Predictability in Real-Time Software Design

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 7 september 2005 om 16.00 uur

door

Jinfeng Huang

deboren te Yangquan, China
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. R.H.J.M. Otten
en
prof.dr. H. Corporaal

Copromotor:
dr.ir. J.P.M. Voeten
Summary

The large gap existing between system requirements and system realizations has been a pertinacious problem in the design of complex systems. This holds in particular for real-time software systems with strict timing constraints and safety-critical requirements. Designers have to resort to a stepwise and piecewise design process, where design decisions are made on the basis of partial information of the system (e.g. in the form of an abstraction or a component). To ensure the effectiveness of this design process, predictability is considered to be an essential feature of a design methodology. Predictability implies that design decisions made on the basis of a partial system are still valid in a more complete system.

In this thesis, we investigate from a semantic point of view how to support predictability in design approaches for real-time software. Based on this investigation, we identify predictability problems for two kinds of real-time system design approaches: platform-independent and platform-dependent approaches. Due to the assumption of instantaneous actions, platform-independent approaches generally provide better predictability support for system modelling than platform-dependent approaches do. However, due to the non-zero execution time in realizations and the uncontrollable physical time, timing differences are always observed between a model and its realization. Without careful treatments of these timing differences, the realization may exhibit unexpected behavior w.r.t. the model. As a consequence, the properties verified in the model may not hold in the realization.

To address this property inconsistency problem, we formalize the timing behaviors of real-time systems (e.g. models and realizations) as timed systems. Two proximity functions are defined to measure the absolute and relative timing differences respectively between these timed systems. Furthermore, we formalize real-time properties as temporal logic formulas. Within this mathematical framework, we prove real-time property relations between timed systems on the basis of their absolute and relative timing differences. Consequently, the influence of absolute and relative timing differences on real-time properties of timed systems can be predicted quantitatively.

To apply the above results to real-time system design, we build a sequence of (absolute and relative) timing differences to bridge the timing difference between a model and its realization. We then propose two parameterized hypotheses to bound the absolute and relative timing differences between the model and the realization. The satisfaction of both hypotheses guarantees that properties of the model can be preserved into the realization up to a small deviation. A synthesis approach for real-time software is proposed that shows that both hypotheses can indeed be complied
with during software synthesis. By estimating and adjusting the parameters of both hypotheses, the correctness of the model can be preserved into the realization. As a case study, a rail-road crossing system is designed, which demonstrates the effectiveness of the proposed approach.
Samenvatting

De grote kloof tussen systeemeisen en systeemrealisaties vormt een hardnekkig probleem bij het ontwerpen van complexe systemen. Dit geldt in het bijzonder voor softwaresystemen met strenge tijdseisen en kritische vereisten qua veiligheid. Ontwerpers moeten hun toevlucht nemen tot een stapsgewijze en componentsgewijze ontwerpprocessen, waarbij ontwerpbeslissingen op basis van partiële informatie (bijvoorbeeld in de vorm van een abstractie of een component) van het systeem genomen worden. Een cruciaal onderdeel benodigd voor de doeltreffendheid van zo’n ontwerpproces is voorspelbaarheid. Voorspelbaarheid impliciet dat ontwerpbeslissingen, die op basis van eigenschappen van een partieel systeem gemaakt worden, geldig blijven naarmate dat systeem in meer detail wordt uitgewerkt.

In dit proefschrift onderzoeken we hoe, vanuit een semantisch gezichtspunt, de voorspelbaarheid in ontwerpprocessen voor tijdgebonden softwaresystemen ondersteund kan worden. Gebaseerd op dit onderzoek, identificeren we voorspelbaarheidsproblemen voor zowel platform-afhankelijke als voor platform-onafhankelijke ontwerpbenaderingen. Voor het modelleren van systemen is een platform-onafhankelijke benadering te prefereren boven een platform-afhankelijke benadering. Dit is het gevolg van de aannemen dat acties instantanen zijn in een platform-onafhankelijke benadering. Echter, in de werkelijkheid nemen acties in een systeemrealisatie immer tijd in beslag, waarbij bovendien het precieze fysieke tijdstip slechts met beperkte nauwkeurigheid te sturen is. Hierdoor zal een model en diens realisatie dus een verschil qua tijdgedrag vertonen. Wanneer niet uiterst zorgvuldig met deze verschillen wordt omgegaan, kan een systeemrealisatie onverwacht gedrag gaan vertonen welke niet door het model voorspeld kan worden. Derhalve kunnen eigenschappen die valide zijn in het model, niet standhouden in de realisatie.

Om het inconsistentieprobleem tussen systeemmodellen en systeemrealisaties aan te pakken, onderzoeken wij op een formele manier het behoud van eigenschappen tussen tijdgebonden systemen. Wij formaliseren de tijdgedragingen van (zowel modellen als realisaties van) systemen als getimedede systemen en wij definieren verschillende maten voor absolute en relatieve tijdsverschillen tussen deze systemen. Bovendien formaliseren we reële-tijdseigenschappen als temporele logische formules. Binnen dit mathematische kader, bewijzen wij dat reële-tijdseigenschappen behouden kunnen worden tot op een kleine afwijking na, welke duidt op de (absolute of relatieve) afstand tussen twee getimedede systemen.

Om de resultaten van het behoud van eigenschappen toe te passen op systeemont-
werpen, wordt het tijdsverschil tussen een model en een realisatie uitgedrukt in termen van een opeenvolging van absolute en relatieve tijdsverschillen. We introduceren twee geparametriseerde hypotheses die de tijdsverschillen tussen een model en de realisatie karakteriseren. Verder introduceren we een synthesebenadering voor softwaresystemen in reëel tijd. De benadering toont aan dat de hypotheses in syntheseereedschappen kunnen worden geïmplementeerd. Door het schatten en het aanpassen van de parameters in beide hypotheses, kunnen ontwerpers de correctheid van het model behouden in de realisatie. Als casus wordt een spoorwegovergangsysteem ontworpen, welke de doeltreffendheid van de voorgestelde benadering aantoont.
Acknowledgement

When Prof. Mario Stevens and Prof. Yibin Hou offered me a research position at the Eindhoven University of Technology five years ago, I saw this as a big change in my life and it took me some time to make the decision. Now you are holding the result of my decision, which would never have been accomplished without the help, cooperation and encouragement of many individuals. I would like to thank them all here.

