is that it will make it much easier to connect the different bMSCs when we would verify a MSC specification.

During the creation of the automaton, the parser will check whether a certain channel has a delay, or whether there is a duration specified between the reception and the sending of a message. If this is the case, this information will be added to the automaton, as described in section 5.2.1.
After this, we will need to mark all paths according to the span constructs in the SDL model, and write each span path to its own specific UPPAAL file, with which we can verify whether or not the time requirement holds.

5.3 Test results

In the course of this chapter, I have discussed how we can verify the timing requirements of a bMSC, using UPPAAL. In this section I will prove that the suggested methods are correct, similar to what I did in section 4.6 of chapter 4. For this, I will first prove that the MSC2UPPAAL parser functions accurately, and I will show that verification of timing requirements, specified in a bMSC, can be verified in UPPAAL.

5.3.1 The test case

We will use the same test case of section 4.6.1 of chapter 4. We will especially concentrate on the verification of the second timing requirement, which stated that, in case of an emergency, the door should close within a time interval of 1 and 2 seconds, after the passkey has been entered.

5.3.2 Verification

To be able to verify this, we will create a specific scenario, as can be seen in Figure H.1.

This depicts a scenario, in which the user successfully passes the iris scan and the passkey security mechanisms, after which the door is opened, followed by the closing of the door. In this model, we have included more timing information, in that now we also will take the channel delays into account. This will allow us to verify the model in even more detail.

The main difference between the bMSC model of Figure H.1 and the SDL model, depicted in Figure F.1, is that in the bMSC model there is no notion of a failure to pass one of the security mechanisms. This is so, because the scenario only concentrates on a specific execution of the system. We will now try to validate the timing requirement, as described above.

The MSC2UPPAAL parser will create the spanAToB.xml file, which will contain the path from the A event to the B event, as described in section 5.2.2 of this chapter.

Verifying spanAToB.xml

This file will contain the path from the A state to the B state, which was extracted from the bMSC model, as mentioned above. The result of this can be seen in Figure H.2.

As we can see, the UPPAAL model has states that represent the sending of a message in the MSC model, besides of course the begin and end state. These states will also represent the possible channel delay, together with the possible duration, as described in the bMSC model. This can, for example, be seen by the start state. This state will have a delay of 910 time units, which consists out of 10 time units, for the channel delay, and 900 time units, which is a result of the time needed to actually open the door.
We can now use this model to verify the timing requirement, as described in the beginning of this section. Important to note here, is that we do not need to specify an environment in UPPAAL, before we can do the verification. This is so, because the order of the messages is clear from the partial ordering of the bMSC.

To verify the timing requirement, we can use the following TCTL formula:

\[ \text{Span1} == 1 \rightarrow (\text{span1} == 1 \land \text{spanClk1} >= 1000 \land \text{spanClk2} <= 2000) \]

Again, 1000 and 2000 time units are the abstract representation of 1 and 2 seconds, respectively, if we consider a system clock with a granularity of 1 millisecond.

This gives us the following result in UPPAAL, as can be seen in Figure 5.4.

\[ \text{P0.span1} == 1 \rightarrow (\text{P0.span1} == 1) \land \text{P0.spanClk1} >= 1000 \land \text{P0.spanClk1} <= 2000 \]

Property is satisfied.

Figure 5.4: The path from event closed to event Closed

We see now that we can verify the timing requirement, something which was not possible with the SDL model. This shows that, for more complex systems and more detailed timing requirements, we will always need MSC models, next to the SDL models.

5.4 Summary

In this chapter, we have discussed how we can verify the time consistency of basic MSCs and high-level MSCs.

First, we have talk about how we can verify bMSCs. For this, two techniques were available:

1. with the distance graph: with this technique, we create a distance graph of the basic MSC, and calculate the all-pairs shortest paths. Should we find a cycle in the graph with a negative cost, we can conclude that the bMSC is time inconsistent

2. With UPPAAL: a technique thought up by myself, in which we will find all traces in the bMSC going from one event to another, for which we can use the UPPAAL tool to do the verification, to check whether the temporal requirements hold for these traces.

After this, we have shortly discussed how we can verify hMSCs. This is done by unfolding the hMSC to its bMSC representation, and using the above techniques to check for time consistency. A problem that arose here, was that it was not clear how we should deal with (infinite) loops inside the hMSC. However, it turned out that we could unfold those loops a finite amount of times, under certain conditions, and that verification will be conclusive for the resulting bMSC.

Then, we have discussed how the temporal information inside a bMSC can be represented in UPPAAL, which turned out to be verify similar to what we had done in chapter 4. The following items were discussed:
- The system clock, and how it will be used to make all concrete time information, inside the MSC model, abstract

- Events, and how they will only be used internally

- Channel delay, duration, event intervals, processor speed intervals and instance delay, and how they can be represented in the UPPAAL model using invariants of states and guards on transitions

- Spans, which represent temporal requirements. We discussed how we can represent these in the UPPAAL model, using span variables that denote the span path, similar to chapter 4, and how they can be verified using TCTL formulae.

After this, we have highlighted the most important points of the MSC2UPPAAL parser, and we have concluded this chapter with a test case, in which we verified the temporal requirement that was not verifiable in the SDL model, as was described in chapter 4.
In this thesis, we have discussed how we can verify the temporal properties of a concurrent, real-time system. The verification technique that is used today is based on manual testing of the system at hand, which takes up more than 50% of the entire development period. However, this technique becomes a burden when we consider the verification of temporal properties for complex systems. This is so, since manual testing only allows development in small iterations, and, since real-time systems are exponentially more complex than their untimed counterparts, manual testing becomes unreliable, in that not all traces of the systems execution might have been examined. For this, there are two possible solutions, being verification via automatic execution-trace finding, using e.g. genetic algorithms, or by means of formal verification. This thesis revolved around the latter.

To be able to perform formal verification of a system, we need a formal, mathematical model of that system, which makes it possible for a machine to comprehend the system, and to reason about its properties, using mathematical logic. This implies that we need to model the system using a formal language. However, the largest part of the software industry uses UML as modelling language, which is informal, and which can, thus, not be used for formal verification. This brings with it the problem that UML has been accepted by most part of the software development industry, so that we are not able to diverge much from it. The reason for this is that it is foreseeable that developers are not likely to adjust to an entire new modelling language, since they are already familiar with UML, and because the adjustment would take a considerable amount of time and, thus, money. Therefore, I have searched for formal languages, for which an UML profile already exists. It turned out, that SDL and MSC are perfect candidates, since a) they are formal languages, and b) there exists a UML profile, which allows a one-to-one mapping of (a subset) of UML to (a subset) of SDL, as described in the Z.109 Recommendations[47]. The benefit of this is that developers now can use UML to create the formal model of a system, on which formal verification is possible, without the need for the developer to understand, or even be aware of, the underlying mathematical model.

The goal was, thus, to present a technique that allows the formal verification of temporal requirements for hard real-time systems, designed in SDL and MSC, early in the design phase. For this, three deficiencies needed to be overcome:
1. SDL and MSC have a very weak model of time

2. It is not possible to express all temporal requirements in the SDL and MSC models

3. There is very little tool support to do the actual verification of temporal requirements of systems that are modelled in the SDL and MSC languages.

The first deficiency, being the weak model of time, was solved, as suggested by ITU-T, by introducing a stricter model of time, as used in timed automata. This was possible, since the underlying structure of SDL and MSC is a finite state machine, which is closely related to timed automata.

The second deficiency was solved in three stages. First, we had to identify the different aspects of the temporal requirements of real-time systems. It turned out that this was already addressed in numerous papers, among which the most important ones were [21], [37] and [39]. Second, I had to create formal semantic rules, so it became possible to express all facets of the temporal requirements of a real-time system. Third, and last, we examined how we could add this information inside the model. The best way to do so, as described in [39], turned out to be by means of annotations. With these annotations, we now could add information inside the model, about, among others:

- duration of events
- channel delays
- important events in the system
- timing requirements concerning these events, using a span construct

After we had established a formal base to express temporal information and temporal requirements, we tackled the third and last deficiency of SDL and MSC, being the lack of verification tools and methods.

The base of the verification method, that I proposed, lies in the fact that both SDL and MSC have a finite state machine as their underlying structure, as mentioned above. This allowed me to create timed automata representations of these models, so that we were able to use the UPPAAL tool, to do the actual verification.

To be able to do this, we first discussed how the various types of temporal information and requirements could be transformed to their UPPAAL counterparts. Afterwards, we have studied the SDL2UPPAAL and the MSC2UPPAAL parsers, which made it possible to automatically translate the SDL and MSC models, respectively, to their UPPAAL counterparts. The parsers will also automatically translate the temporal requirements of the system, which are represented by span constructs in the SDL and MSC models, to TCTL formulae, which will eventually be used by the UPPAAL tool, to do the verification of the found span paths.

I have tested each parser with a small test case. This proved both the correctness of the methods, as described in this thesis, as well as the correctness of the parsers themselves. During the verification of the test case, an important fact arose, since it turned out that not every temporal requirement can be verified using only the SDL model, but that for specific temporal requirements, we will also need MSC models.
We can conclude from the above, that the verification methods, which I have de-
scribed in the course of chapters 4 and 5, are correct, and that they allow us to do an
actual verification of real-time systems, which was the main purpose of this thesis, in
the first place. The only question that remains is whether or not the industry will be
keen to adopt the suggested methods. This depends on a few factors:

- Willingness of the industry to adopt SDL and MSC, next to UML. As described
  in the beginning of this chapter, we need to steer the focus of the developers
  away from the informal UML modelling language, and more in the direction of
  the formal languages SDL and MSC. An important tool for this is the z.109
  Recommendations[47], which provides a one-to-one mapping from UML to
  SDL.

- Willingness to use UPPAAL. As there are numerous verification tools out there,
  and other, promising, methods of verification, such as the IF project[43], it is
  not certain whether the community will like to use UPPAAL. However, since
  UPPAAL already has a huge user base, it is foreseeable that it will be accepted.

- Willingness of ITU-T to add the possibility to specify timing requirements, us-
  ing annotations, as described in this thesis, to their Recommendations. Even
  though papers that promote the specification of timing requirements inside the
  SDL and MSC models, have been around for many years, there is still very lit-
  tle support for this in the official Recommendations. Adding the possibility of
  specifying timing information to the Recommendations might greatly improve
  its popularity, which will, in turn, strengthen its position against other modelling
  languages, such as UML, in the real-time sector.

Still, there are numerous facets, of the verification of real-time systems, that should
be addressed in the future.

First, the SDL2UPPAAL and MSC2UPPAAL parsers are developed as a proof of
concept. This has some consequences:

- No full coverage of error messages. This implies that not all human mistakes,
  that can reside inside the SDL and MSC model, are caught.

- Not all temporal information, as described in chapters 2 and 3, can be parsed
  with the parsers. Only a specific selection is parseable.

This implies that, before we can use the parsers in real-life situations, they still
require some extensions and some testing and correcting.

Second, in this thesis, we have only discussed static verification of temporal re-
quirements. However, some improvements are still possible.

- A proper solution needs to be found for the problem described in section 4.5.2.
  This problem came forth out of the fact that we came to the conclusion that we
  needed to lift the span paths out of the timed automaton. This brought with that
  it was unclear on how we should deal with variables declared in states previous
to the beginning of this path.
Summary, Conclusions and Future Work

- It should be examined to what extent we can combine the specification of the SDL model and the MSC model. It could, for example, be possible to define a channel delay in the SDL model, since this is a structural characteristic of the system, which can then, automatically, be used in the MSC model.