First of all, I would like to express my sincere appreciation to my former promoter Prof. Mario Stevens, who sadly is no more with us. I recognized Prof. Mario Stevens as a respectful supervisor and a warmhearted person. He was always willing to help me through when I had difficulties in work and life. I will miss our joyful conversations on things from scientific research to life experience.

I would also like to extend my appreciation to Prof. Ralph Otten and Prof. Henk Corporaal for their willingness to succeed Prof. Mario Stevens as my promoters, and for their support of my research and especially for their careful review of this dissertation.

I would like to thank my former copromoter Ad Verschueren for his patience and support during my research on mobile network system design. I owe special thanks to Jeroen Voeten who became my copromoter after Ad Verschueren had left. He helped me through difficult times on the course of redefining my research topic. I sincerely thank him for his guidance, advice and support, for his confidence in me to carry out this work, and for his considerable help during the writing of this dissertation. Special thanks also go to Marc Geilen, who taught me a lot about temporal logics. His suggestions have significantly improved the proofs in this dissertation. I would also like to thank Piet van der Putten for his continuous interests in my work.

I am grateful to Prof. Loe Feijs, Jozef Hooman and Prof. Kim Larsen for their valuable comments on the draft version of this dissertation.

This research work has been carried out within the PROGRESS project “Internet-based Monitoring and Control of Embedded Systems”. I would like to thank people involved in this project, especially Prof. Loe Feijs, Johan Lukkien, Piet van der Putten, Andre Ventevogel, Ron Niesten and Henri Aalderink for their constant support and close cooperation.

Next I would like to express my gratitude to Leo van Bokhoven, whose Ph.D work provided inspiration for my research. I also would like to thank Bart Theelen especially for his help in solving many \LaTeX\ problems during the writing of the disser-
I thank Zhangqin Huang, Frank van Wijk, Mark Verhappen, Oana Florescu and Simona Vlad for the interesting discussions that we had.

I would like to thank my colleagues at the (Eindhoven) Embedded System Institute (EESI), especially Marloes van Lierop for her kind support of my work, and Giovanni Russello, Ana Sokolova, Mohammad Mousavi, Kalok Man, Stephan Mietens, Mugurel Ionita, Richard Verhoeven, Harm van Beek and Louis van Gool, for the enjoyable time we have spent together.

I would like to express my gratitude to the secretaries from the (E)ESI and the ICT/ES group for their help in many aspects of my work. Special thanks go to Rian van Gaalen for the helpful English course she gave.

In addition, I would like to thank Rian van Gaalen and Leo van Bokhoven for their help on the translation of my summary into the Dutch language.

I would like to thank the former and current members of the Chinese Student Association (Eindhoven branch), who provide the platform for us to get in touch with Chinese friends and to make us feel not far away from home. I also would like to thank all my friends and colleagues who helped me directly or indirectly in my research and life.

I am deeply indebted to my parents, my mother- and father-in-law, my brother and my sister. Although being far away, they are always supporting me in all aspects of my life. Finally, I would like to give a loving thank to my wife. We encourage and help each other to go through this most challenging period in our life together.

Jinfeng Huang
Eindhoven, July 2005

The research presented in this dissertation is supported by PROGRESS, the Program for Research on Embedded Systems and Software of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW and the Netherlands Organization for Applied Scientific Research TNO through the funding of project EES.5413 “Internet-based Monitoring and Control of Embedded Systems”.
Contents

Summary iii

Samenvatting v

Acknowledgement vii

I Background and problem analysis 1

1 Introduction 3

1.1 Real-time systems 3

1.1.1 The constituents of real-time systems 4

1.1.2 Timeliness: an essential characteristic of real-time systems 4

1.1.3 Temporal constraints on real-time software 5

1.2 Models in real-time software design 6

1.2.1 Benefits of using models 6

1.2.2 Risks of using models 7

1.3 Research objectives 8

1.3.1 Predictability of design approaches for real-time software 9

1.3.2 Real-time property-preservation between timed systems 9

1.3.3 Correctness-preserving real-time software synthesis 10

1.4 Thesis outline 10

2 Design approaches for real-time software 13

2.1 Predictability in multi-stage design process 13

2.2 Design languages 15

2.2.1 Requirement languages 16

2.2.2 Modelling languages 16

2.2.3 Implementation languages 18

2.3 Effective design transformations 18

2.3.1 Design transformations 18

2.3.2 Compositionality 20
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Compositionality in design transformations</td>
<td>22</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Composability in design transformations</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Design approaches for real-time software</td>
<td>24</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Platform-dependent design approaches</td>
<td>24</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Platform-independent design approaches</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Timeliness considerations</td>
<td>34</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Orders of actions</td>
<td>34</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Task-level and action-level scheduling</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary</td>
<td>38</td>
</tr>
</tbody>
</table>

## II Real-time property-preservation between timed systems  41

### 3 Preliminaries  43

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Representations of system behaviors</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Graph representations of untimed behaviors</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Graph representations of timing behaviors</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Representations of real-time properties</td>
<td>47</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Logics for qualitative real-time properties</td>
<td>47</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Logics for quantitative real-time properties</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Summary</td>
<td>50</td>
</tr>
</tbody>
</table>

### 4 Proximity measures between timing behaviors  51

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Formalization of timed systems</td>
<td>51</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Sequential timed systems</td>
<td>52</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Concurrent timed systems</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Timed state sequence space</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Proximity measure based on absolute timing differences</td>
<td>57</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Proximity measure based on relative timing differences</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Discussions</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Related work</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>67</td>
</tr>
</tbody>
</table>

### 5 Weakening real-time properties  69

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Formalization of real-time properties</td>
<td>69</td>
</tr>
<tr>
<td>5.1.1</td>
<td>$MTL$ logic</td>
<td>69</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The semantics of the $until$ operator</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Weakening formulas</td>
<td>72</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Weakening relation derived from sub-formulas</td>
<td>73</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Weakening relation derived from time-bounds</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Weakening functions</td>
<td>74</td>
</tr>
</tbody>
</table>
5.3.1 Operators over time-bounds .................................................. 75
5.3.2 Absolute $\epsilon$-weakening function .................................. 78
5.3.3 Absolute $\epsilon_c$-weakening function .................................. 79
5.3.4 Relative $[x, y]$-stretching function ..................................... 80
5.4 Discussion and summary ......................................................... 81

6 Tube functions 83
6.1 Absolute $[x, y]$-tube function for sequential timed systems ...... 84
6.2 Absolute $[x, y]$-tube function for concurrent timed systems ...... 87
6.3 Relative $[x, y]$-tube function for sequential timed systems ...... 90
6.4 Relative $[x, y]$-tube function for concurrent timed systems ...... 91
6.5 Summary ................................................................. 93

7 Real-time property-preservation between timed systems 95
7.1 Property-preservation based on absolute timing differences: sequential case .................................................. 96
7.1.1 Absolute $\epsilon$-spanning distance ........................................ 96
7.1.2 Property-preservation between timed state sequences ........ 97
7.1.3 Real-time property-preservation between timed systems .... 100
7.1.4 Property-preservation with unless formulas ......................... 100
7.2 Property-preservation based on absolute timing differences: concurrent case .................................................. 102
7.3 Property-preservation based on relative timing differences: sequential case ................................................. 104
7.4 Property-preservation based on relative timing differences: concurrent case .................................................. 106
7.5 An example of real-time property preservation ....................... 106
7.6 Related work .............................................................. 108
7.7 Summary ................................................................. 109

III Predictable real-time software design 111

8 Towards predictable real-time software synthesis 113
8.1 Bridging different time domains .......................................... 114
8.1.1 Several related concepts ................................................. 114
8.1.2 Timing differences between various time domains ............ 116
8.2 Hypotheses for real-time software synthesis ......................... 121
8.2.1 Absolute $[x, y]$-hypothesis ............................................... 121
8.2.2 Relative $[x, y]$-hypothesis ............................................... 124
8.3 Correctness-preserving software synthesis ............................. 125
8.3.1 POOSL .............................................................. 125
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.2</td>
<td>Rotalumis</td>
<td>127</td>
</tr>
<tr>
<td>8.3.3</td>
<td>A toy example</td>
<td>128</td>
</tr>
<tr>
<td>8.4</td>
<td>Discussions</td>
<td>129</td>
</tr>
<tr>
<td>8.5</td>
<td>Summary</td>
<td>131</td>
</tr>
<tr>
<td>9</td>
<td>A case study – a railroad crossing system</td>
<td>133</td>
</tr>
<tr>
<td>9.1</td>
<td>A railroad crossing system</td>
<td>133</td>
</tr>
<tr>
<td>9.1.1</td>
<td>System description</td>
<td>133</td>
</tr>
<tr>
<td>9.1.2</td>
<td>A parallel solution</td>
<td>134</td>
</tr>
<tr>
<td>9.2</td>
<td>Requirement analysis</td>
<td>137</td>
</tr>
<tr>
<td>9.3</td>
<td>System modelling</td>
<td>139</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Initial model</td>
<td>139</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Untimed refinement</td>
<td>141</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Timed refinement</td>
<td>143</td>
</tr>
<tr>
<td>9.3.4</td>
<td>Synthesis model</td>
<td>144</td>
</tr>
<tr>
<td>9.4</td>
<td>System synthesis</td>
<td>144</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Implementation of communication interfaces</td>
<td>145</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Parameter estimation in the absolute $[x, y]$-hypothesis</td>
<td>146</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Parameter estimation in the relative $[x, y]$-hypothesis</td>
<td>147</td>
</tr>
<tr>
<td>9.5</td>
<td>Summary</td>
<td>148</td>
</tr>
<tr>
<td>10</td>
<td>Conclusion</td>
<td>149</td>
</tr>
<tr>
<td>10.1</td>
<td>Contributions</td>
<td>149</td>
</tr>
<tr>
<td>10.2</td>
<td>Options for future research</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>160</td>
</tr>
</tbody>
</table>
Part I

Background and problem analysis

In complex system design, the large gap existing between requirements and realizations has been a pertinacious problem. This holds in particular for real-time software with strict timing constraints and safety-critical requirements. Designers have to rely on a stepwise and piecewise design process, where design decisions are made at different abstraction levels or/and at different components of the system. To ensure the effectiveness of this design process, predictability becomes a crucial feature of a design approach. The goal of design predictability is to allow designers to predict properties of interest for future design outcomes based on already accomplished design results.

In part I of this thesis, we present an overview of existing design approaches for real-time software focusing on their support for predictability. Chapter 1 presents some preliminaries for later discussion. Chapter 2 investigates several principles for predictable design from a semantic point of view. Based on these principles, the predictability support of existing design approaches is analyzed and illustrated by examples.
Chapter 1

Introduction

With the decreasing cost and increasing computational capabilities of processors, we have witnessed widespread use of real-time software in various control applications, such as medical instruments, avionic and flight control, traffic control, telecommunications and consumer electronics. On one hand, the functional complexity of these systems increases dramatically to meet various new emerging demands. On the other hand, these systems have to satisfy strict timing constraints and safety-critical requirements to avoid potential financial loss or even the loss of life. The effective and efficient design of real-time systems has become a major concern for both industry and academia. This thesis addresses the issues of predictability support in real-time system design, which provides opportunities for designers to conquer a complex design problem step by step. We especially focus on the predictable generation of realizations from real-time models.