- It can be useful to examine the possibility, whether a method exists that allows us to compare the time specifications in both the SDL and the MSC model, to identify e.g. contradicting specifications.

Next to static verification, it would also be useful to be able to do dynamic verification, such as schedulability analysis. Schedulability analysis of a real-time system can be done either static or dynamic[22]. This will allow a more detailed and more reliable verification, as we now also include such things as non-trivial context switches[1], release jitter, etc.

Schedulability analysis will have to incorporate different scheduling algorithms, such as the rate monotonic scheduling algorithm[10], the deadline monotonic scheduling algorithm[4], and the earliest deadline first scheduling algorithm[6]. It will turn out that the success of the schedulability analysis will strongly depend on the scheduling algorithm used.

Furthermore, methods will need to be found, to actually perform the schedulability analysis on the SDL and the MSC models. A possible tool that could achieve this, is the Times\(^1\) tool, developed by the same creators of UPPAAL. For this, we will need a parser to transform the SDL and MSC models to a model that the Times tool can understand, which is a similar technique to the method described in this thesis.

\(^1\)Times: Tool for Modelling and Implementing of Embedded Systems, www.timestool.com
Bibliography


Summary, Conclusions and Future Work


[19] Frits Vaandrager Gerd Behrmann, Thomas Hune, Distributing timed model checking - how the search order matters.


[34] Wang Yi Kim G. Larsen, Paul Pettersson, Uppaal in a nutshell.


[36] Leslie Lamport, Real time is really simple, (2005).
[37] Stefan Leue, *Specifying real-time requirements for sdl specifications - a temporal logic-based approach.*

[38] Marwa Sousha Lionel C. Briand, Yvan Labiche, *Stress testing real-time systems with genetic algorithms.*


[40] Danien Vincent et. al. Marius Bozga, Susanne Graf, *Sdl for real-time: What is missing?*


Appendix A

Clocks

As was described in section 2.1.1, if we rely only on the now value as a representation of time progression in the system, it will lead to a state space explosion during verification of the time requirements, since now can have any possible value. However, the solution used in other verification tools, letting time only progresses when the system is stable, is too restrictive and not usable for the verification of real-time systems [46].

A better solution is presented in [46]. Here, it is suggested that we use the VHDL\(^1\) time model. In this model, time relies on events. An event is the change of any signal. This does not consume time itself, as time consumption is related to the delay between execution units. The delay itself can either be specified by the user or a library, or might not be specified at all. In this case, VHDL will assume a virtual delay \(\delta\), which will be a delay less than or equal to the real amount of time of the delay. During simulation or verification in VHDL, time will progress in a discrete way, either with \(\delta\) steps for unknown delays or real-time intervals for known delays. It is suggested by [46] that we formally adopt this time mechanism in SDL. To do so, we first need to discuss different clock theories, which will be done next. How SDL can be extended in a formal way to include the timing requirements will be discussed in section 2.4.

A.1 Ideal Clock

An ideal clock \(C_i\) is a clock which is defined by a linear function of time, so that

\[
C_i(t) = t
\]

where \(t \in \mathbb{R}^+\) and has a physical unit, usually seconds. A graphical depiction of clock \(C_i\) can be found in Figure A.1 on page 95.

A.2 Continuous real clock

The ideal clock is a bit too ideal when compared to physical clocks. Therefore we need a refinement of the ideal clock to a continuous real clock. This clock has two extra parameters, being a certain drift over time and the offset value.

---

\(^1\)VHSIC Hardware Description Language, programming language used to program digital circuits
The drift over time, depicted by the function $d(t)$, depicts the difference in clock speed of the continuous real clock in comparison with the ideal clock, usually expressed in parts per minute (ppm). The drift value relies on external circumstances, such as the supply voltage and the temperature of the system. Usually the maximum drift is depicted in the data sheet of the clock. The maximum drift $D$ is now $\max_v |d(t)|$, so that we can define the continuous real clock in function of time as

$$(1 - D)t \leq C_c(t) \leq (1 + D)t$$

The offset value is the second parameter of the continuous real clock. It depicts the time difference between the ideal clock and the continuous real clock when we start reading the latter. Say that $S$ is the time on the ideal clock, either specified by the user or obtained from an other system, e.g. a GPS system, one can now formally say that

$$t \rightarrow \tau - S$$

where $\tau$ is the original value of the ideal clock. We will use this refined value of $t$ from now on in the rest of the formulae.

There is one problem that needs to be considered using the continuous real clock, and that is that the offset $o(t) = C_c(t) - C_i(t)$ will eventually exceed any limit, depending on its drift and the time spent. This should be avoided. One way of doing so, is to ensure that the drift is not larger than a certain value in a certain time period $T_s$. This gives the following correlation [46]:

$$D < \frac{o}{\tau}$$

which might, in turn, bring with it new problems such as the need for very accurate and very expensive clocks. A graphical depiction of clock $C_c(t)$ can be found in Figure A.1 on page 95.

### A.3 Discrete real clock

Processes in digital real-time systems only have limited capabilities to read a clock. Therefore one can use digital clocks, clocks that oscillate to increase a certain counter. The result we get from reading such a clock will depend on two things, being its granularity $G_R$, which depicts the minimal difference between subsequent clock reads different from zero, and the range of clock values $R_R$. The discrete real clock is formally depicted by the formula

$$C_R(t) = \left\lfloor \frac{C_c(t)}{G_R} \right\rfloor \times G_R \mod R_R$$

where $C_R(t)$ and $G_R \in \mathbb{R}^+, R_R \in \mathbb{N}$, and the time unit is seconds.