The purpose of this chapter is to introduce the basic concepts of real-time software design and to present an overview of this thesis. Section 1.1 introduces the constituents and important characteristics of real-time systems, in which real-time software plays a dominant role to ensure correct system functioning. We also show the necessity of a structural design approach for real-time software. It is fundamentally different from those approaches where real-time is not an issue. Section 1.2 introduces several modelling concepts, which play an essential role in coping with the complexity of systems in general and real-time software in particular. Section 1.3 explains the objectives of this research. The structure of this thesis is shown in Section 1.4.

1.1 Real-time systems

In this section, we present some basic concepts of real-time systems.
1.1.1 The constituents of real-time systems

A real-time system is typically divided into three parts (shown in Figure 1.1), a real-time controller, operators and controlled objects. The collective of controlled objects and operators is referred to as the physical environment [55].

The controlled objects usually consist of a set of parallel physical processes, such as sensor organs and arms of a robot or network terminals in a real-time network. The behavior of these controlled objects is often highly dynamic. For example, terminals can send a packet at their own rates and the movement of a robot can be affected by its surroundings. The operators can be optional in a real-time system. They monitor the state of controlled objects and operate on the controlled objects through the real-time controller.

The core of real-time system design is the real-time controller, which is often in the form of real-time software with the following two prominent characteristics:

1. **Concurrency:** Due to the concurrent nature of the controlled objects and the operators, the real-time software is generally composed of a set of parallel processes (tasks). Although it is possible to design a sequentialized real-time software process to control parallel physical processes, design effectiveness and design efficiency are lost in some cases. This problem is especially prominent when there are tens or even hundreds of concurrent processes, each of which has its own functional and timing constraints as well as intricate dependency relations with other processes.

2. **Timeliness:** The real-time software must react to the stimuli from the controlled objects or to the operations from the operators under certain timing constraints. In general, these reactions must conform to the requirements of the environment. This indicates that the timing behavior of real-time software has to be reasoned about in the (external) physical time domain, instead of in the (internal) hardware time domain.

1.1.2 Timeliness: an essential characteristic of real-time systems

An essential characteristic of real-time systems that distinguishes them from non-real-time systems is timeliness. This characteristic states that the correctness of a real-time control system depends not only on the correct output values of the computa-
tion but also on the correct physical time at which the outputs are issued.

We classify real-time systems according to possible consequences caused by a timing failure (a wrong action issuing time) during system operations. This is different from the criterion generally adopted in literature, which classifies real-time systems based on the consequences resulting from the missing of a deadline. Timing failure covers various possible timing constraints and has a more general meaning than the missing of a deadline. For example, a missile interception system requires that the missile is fired within a small time interval to intercept the coming missile. Even if the deadlines of a system are always met, earlier firings can still result in interception failures.

Based on the consequences of timing failures, real-time systems can be roughly classified into three categories: hard, firm and soft real-time systems [82]. Hard real-time systems do not tolerate any timing failures that may lead to a catastrophe. Examples of hard real-time systems are nuclear control systems and flight control systems. Soft real-time systems can accept timing failures. These result in the degradation of system performance. A typical example of soft real-time systems is an online flight reservation system. Firm real-time systems allow occasional timing failures that result in useless responses. An example is a multimedia network application, in which several lost or delayed packets are acceptable. Besides these categories, the concept of weakly hard real-time software is introduced by [14]. It allows occurrences of a few timing failures in a predictable way.

1.1.3 Temporal constraints on real-time software

The timing behavior of real-time software can be viewed as a set of action sequences, each of which represents an execution of the software. Generally speaking, temporal constraints on real-time software are conditions on the timing of actions, such as action enabling, firing, synchronization and termination [46]. These temporal constraints of real-time software are ideally specified explicitly on individual tasks or actions [61]. By verifying whether the system actually meets these temporal constraints, designers can ensure the correctness of the system and produce more reliable real-time software. Two categories of techniques are often applied to examine the satisfaction of temporal constraints in a real-time software system: schedulability analysis and formal verification.

- **Schedulability analysis**: A real-time software system is considered as consisting of a set of tasks, each of which is characterized by a set of parameters (constraints), such as worse-case execution times (WCET), deadlines and priorities. The aim of real-time scheduling is to devise a feasible scheduling scheme, which assigns computation time to each task on the basis of its parameters to ensure that all temporal constraints are satisfied.

- **Formal verification**: Both the timing behavior and the temporal constraints of real-time software are formalized. For example, critical temporal constraints of real-time software can be expressed by real-time temporal logics [8] and the timing behavior of real-time software can be modelled by formal frameworks such as timed CCS [72] and timed automata [5]. By making use of techniques
such as model checking [23] and theorem proving [103], the satisfaction of temporal constraints by real-time software can be analyzed formally.

Both techniques can assist designers to analyze critical temporal constraints of real-time software, but they are not complete solutions for designing predictable real-time software. For example, schedulability analysis provides techniques for determining a set of temporal constraints (such as deadlines) for a restricted class of real-time systems. However, it lacks the support to reason about functional properties of a real-time system. Formal verification focuses on property analysis of a formal model, but it is not concerned with preserving properties from a model to its realization. A small timing deviation between the model and its realization may cause properties that are valid in the model to be invalid in the realization. In Chapter 2, the merits and pitfalls of both techniques for real-time software design are further investigated.

1.2 Models in real-time software design

The evolution of software exhibits a rapid growth of complexity to meet demands that are increasing in number and diversity. For example, the early operating system MS-DOS 1.0 only contained 4000 lines of assembly code and took a single designer several weeks to design, while the recent operating system Windows XP contains about 40 million lines of code and requires thousands of man-years to design [63]. The evolution of real-time software follows a similar path in this aspect. Large real-time software systems, such as telephone switches and flight control systems, can consist of millions of lines of source code nowadays. It is too expensive and time-consuming or sometimes even infeasible to use the traditional code-centric design approaches [86], where designers express their thoughts and ideas using an implementation language directly, to build software for these complex systems.

To cope with the increasing complexity, designers have to resort to a multi-step design process. At each step of the process, designers either refine a high-level system description with more details to approach the realization, or abstract a low-level system description by removing irrelevant details to improve its comprehensibility. By step-wise model transformations (refinements or abstractions), the gap between system requirements and system realizations can be gradually filled. Furthermore, the use of models also facilitates communications within the design team. In the following, the major benefits and risks of using models in software design are discussed in more details.

1.2.1 Benefits of using models

Generally speaking, a model is an abstract representation of a physical system, which can help designers to understand and solve design problems more effectively and efficiently during system design. Modelling techniques can offer many benefits to the construction of correct real-time software. These benefits are illustrated as follows.

- Economic feasibility: For many modern real-time software systems, it is often too
costly to build and experiment with their physical realizations to learn how to improve design solutions. Sometimes it is even infeasible to experiment with their physical realizations, e.g., in the case of the control software for nuclear reactors. Making an adequate model to predict critical properties of the software, such as performance and safety properties, can avoid expensive and time-consuming design iterations.

- **Efficiency**: The use of models can improve the efficiency of software design at least in the following two aspects. First, models abstract realizations from irrelevant information, which helps designers to focus on essential problems and evaluate alternative solutions efficiently. Second, models created at a high level of abstraction represent a set of possible low-level implementations. As a consequence, the reusability of the model for different applications is enhanced. For example, a platform-independent model can be implemented on different platforms.

- **Correctness preservation**: During software design, the software model can evolve from a high-level to a low-level of abstraction, while keeping the desired properties that were verified at the higher level. This process is called a *correctness-preserving model refinement*. Furthermore, a unique and significant advantage of using software models is that they can be directly transformed into realizations using computer-aided tools [83]. If a transformation can keep the desired properties of a model in its realization, it is referred to as *correctness-preserving system synthesis*.

Here we only listed several major benefits gained by using modelling techniques. It is fair to say that models play an essential and indispensable role during the design of complex software. However, many industrial practices have also shown that software models do not always deliver the expected merits, several important reasons of which are explained in the following subsection.

### 1.2.2 Risks of using models

Generally speaking, the growth of system complexity is due to the increase of the variety (distinction), and dependency (connection) of parts or aspects of a system [41]. To reduce the complexity of software, certain assumptions are often made in software models, which intend to remove irrelevant information and weak dependencies within a system. For example, formal models of mobile networks treat mobility and functionality in an orthogonal way [47, 84] and formal models of real-time systems treat time progress orthogonally from behavioral progress [5, 39, 73].

Without awareness and careful treatment of the assumptions put on the model, analysis results derived from the model may be incorrect or inaccurate in the realization, where some assumptions are not valid anymore. Failures of many real-life computer systems have demonstrated this. Here are two examples to demonstrate that the violation of assumptions made during system design can lead to failures during system operation.

- **Denver airport baggage handling system** [75]: The automated system of the new
Denver airport was supposed to improve baggage handling by using a computer tracking system to direct baggages contained in unmanned carts running on a track. Significant mechanical and software problems were detected during system testing. This unfinished system postponed the opening of the airport from March 1994 until February 1995. The delay resulted in a loss of roughly $1 million per day on operation cost and interest. The reason for the above failure can be largely attributed to the violation on assumptions made during system design. For example, designers explicitly or implicitly assumed that bags never fall from carts during the transportation in their design. When the system was put into use, some bags fell onto tracks causing carts to jam and further broke down the whole system.

- Patriot missile defense system [69]: During the Gulf war on 25 February 1991, a Patriot missile defense system failed to intercept an incoming scud rocket, which resulted in 29 people killed and 97 people wounded in an American military barrack in Dhahran. The failure was caused by a 0.3433 second drift over a period of 100 hours, which led to a tracking error of 678 meters. The reason causing the problem was also identified. The system was initially assumed to be in continuous operation for no more than 14 hours. However, no mechanism was provided to automatically prevent the incorrect usage of the system.

To ensure the validity of the conclusions derived from a model for later design, model assumptions have to be carefully defined and be conformed to in later design stages. In the example of the Denver airport baggage handling system, designers assumed that bags would never fall from carts. The later realization of the system should try to avoid the violation on the assumption by taking measures, such as limiting the maximum load of the carts or manually checking the tracks. Similarly, in real-time software design, certain assumptions are often made on the execution times of tasks or actions (such as instantaneous actions and worse case execution times). These assumptions should be reasonable and should hold in later design steps. In the case that the assumptions can not be fully satisfied, the influence on succeeding design stages should be estimated to avoid incorrect prediction for the behavior of a future realization.

In addition to the assumption consistency problem, making a proper abstraction for the investigated design issues of the system is no easy task. It often requires designers to have a good understanding of the design issues and the ability to capture the system essentials.

1.3 Research objectives

Due to the complexity of real-time software, it is necessary to carry out system design in a stepwise and/or piecewise manner. From the stepwise perspective, the system is specified and analyzed at different levels of abstraction. From the piecewise perspective, the system is constructed by a set of components. In both perspectives, it is crucial for a design approach to ensure that design decisions based on a partial system (e.g. in the form of an abstraction or/and a component) are still valid in a
more complete system. To this end, the design approach should support predictability, i.e. the capability of a design approach to predict desired properties of future design outcomes based on already accomplished design results.

The research in this thesis focuses on the following interrelated topics:

- predictability of design approaches for real-time software;
- real-time property-preservation between timed systems;
- correctness-preserving real-time software synthesis.

### 1.3.1 Predictability of design approaches for real-time software

In the past decades, a number of design approaches for real-time software have been proposed by both academic institutions and industrial organizations. It would be beneficial to compare the typical approaches and to sort out what has been accomplished and what needs to be done. We investigate the predictability support of these approaches.