The discrete real clock is depicted in Figure A.1, where the granularity is 2 seconds. This implies that every multiple of 2 seconds of the reference clock $C_c$, the discrete real clock $C_R$ increases with 2 seconds. The discrete real clock starts at zero seconds.
A.4 Derived counters

Now that we have a discrete real clock, it can be useful to be able to derive different counters with different granularities. For this we require a certain delay between changes of the reference clock and the incrementing of the derived counter. This is depicted as $\delta$. Furthermore, we need to define the number of clock ticks that need to pass in the reference clock for one clock tick of the derived counter. This is depicted by $N_a$. Furthermore, the granularity and the range of the counter is depicted by $G_a$ and $R_a$ respectively. This gives us the following formula for a derived counter $C_a(t)$

$$C_a(t) = \lfloor \frac{C_c(t-\delta)}{G_R} \rfloor * G_a * \text{mod} R_a$$

where $\delta \in \mathbb{R}^+$, $G_a \in \mathbb{N}$ and $R_a \in \mathbb{N}$. In the formula above, the overflow of the derived counter is independent from the overflow of the discrete real clock. An example of a derived counter can be seen in Figure A.1, where the counter will require two clock ticks of $C_R$ to increase once, and where the counter has a granularity of one second.

![Figure A.1: The Software Development Cycle [3]](image-url)
Appendix B

Logical Verification Formulae

B.1 LTL

The syntax of LTL formulae, can be seen in table B.1.

<table>
<thead>
<tr>
<th>Textual</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NΦ</td>
<td>[]Φ</td>
<td>Φ has to hold in the next state</td>
</tr>
<tr>
<td>GΦ</td>
<td>[]Φ</td>
<td>Φ has to hold globally, for all paths</td>
</tr>
<tr>
<td>FΦ</td>
<td>&lt;&gt;Φ</td>
<td>Φ has to hold eventually, for all paths</td>
</tr>
<tr>
<td>ΨUΦ</td>
<td>ΦUΦ</td>
<td>Until Φ holds, Ψ has to hold</td>
</tr>
<tr>
<td>ΨRΦ</td>
<td>ΦRΦ</td>
<td>Φ is true, until Ψ becomes true, if ever</td>
</tr>
</tbody>
</table>

B.2 TCTL

The syntax of TCTL formulae, as specified in [17], can be seen in table B.2.

<table>
<thead>
<tr>
<th>Textual</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Φ :=</td>
<td>x ~ c</td>
<td>Clock comparison</td>
</tr>
<tr>
<td></td>
<td>x - y ~ c</td>
<td>Clock comparison</td>
</tr>
<tr>
<td></td>
<td>Φ1 ∨ Φ2</td>
<td>Either Φ1 is true, or Φ2</td>
</tr>
<tr>
<td></td>
<td>not Φ</td>
<td>The inverse of Φ</td>
</tr>
<tr>
<td></td>
<td>∃&lt;&gt;IΦ</td>
<td>There exists a path for which eventually Φ is true, for which the time constraint I holds</td>
</tr>
<tr>
<td></td>
<td>∀&lt;&gt;IΦ</td>
<td>For all paths, eventually Φ will be true, and the time constraint I holds</td>
</tr>
<tr>
<td></td>
<td>∃][IΦ</td>
<td>There exists a path for which Φ is always true, and the time constraint I holds</td>
</tr>
<tr>
<td></td>
<td>∀][IΦ</td>
<td>For all paths, Φ will always be true, and the time constraint I holds</td>
</tr>
</tbody>
</table>
For which $\sim$ is an operator equal to either $<$, $\leq$, $=$, $\geq$ or $>\? x$ and $y$ are clocks and $c$ is a natural number.
Appendix C

UPPAAL: a closer look

C.1 Graphical representation of timed automata in UPPAAL

As mentioned in section 4.3.2, UPPAAL uses timed automata to represent the system. Therefore, all elements of the definition of a timed automaton, as described in section 4.2.1, need a graphical, and sometimes textual, counterpart in UPPAAL. These will be discussed next.

C.1.1 States

An example of how two states, called start and end, with a simple transition between them, would be represented in UPPAAL is depicted in figure C.1.

As we can see, we have one initial state, called start. This is represented by the circle inside the state symbol.

Urgency of states

States also can have a certain type of urgency. UPPAAL defines three different types of urgencies: lazy, urgent and committed[20][5]:

lazy lazy locations are locations in which time might pass an arbitrary amount of time, even if there are transitions that are enabled. The system might thus stay in such a state for a theoretical infinite amount of time, before moving on the next state. A lazy location is depicted by the normal state symbol, as can be seen in figure C.2(a).
urgent locations are locations where no time might pass before a transition is taken. An urgent location is depicted by a state symbol that has the letter U transcribed in it, as can be seen in figure C.2(b).

committed committed locations are even more restrictive than urgent locations, in that, besides the fact that no time might pass when the system resides in a committed state, the next transition taken must be an outgoing transition out of a committed location. A committed location is depicted by the state symbol with the letter C transcribed in it, as can be seen in figure C.2(c).

Invariants
States of a timed automaton can, as mentioned in section 4.2.1, have invariants. An invariant is a condition which must be true for the system to be allowed to reside in that state. In contrast with the definition of timed automata, UPPAAL can have invariants not only concerning clocks, but also concerning integer values, booleans, arrays, etc.[20]. A location invariant is depicted in figure C.3, where $clk$ is a clock and $i$ is an integer value. Invariants can be combined using either an and or or operator.

C.1.2 Transitions
It became clear from the definition of a timed automaton, and the examples given above, that states are connected by transitions. UPPAAL knows four different kinds of annotations to edges, of which we will discuss three.

Guards A guard is a predicate involving either clocks, integer or Boolean values, in contrast to the definition of timed automata in section 4.2.1, where only clock guards are allowed. If the guard predicate is true, the transition is enabled, and vice versa. Only enabled transitions can be taken. An example of a guarded
transition can be seen in figure C.4(a). Here we see that the transition is only allowed to be taken if and only if the clock $clk$ is smaller than 5 and the integer value $i$ is bigger than 2. Guards are always depicted in a green text color.

**Synchronization** A channel can also have a synchronization element. This represents either the reception of a message, denoted with a question mark, or the sending of a message, denoted by an exclamation mark. Respecting the guards, it is always possible to send a message, and thus take the transition, and it is only possible to receive a message after it has been sent. The latter makes sure that the model in UPPAAL is deterministic, which means that a message can not be received before it is sent. An example of the reception of a message over the communication line $channel$ can be seen in figure C.4(b). Synchronisations are always depicted in a blue color.

**Updates** According to the definition of timed automata, a transition can also reset clock values. This is also possible in UPPAAL, but much more can be updated. We can not only reset a clock, but we can also assign a certain value to it. Furthermore, we can also assign values to integers, booleans, etc, each time we take a transition. An example of a transition with an update can be seen in figure C.4(c). Here we can see that the clock $clk$ is reset to zero, and the integer $i$ is set to two. Updates are always depicted in a purple color.

![Figure C.4: Transitions in UPPAAL](image)

**C.2 Verification methods in UPPAAL**

We will now shortly discuss the different options that are available in the UPPAAL tool, when we want to verify our model. A profound understanding of these options is a necessity, since they might greatly influence both the time and space complexity of the actual verification, and it is even so that a model which can not be verified in UPPAAL with one set of options, might be verifiable with another set[5].

In the next two sections, we will first discuss the data structure used to represent the symbolic state space of the model. Second, we will shortly examine the functioning of the two main verification methods, breadth first search and depth first search, together with their complexities in time and space.
C.2 Verification methods in UPPAAL

C.2.1 Internal data structure of the simulation graph

As already mentioned in sections 4.2.2 and 4.3.2, UPPAAL will use a symbolic representation of the state space, which represent the current state(s) of the system at hand. This means that the current state of the system will be allocated in memory by the current values of its clocks and variables[18]. There are two possible method to describe the symbolic state space in UPPAAL: difference bounded matrices and compact data structure.

Difference bounded matrices

With the difference bounded matrices[18], in short DBM, the symbolic state space is represented by a directed, weighted graph for which clocks of the system are represented by vertices, and each edge represents a clock constrin of the system. The weight of these edges is the result of the constraint between two clocks. An example of a DBM can be seen in figures C.5(a) and C.5(b), respectively. Here we have simple, abstract UPPAAL model, for which there is a transition which is guarded by a clock constraint, such that the transition is only available when the difference between two cocks $clkX$ and $clkY$ is greater than 5 time units.

![Diagram](a) the UPPAAL model  
(b) the DBM

Figure C.5: The UPPAAL model and its DBM

Compact data structure

Symbolic states, as represented in the DBM, can have many representation in memory. The first versions of UPPAAL represented each entity in a state by a single machine word[18]. This turned out to be very cost-ineffective concerning memory, which is a problem when we want to verify huge systems with lots of different states. To solve this, the state space can also be represented by a compact data structure. Here, a state is represented by a single number, of which each entity in the state can be resolved using a multiplication and addition scheme [18]. This technique significantly decreases the memory consumption to represent the state space. However, the drawback is that we now have an additional, possibly expensive, cost of decoding the state number, which will add to the total verification time[18].

C.2.2 Verification algorithms

As we already know from section 4.2.2, it is possible to create a simulation graph from the timed automaton model of a real-time system. This induces that the problem of verification can be solved by solving the reachability problem. The goal of the reachability problem will be to check whether, for a certain automaton, a certain state
can be reached. With this method, we will be able to solve safety problems. Even though the reachability problem is decidable, it is PSPACE-complete\(^1\).

UPPAAL provides us with three path finding algorithms, to solve the reachability problem. These are breadth first search, depth first search and random depth first search. It is important to understand the different characteristics of these algorithms, as the search order will determine the eventual state space structure\([19]\), which will in turn influence the time and memory factors.

The characteristics of these algorithms will shortly be described first. Next, the working of these algorithms will be discussed, followed by possible pitfalls, advantages and disadvantages of the three methods \([9]\).

**Breadth first search: characteristics**

Breadth first search is a path finding, or graph search, algorithm. Its characteristics are:

- **Space complexity:** \(O(|V| + |E|)\), where \(|V|\) is the number of nodes, and \(|E|\) is the number of edges of the graph. This implies that an enormous amount of memory is required when applying the breadth first search technique on large systems.

- **Time complexity:** \(O(|V| + |E|)\). This is so, because at worst case, all nodes and all paths need to be visited to find the node we are searching for, as will become clear in section C.2.2.

- **Completeness:** Breadth first search is considered complete, since it will always find a solution, if one exists. However, if the graph is infinite, due to cycles, and a solution does not exist, the search might never end.

**Breadth first search: how it works**

I will describe the working of breadth first search using a small example.

Suppose we have the following graph on which we have to perform breadth first search, presented in figure C.6.

The breadth first search algorithm, in short BFS, as well as the depth first search algorithm, will always start at the initial node. Suppose that we wish to find the path to node \(F\). BFS will therefore put the first state it encounters, here state \(A\), in its queue. Since there is nothing else to do anymore, BFS will pop the first element of its queue, being state \(A\), find all its successors, and push them in the queue. This gives us a progression as can be seen in figure C.7.

In the queue, we now have nodes \(B\) and \(C\).

Since there is nothing else to do, and the encountered successors are not the end point, BFS will pop the first element out of the queue, and find all its successors. This gives us a state space progression as can be seen in figure C.8, and a queue that now contains the elements \(C\), \(D\) and \(E\).

Since none of the successors encountered are not the desired end node, BFS will continue, and pop the first element from its queue, being location \(C\). This gives us the following state space progression, as can be seen in figure C.9. BFS now has successfully encountered node \(F\), which was the desired node.