More specifically, we classify existing design approaches for real-time software into two categories: platform-dependent design approaches and platform-independent design approaches, based on the timing concepts adopted for system modelling. Platform-dependent design approaches often interpret the timing behavior of a model in the hardware time domain, where time is counted by a hardware clock. Platform-independent design approaches often interpret the timing behavior of a system model in a virtual time domain, where time is counted by a system variable. With respect to design approaches in each category, we investigate the predictability of four major design activities: model analysis, model integration, model transformation (refinement/abstraction) and system synthesis. Model analysis refers to the activity of analyzing properties of a model or locating errors of a model. Model integration refers to the activity of integrating smaller components into a larger one. Model transformation refers to the activity of refining a model with more implementation details or abstracting a model by removing irrelevant information. System synthesis refers to the activity of transforming a model into a realization.

### 1.3.2 Real-time property-preservation between timed systems

Platform-independent design approaches provide better predictability support than platform-dependent design approaches for model analysis, model integration and model transformation. However, in platform independent design approaches, due to the assumption of instantaneous actions in the model and non-controllable physical time in the realization, it is impossible for designers to ensure that corresponding actions in the model and the realization are always observed at exactly the same moment. The model can only be an approximation of its realization w.r.t. the issuing time of actions. As a result, real-time properties verified in the model cannot be directly carried over to the realization.
We formalize a real-time system (either a model or a realization) as a timed system. We aim at building real-time property relations between timed systems, so that real-time properties of one timed system can be predicted from those of the other.

1.3.3 Correctness-preserving real-time software synthesis

In principle, models of software systems can be automatically transferred into realizations with the aid of computer-based tools. Such an automatic transformation can significantly reduce the design cost and time and improve the productivity. If a link can be established between the semantic domains of models and realizations, the correctness of transformations can be guaranteed resulting in reliable realizations.

However, in most existing design approaches for real-time software design, the automatic transformation from a model to its realization lacks mechanisms to ensure the consistency between their timing semantics. Designers may find that the generated realization exhibits totally different real-time properties than the model does. This is denoted as unpredictability during software synthesis.

Based on the real-time preservation results for timed systems, we aim at proposing a software synthesis approach, which can guarantee the correctness of the realization during software synthesis.

1.4 Thesis outline

The overall structure of the thesis is illustrated in Figure 1.2. The thesis is roughly divided into three parts. Part I gives an overview of existing design approaches for real-time systems. By investigating these approaches, predictability problems are identified. Part II is devoted to the formal investigation of the preservation of real-time properties between timed systems. Finally, Part III presents a predictable synthesis approach for real-time software, which is based on the property-preservation results in Part II. A case study is given to demonstrate how the proposed approach can be applied in practice.

- Part I: Background and problem analysis
  - Chapter 1: Introduction
    The content of the research is sketched. The objectives and organization of the thesis are presented.
  - Chapter 2: Design approaches for real-time software
    This chapter gives a survey of existing design approaches for real-time (especially software) systems. Merits and pitfalls of these approaches are briefly discussed. The major problems causing design unpredictability for real-time systems are also analyzed.

- Part II: Real-time property preservation between timed systems
  - Chapter 3: Preliminaries
    In this chapter, a brief introduction is given of the mathematical basis employed in this part.
Figure 1.2: The outline of the thesis

- Chapter 4: Proximity measures between timing behaviors
  In this chapter, we formalize the timing behavior of timed systems and propose two different proximity measures to specify the “distance” between timed systems.

- Chapter 5: Weakening real-time properties
  In this chapter, we use the linear-time temporal logic MTL to formalize real-time properties. A weakening relation between MTL formulas is investigated. Several weakening functions are proposed for MTL formulas, which can be used to establish real-time property relations between timed systems.

- Chapter 6: Tube functions
  In this chapter, we propose two special functions (called tube functions), which give another characterization of the proximity between timed systems. The properties of the tube functions facilitate the proof of real-time property-preservation.

- Chapter 7: Real-time property-preservation between timed systems
  In this chapter, we prove real-time property relations between two timed systems. The results show that real-time properties between two timed systems can be preserved up to a small deviation. Furthermore, the extent of the deviation can be quantitatively predicted on the basis of the proximity of the timed systems. The preservation results hold for both sequential and concurrent timed systems.

- Part III: Predictable real-time software design
– Chapter 8: Towards predictable real-time software synthesis
In this chapter, we apply the real-time property-preservation results to real-time software synthesis. More specifically, two parameterized hypotheses are proposed to capture bounds on timing differences between a model and its realization. The hypotheses are used to predict real-time properties of the realization from those of the model. Furthermore, we propose a synthesis approach to show that both hypotheses can indeed be complied with during software synthesis.

– Chapter 9: A case study – a rail-road crossing system
A rail-road crossing system is designed using the proposed approach. It demonstrates that this approach can actually guide designers to preserve the correctness of a model into its realization during the software synthesis.

– Chapter 10: Conclusions
Contributions of this work and directions for future work are summarized in this chapter.
Chapter 2

Design approaches for real-time software

In the past decades, a number of design approaches for real-time software have been proposed by academic institutions and industrial organizations. These approaches adopt varying design schemes based on different design assumptions. It would be beneficial to compare typical approaches and to sort out what has been accomplished and what needs to be done. In this chapter, we review and compare existing design approaches for real-time software. Special focus is put on their predictability support.

This chapter is organized as follows. Section 2.1 first defines predictability and shows its importance in a multi-stage design approach. Section 2.2 and 2.3 investigate the basic requirements for design languages to support predictable design. Based on this investigation, Section 2.4 identifies the absence of predictability support in existing design approaches for real-time software, which is demonstrated for both platform-dependent and platform-independent design approaches. Section 2.5 points out some limitations of both platform-dependent and platform-independent approaches based on timeliness considerations. Section 2.6 summarizes this chapter.