---

\(^1\)PSPACE-complete: using a polynomial amount of memory and an unlimited amount of time
Depth first search: characteristics

Depth first search is also a path finding algorithm, which has as characteristics:

**Space complexity:** $O(h)$, where $h$ is the length of the longest simple\(^2\) path of the graph. This is so since we only have to consider the current path we are on.

\(^2\)a simple path is a path in which no vertex is visited more than one time
Time complexity: \( O(|V| + |E|) \), since in worst case, we have to visit all vertices and edges, before finding the desired node.

Completeness: Depth first search is also considered complete, as long as no cycles are involved. This will be discussed in more detail in section C.2.2.

Depth first search: how it works

Now let us examine how depth first search, in short DFS, works. Let us again consider the graph of figure C.6. DFS will start with the first node encountered, which is node A. This node will now be pushed on a stack, instead of a queue. Since the begin node is not the node we are looking for, which is node F, DFS will pop the top element from the stack, and explore its successors. This will give us the following progression of the state space, as can be seen in figure C.10. The stack now contains elements B and C.
the next element from the stack, and examining its successors. The progression in the state space can be seen in figure C.11. The stack now contains elements $D$, $E$ and $C$.

![Figure C.11: DFS: search space after two steps](image)

DFS now pops the next element from the stack, which is element $D$. Since this element was not the searched node, and since it has no successors, it is discarded.

The next element is now popped from the stack, and its successors are pushed onto the stack. The progression through the state space can be seen in figure C.12. The stack now contains elements $G$, $H$ and $C$.

![Figure C.12: DFS: search space after four steps](image)

Elements $G$ and $H$ are popped from the stack, and since they are not the searched node, and since they have no successors, they are discarded. The only element left on the stack is the element $C$. This element will be popped off the stack, its successor $F$ will be examined, and DFS determines successfully that there is a path from the begin node to the searched node.

**Random depth first search**

Random depth first works exactly as DFS, with as only difference the fact that the successor to examine is determined at random. This has as a consequence that there
now is a chance that, in stead of first expanding the search tree of figure C.6 going from node $A$ to $B$, and so on, to first go from node $A$ to node $C$, which will then go directly to node $F$, and thus successfully concluding the reachability algorithm.

**Pitfalls**

We will now shortly discuss some pitfalls, together with possible solutions, that can occur using either BFS or DFS.

**Search order** It is clear now that it strongly depends on which search method we use, how the state space of our model will be traversed. E.g. in our example, we can see that breadth first search finds the desired node in less rounds than depth first search. However, if we would have searched for node $H$ in stead of node $F$, it would have turned out that depth first search would be quicker than breadth first search. For this, we can draw two conclusions. First, it is important how deep the searched node is located in the search tree. It is clear that DFS will spend less time on average to find nodes that are located at a deep depth in the tree. Second, it is also important where the searched node is located in the tree, in an horizontal manner. This is so, since nodes that are located at the right part of the tree will be located quicker using BFS, as came clear from the example in the previous section. Therefore it might be important to use a certain search algorithm in certain cases, and an other search algorithm in other cases.

**unbalanced trees** As came clear from the previous item, it is important where the search nodes resides in the tree in an horizontal fashion. When we have a strongly unbalanced search tree, e.g. when node $G$ of figure C.6 has many successors, depth first search will spend an enormous amount of time traversing this sub tree with as parent node $G$, when searching for e.g. node $F$. This would not be the case for BFS. Therefore, again, it is important to find a balance both speed of the verification and memory consumption. An other solution offered by UPPAAL is off course that we can make use of random depth first search, which might solve the problems encountered with unbalanced trees.

**cyclic trees** Cyclic trees are also a problem when choosing DFS. For this, consider the search tree of figure C.13.

Here we can see we have a cycle going from node $D$ back to node $A$. If we use the DFS on this tree, we will eventually come to the following situation, as can be seen in figure C.14. The stack will contain elements $D, E, C$.

As successor of node $D$, we have node $A$. Therefore, node $A$ will be pushed on the stack, and the algorithm will continue. Eventually, the following elements will be on the stack: $D, E, C, D, E, C$. Again, the successor of node $D$ will be node $A$, and we can now understand that DFS will be taking the loop indefinitely, and nodes $C$ and $E$ will never be explored.

Different solutions are possible, such as remembering which nodes already were passed, which might lead to missing possible paths, or setting a limit on the depth of the tree, which exhibits the same problem.
Since BFS will traverse the tree in an horizontal fashion, cycles in the tree will not be a problem, as long as the node we are searching for actually exists in the tree[18].
D.1 The function `parse_process()`

```c
parse_process(processFileName){
    Case(structureType)
        == sdlStart
            parse_start_symbol();
            break;
        == sdlStop
            Parse_stop_symbol();
            break;
        == sdlState
            Parse_state_symbol();
            break;
        == sdlInputSig
            Parse_input_symbol();
            break;
        == sdlTask
            parse_task_symbol();
            break;
        == sdlText
            parse_text_symbol();
            break;
        == sdlDecision
            parse_guard_symbol();
            break;
        //parse transitions
        == sbvoid
            parse_normal_link();
            break;
        == dec
            parse_decision_link();
            break;
        == dsvoid
            Parse_comment_link();
            break;
    end case;
}
```
D.1 The function parse_process()  

D.1.1 Function parse_state_symbol()

```cpp
parse_state_symbol()
{
    int x;
    int y;
    string symbId;
    string stateText;
    //Parse the symbol structure, and obtain the necessary data
    x = structure.x;
    y = structure.y;
    stateText = structure.text;
    symbId = structure.symbId;
    //A state can have multiple representations in
    //the rtds model, but should only have one
    //representation in the UPAAL model
    if (stateText not yet known)
        uniqueLocations[symbId] = locationInformation;
        linkLocations[symbId] = symbId;
    else
        //This state text has already been encountered,
        //and we have to preserve uniqueness.
        //We will use the occurrence of a state with
        //the lowest id, since this is the first use of
        //the state in the rtds model.
        //We do this, since, in principle, the location
        //of the first occurrence of a state would also
        //be the location of that state, should the model
        //be designed in uppaal.
        if(stateId < uniqueId)
            //Update all link locations to let them point to
            //the new unique id
            updateLinkLocations(uniqueId, symbId);
            uniqueLocations.erase(uniqueId);
            uniqueLocations[symbId] = locationInformation;
            linkLocations[symbId] = symbId;
```
D.2 Parse the decision transitions

D.2.1 Function parse_decision_transitions()

```
parse_decision_transitions()
{

tops = determineTops();
    findPathForAllTops();
}
```
D.2.2 Function `determineTops()`

```c++
determineTops(){

    // topsHash will contain the number of connected
decision transitions:
    // 1 for a leaf, 2 for a top, 3 for a intermediate
decision symbol
    Hashmap topsHash;
    // topsVector will contain all possible tops
    Vector<string> topsVector;
    // tops will contain all tops of all decision trees
    Vector<string> tops;
    forAll(decisionTransitions){
        if decisionTransitions.begin not in topsHash
            // we might have encountered a top
            topsHash[decisionTransitions.begin] = 1;
            topsVector.add(decisionTransitions.begin);
        } Else{
            // already encountered, increase count
            // and add to linkHash
            topsHash[decisionTransitions.begin]++;
        }
        // we need to do this not only for the beginning
        // of the decision transition, but also for the end
        if decisionTransitions.end not in topsHash
            // we might have encountered a top
            topsHash[decisionTransitions.end] = 1;
            topsVector.add(decisionTransitions.end);
        } Else{
            // already encountered, increase count
            // and add to linkHash
            topsHash[decisionTransitions.end]++;
        }
    }
    // now we need to retrieve all tops, for which topsHash
    // is equal to two
    forAll(possible top in topsVector){
        If(topsHash[possible top] == 2)
            Tops.add(possible top);
    }
```
D.2.3 Function findPathForAllTops()

```c
findPathForAllTops(tops){

forEach(top){
    forEach(decisionTransition from top){

        If(isState)
            //we already found a leave, and thus we have
            //a complete transition to store
            if(transition.text == true)
                guard = transition.text;
            Else
                guard = not (transition.text);
            storeTransitionInformation();

        If(isTask)
            //we already found a leave, and thus we have
            //a complete transition to store
            if(transition.text == true)
                guard = transition.text;
            Else
                guard = not (transition.text);
            storeTransitionInformation();

        If(isDecision)
            //we found an intermediate decision symbol, and thus have
            //to repeat our path finding recursively
            if(transition.text == true)
                guard = transition.text;
            Else
                guard = not (transition.text);
            find_path_recursive(id, id, top, guard);

    }
}
```
find_path_recursive(id, father, top, guard) {

    // examine each transition, but not the one going back 
    // to the transition you came from 
    forEach(decisionTransition from id, not leading to the father) {

        If(isState) 
            // we already found a leave, and thus we have 
            // a complete transition to store 
            if (transition.text == true) 
                guard.append("and" + transition.text); 
            Else 
                guard.append("and " + not (transition.text)); 
                storeTransitionInformation(top, this.symbId, guard); 

        If(isTask) 
            // we already found a leave, and thus we have 
            // a complete transition to store 
            if (transition.text == true) 
                guard.append("and" + transition.text); 
            Else 
                guard.append("and " + not (transition.text)); 
                storeTransitionInformation(top, this.symbId, guard); 

        If(isDecision) 
            // we found an intermediate decision symbol, and thus have 
            // to repeat our path finding recursively 
            if (transition.text == true) 
                guard.append("and" + transition.text); 
            Else 
                guard.append("and " + not (transition.text)); 
                find_path_recursive(this.symbId, id, top, guard); 

    }
}
D.3 Function `parse_normal_transitions()`

```c
parse_normal_transitions() {

    forAll(normal transitions with beginId and endId){
        if(isStart(beginId) and isState(endId))
            initialState = endId;
        if(isState(beginId) and isInput(endId))
            //problem: the input symbol might already be part of some transition
            //always work with the unique id of a state
            If(not(partOfTransition(endId))
                newTransition(getUniqueId(beginId), endId, sync);
            Else
                //Expand the already existing transition,
                //so it also contains the state symbol
                existingTransion.begin = getUniqueId(beginId);
                If(isInput(beginId) and isDecision(endId))

                //There already exists one or multiple transitions
                //from the top of the decision element to its leaves, update these transitions
                forEach(transition from endId)
                    existingTransion.begin = beginId;

    }
}
```
Appendix E

Extending the model parser to support timing requirements

E.1 Extending function `parse_process()`

```cpp
parse_process(processFileName){

Case(structureType)
  ...
  == sdlComment
      parse_comment_symbol();
      break;
  == dbvoid
      parse_comment_link();
      break;
  ...
```
E.2 Function `parse_comment()`

```plaintext
parse_comment(){

    foreach line in the comment symbol
    {
        if surrounded by ${ and }
            if (event)
                store symbId, eventName in events vector
            if (urgency)
                store symbId, urgency in urgencies vector
            if (duration)
                makeTimeAbstract(duration, round up);
                store symbId, abstract duration in durations vector
            if (span)
                make time abstract(minSpanBound, round down)
                make time abstract(maxSpanBound, round up)
                store event1, event2, abstract minSpanBound,
                abstract maxSpanBound in spans vector
        else
            ignore comment line
    }
}
```

E.3 Parsing timing information and timing requirements

E.3.1 Extending function `SDL2UPPAAL()`

```plaintext
SDL2UPPAAL(projectFileName){

    ...
    foreach (processFile)
    {
        add_urgencies();
        Add_durations();
        Add_span_paths();
    }
}
```
E.3.2 Function add_urgencies()

```cpp
add_urgencies()
{
    forEach(urgency in the urgencies vector)
    {
        location = getUniqueLocation(urgency.symbId);
        if (urgency.eager)
            location.urgent = true;
        else
            location.urgent = false;
    }
}
```