2.1 Predictability in multi-stage design process

The aim of real-time software design is to fill the gap between requirements and realizations. However, this design gap has increased tremendously, due to the continuous increase of the functional complexity of real-time software systems, and the stringent timing requirements imposed on them. Since traditional code-centric design approaches are obviously not capable of coping with this increasing complexity, designers have to resort to a multi-stage design process. At each design stage, only a part of the system is considered, so that some specific design problems can be addressed and tackled.

In general, predictability refers to the capability (of a design approach) to make de-
Design decisions based on accomplished work, which not only hold at the current design stage but also hold for later design stages. Since early design stages only involve partial information of the system (e.g. in the form of an abstraction or/and a component), it is challenging to ensure that design decisions made on the basis of a partial system are still valid in a more complete system.

A multi-stage design process can be carried out in a stepwise or/and piecewise manner. During a stepwise design process, a series of design decisions are made at different abstraction levels. At each abstraction level, only a subset of the desired properties of the system are investigated. To smoothen such a design process, it is crucial for a design approach to support correctness preservation, allowing desired properties to carry over between different abstraction levels of the system. As a consequence, the properties of interest verified at a high level of abstraction can be preserved into the realization in a stepwise manner.

During a piecewise design process, the system is constructed by the recursive composition of separately exploited components. Each component only contains a part of the total functionality of the system. To ensure that design decisions made in each component are still valid in the integrated system, compositionality is considered as a key feature of a design approach emphasizing semantic independency of components. As a consequence, the behavior of each component remains unchanged during the integration, and properties of the integrated system can be derived from those of the components.

Design of a complex system can involve both stepwise and piecewise design processes. The transformation of a system from one abstraction level to another can be achieved by a set of independent transformations of its components. Compositionality ensures that the transformation of each component can be carried out independently. On the other hand, the integration of components are usually reasoned about on the basis of the integration of their abstractions, which is ensured by correctness preservation. Therefore, compositionality and correctness preservation are two interdependent and indispensable features of a design approach. Figure 2.1 illustrates the relationship between the two concepts.
In the next section, we show that the semantics of a design language (requirement, modelling or implementation language) has a direct impact on the predictability support during the stepwise and piecewise design process.

### 2.2 Design languages

A design approach identifies a set of comprehensive and consistent concepts (such as abstraction, concurrency and communication) which guide designers from initial design ideas to a complete design solution in a systematic way. These design concepts are often supported by languages, rules, techniques and tools, among which design languages play a fundamental role. The goal of design languages is to provide a set of primitives (or vocabularies), by which designers can effectively express their thoughts. According to the different abstraction levels of design thoughts, three categories of design languages, requirement, modelling and implementation languages, are often involved in a design process.

In general, a design language comprises of two indispensable elements, the syntax (the form) and the semantics (the intended meaning). Clearly, the language itself should be equipped with a precise syntax and unambiguous semantics to describe the domain knowledge. At the same time, designers should fully understand the semantics as well as the syntax of every construct of the language to effectively and accurately express their ideas.

In contrast to the syntax of a language, which can be precisely described and analyzed by scanners and parsers based on a widely-known and well-developed formal theory [91], the semantics of design languages is often inadequate in current practice, due to the following facts:

1. Constructing an adequate semantics for a design language requires a deep insight in the application domain, which can only be obtained with enough domain knowledge. This explains the fact that formal semantic theories often lag behind the emergence of new application domains. For example, the design of concurrent systems and real-time systems can be traced back in as early as the sixties [37], but many important theoretical results of semantics concerning concurrent and real-time systems were obtained only in the eighties and nighties respectively [95].

2. Designing an expressive and concise language with an adequate semantics for the design domain is no easy task. It requires multi-disciplinary knowledge and sufficient domain skills, such as a good understanding of rigorous mathematical theory and rich design experience.

Despite these facts, significant efforts have been made to incorporate formal semantics into design languages, for example, temporal logics for formalizing real-time requirements [7], SDL [90] and SystemC [11, 60] with the interleaving semantics for concurrent behaviors, and C [74, 77] with operational or denotational semantics for sequential behaviors, to mention just a few. In the sequel, we analyze the role of three categories of design languages (requirement, modelling and implementation
language) and introduce the desired properties they should possess to achieve predictable design for real-time software.

### 2.2.1 Requirement languages

Requirements express the needs and constraints that are put upon a system, each of which is a property that must be present in realizations in order to satisfy the specific needs of some real-world application [56]. Usually, requirements are written in natural languages. However, due to the ambiguity of natural languages, complex concepts are usually very difficult to specify precisely. This can result in errors and iterations during the design process.

The formal semantics is proposed as a solution to solve the ambiguity problem. It is embedded in requirement languages to promote comprehensive requirement specification and to facilitate the automatic checking of requirement consistency and completeness. This assists in better understanding and structuring the problem, even when these requirements are not verified against a model or a realization yet [85] ¹. Furthermore, the formal semantics can also contribute to validating design solutions. For example, design solutions can be checked against the requirements (properties) based on formal verification techniques.

In the context of real-time software design, (real-time) temporal logics [7, 13] are the most commonly used languages (or mathematical frameworks) for the formalization of real-time properties of a system. Other frameworks also exist to formally express real-time properties, such as timed automata [5] and duration calculi [106]. The detailed investigation and comparison of formal requirement languages is outside the scope of this thesis. Interested readers are referred to [35].

### 2.2.2 Modelling languages

System modelling is the most challenging and creative activity of the design process. During system modelling, designers need first to understand thoroughly the requirements, carefully explore the design space and finally devise a design solution. The design outcome (model) serves as the basis for later system synthesis, the success of which depends to a large extent on the model itself.

Due to the potential complexity of real-time software, the modelling of such a system can be accomplished by taking a number of stages, where the modelling activities are carried out in a piecewise and stepwise manner. Each stage only considers a part of the system that is relevant to part of design problems.

To improve the likelihood of success of system modelling, modelling languages should be equipped with adequate semantics to assist designers in specifying the desired aspects of the system and to analyze the system behavior of interest. Furthermore, predictability support is crucial for model transformation and model integration. To this end, it is desirable that the semantics of modelling languages for real-time software should possess the following characteristics.