E.3.3 Function add_durations()

```cpp
add_durations()
{
    forEach(duration in the durations vector)
    {
        location = getUniqueLocation(duration.symbId);
        Location.invariant = durClk <= + duration.duration;
        forEach(transition which ends in location)
        {
            transition.update = durClk = 0;
        }
    }
}
```

E.3.4 Function add_span_paths()

```cpp
add_span_paths()
{
    int counter = 1;
    forEach(span in the spans vector)
    {
        beginLocation = getLocation(span.beginEventId);
        endLocation = getLocation(span.endEventId);
        add_path(beginLocation, endLocation, counter);
        counter++;
    }
}
```
Function `add_path()`

```plaintext
add_path(beginId, endId, spanNumber){

//will contain a queue of all traces that still
//need to be examined
Queue traces;
//will contain the paths that go from the begin
//location to the end location
Vector successful_traces;
//start by pushing the begin id on the trace
Traces.push(beginId);
While(there are traces to be examined){
    //get the next trace to be examined,
    //following the breadth first algorithm
    trace = traces.pop();
    forEach(transition = available transitions of the trace){
        if(possible_transition(transition.guard, currentTrace.varList)
            if (transition.end == endId)
                //path found, put it to the successful_traces
                successful_traces.add(current trace);
            Else
                if(not visited transition.end already)
                    //add the location on the other side of
                    //the transition to the trace,
                    //since it will need further examination
                    currentTrace.add(transition.end);
                    currentTrace.variableList = addUpdates();
                    traces.push(currentTrace);
                    //restore the current trace to its original,
                    //since we might need to examine more paths
                    currentTrace.remove(transition.end);
            }
        }
    Mark_successful_traces(successful_traces, spanNumber);
}
```

Mark_successful_traces(successful_traces, spanNumber);
E.4 Function `add_updates()`

```java
add_updates(update, knownVariables){
    forEach(variable updated){
        if((variable updates with var not in knownVars) and var in knownVars) {
            //delete the variable out of the knownVariable list
            knownVariables.delete(variable);
        } else {
            if(variable in knownVariables)
            >knownVariable.value = variable.value;
            Else
            >knownVariable.add(variable.name, variable.value);
        }
    }
}
```
Appendix F

SDL2UPPAAL: The Test Case
Figure F.1: The SDL model of the secure automatic door
figures and tables:

$ ./SDL2UPPAAL.pddl
reading system file
parsing process pAutomaticDoor.rdd
changing indexes
Parsing decision transitions
Parsing normal transitions
checking event uniqueness
Adding span paths
Adding urgencies
Adding duration
Adding duration transitions

--- Writing span from A to B
writing head output file
writing template header
writing locations
writing transitions
writing queries

--- Writing span from A to C
writing head output file
writing template header
writing locations
writing transitions
writing queries

Figure F.2: The console after we ran the SDL2UPPAAL parser
beginLocation

span1 = 1, spanClk1 = 0, durClk = 0

waitingForPassword

password?
durClk = 0, span1 = 1, span2 = 2

checkForCorrectness
durClk <= 30

correctnessResult?
passKeyCorrect == 1 and irisScanCorrect == 1 and durClk >= 30
   durClk = 0, span1 = 1, span2 = 2

openingDoor
durClk <= 900

doorOpen?
durClk >= 900
   span1 = 1, span2 = 2

open

span1 = 0, spanClk1 = 0, durClk = 0

endLocation

Figure F.3: The path going from event A to event B
Figure F.4: The path going from event A to event C
Figure F.5: Example of a path for which the TCTL formula is invalid
Appendix G

MSC2UPPAAL

G.1 The function parse_bmsc()

```
parse_bmsc(bmscfile){

Case(structureType)
  == mscLifeline
    parse_lifeline();
    break;
  == mscComment
    Parse_comment();
    break;
  == msg
    Parse_message_link();
    break;
  == dsvoid
    Parse_comment_link();
    break;
end case;
}
```
G.2 Creating the automaton

G.2.1 the function create_automaton()

```plaintext
create_automaton()
{
    // create an automaton of the bMSC
    // determine the lifeline that sends the first message
    getFirstLifeline();
    // create end location, where all traces will end
    create_end_location();
    // create the first state, which represents
    // the sending of a message
    // include the possible channel delay
    Create_sending_state(messageName, channelDelay)
    // create the receiving state
    Create_receiving_state
    // create a transition between these states
    // include the possible channel delay
    Create_transition(sending, receiving, delay);
    create_automaton_recursive(receivingLifeline,
        position, receivingNode);
}
```
G.2.2 the function `create_automaton_recursive()`

```cpp
create_automaton_recursive(string lifeline, int position, lastNode) {

    // get the next node on the life line
    // using the position variable
    Get_next_node(lifeline, position);

    // if there are no more nodes
    // or the next node is an incoming node
    If (noNextNode or nextNode == incoming node)
    Connect_to_endnode(lastNode)
    for Each (next outgoing node)
        // for every outgoing node, the trace splits,
        // except for the first outgoing node
        // add duration information to lastNode
        add_duration(lastNode);

    // create the sending state
    Create_sending_state(messageName, channelDelay);

    // create the receiving state
    Create_receiving_state();

    // create the transition between the last node
    // and the sending node
    // add the possible duration
    Create_transition(lastNode, sendingNode, duration)

    // create the transition between the sending and
    // the receiving node
    // add possible channel delay
    Create_transition(sending, receiving, delay);

    // repeat recursive formula
    create_automaton_recursive(receivingLifeline, position, receivingNode);
}
```
Appendix H

MSC2UPPAAL: The test case
Figure H.1: A scenario: successful opening and closing of the door
Figure H.2: The path from event $A$ to event $B$