¹Although it is often unrealistic to formalize all the requirements of the desired system in practice, we believe, at least, that critical timing and safety requirements should be precisely specified.
Adequate expressive power

A modelling language should be capable of adequately describing the system behavior. With respect to real-time software, it should allow designers to define data types, to specify system structures and to express concurrency, communication and timing precisely [51]. Furthermore, a modelling language should also facilitate designers to express their thoughts as succinctly as possible.

To enable designers to investigate the design space at different abstraction levels, (the semantics of) a modelling language should also support non-determinism. Non-determinism refers to the situation where the next state of a computation is not determined uniquely by the current state and the given action. Non-determinism often arises when the details of the system are abstracted away. In general, non-determinism not only leaves freedom to obtain optimal realizations but also relieves designers of the burden to consider more details than necessary in a certain design step.

Effective monitoring mechanism

During system design, significant design efforts are spent on observing the faulty behavior and detecting design errors by simulation. For example, 30 – 40% of the design time in embedded system design [64] and 50 – 70% of the design time in distributed and real-time system design [94] are spent on these activities. The effectiveness therefore is closely related to the applied debugging and analysis mechanisms, which enable designers to monitor the states of interest and to evaluate important performance metrics. In traditional design approaches, the monitoring and debugging mechanisms are often supported in the realization phase. With the increasing complexity of systems, an effective monitoring and debugging mechanism is also demanded during system modelling.

Generally, the debugging and analysis mechanisms are implemented by inserting monitoring code into the original software program. To ensure the validity of debugging and analysis results, the semantics of modelling languages should guarantee that the observable behavior of the system is not disturbed by the monitoring code. For real-time software design, it is of utmost importance to keep the timing behavior of a system unaffected by the monitoring code.

Effective model transformation

It is far from simple to get a thorough and clear picture of a large real-time software system. To facilitate the construction of a large-scale system, two mutually inverse transformations, abstraction and refinement, are continuously preformed during system modelling. Abstraction removes irrelevant design details from models, while refinement adds more design details to models. To ensure the consistency between models during this process, the semantics of modelling languages should guarantee that removing and adding design details does not lead to property (especially real-time property) inconsistencies. A more detailed explanation about model transformations will be presented in Section 2.3.
2.2.3 Implementation languages

System synthesis is an activity that converts a model into a realization. During this phase, the system is often depicted by an implementation language (such as Java, C and C++), the semantics of which is usually related with and constrained by the target platform. Due to the different notions used and assumptions made at the modelling phase and the implementation phase, it is not always straightforward to correctly transform a model into a realization. As a consequence, it is difficult to guarantee the validity of the realization w.r.t. the properties that have been verified in the model.

The difficulty of maintaining correctness between a model and its realization is attributed to several reasons. First, during system modelling, certain assumptions are made about the semantics of modelling languages in order to effectively explore the design space. These assumptions are valid at certain abstraction levels, but they do not always hold for the semantics of implementation languages. For example, to facilitate the analysis of the timing behavior of a model, it is often assumed that actions are instantaneous. However, every action does take a certain amount of execution time in any implementation language. Without carefully considering this difference during system synthesis, the realization may exhibit an entirely different behavior than the model does (see Examples 2.3 and 2.4). Second, some primitives and operations defined in modelling languages do not have direct correspondences in implementation languages. For example, during system synthesis, parallel operations in the model are often implemented by means of a specific thread mechanism offered by the target operating system, the semantics of which is not always consistent with that of the modelling language.

In most existing design approaches for real-time software, the gap between the timing semantics of the modelling language and that of the implementation language is not properly bridged. The generated realization may exhibit a different system behavior than the model does. A more detailed investigation of real-time software synthesis will be presented in Section 2.4.

2.3 Effective design transformations

As we have seen in the previous section, it is necessary for the semantics of modelling languages to facilitate effective transformation between different abstraction levels. To this end, compositionality should be supported by the semantics of modelling languages.

2.3.1 Design transformations

Abstraction and refinement are two elementary transformations performed during the design process (as shown in Figure 2.2). Abstraction is the activity that tries to remove (or hide) irrelevant information, which improves the comprehensibility of existing design models and facilitates the evaluation of different design solutions. The major goal of abstraction activities is to improve the understandability of the
design, thereby assisting designers to make correct design decisions. Refinement is the activity that adds more implementation details to models, thereby reducing the gap between models and realizations. The major goal of refinement activities is implementability. Intuitively speaking, abstraction activities intend to clarify what the system (component) can do, while refinement activities intend to clarify how the functionality of the system (component) can be achieved.

We can see from Figure 2.2 that an intermediate design outcome (model) plays dual roles in the design process. It is the abstraction of its lower-level design outcomes and the refinement of its higher-level design outcomes. In other words, it states what the functionality of the lower-level outcomes is and describes how the functionality of the higher-level design outcomes can be implemented. It is natural to see that the top-level of the design process in Figure 2.2 refers to desired properties (requirements), while the bottom-level refers to a realization. Actually, the design process indeed intends to fill the gap between the desired properties (what the system should be) and the realization (how the system functions).

Now the question arises on what condition two design outcomes form an abstraction/refinement pair. In design practice, we can use the following principle.

System A is an abstraction of B (thus B is the refinement of A) iff for any desired property \( p \), if \( p \) is satisfied by \( A \), then \( p \) is also satisfied by \( B \).

In a specific formal framework, a more formal definition can be given. For example, in a state trace based formal framework, each property or model can be interpreted as a set of state traces. The satisfaction relation between a model and a property is defined by the inclusion relation between their corresponding sets of state traces. A refinement actually restricts the set of state traces of a model into one of its subsets. If the set of state traces of the property includes that of the abstraction, then it should also include the set of state traces of the refinement. This is consistent with our abstraction/refinement principle.

Note that Figure 2.2 only gives a single flow of refinement and abstraction transformations. In practice, the design process can be much more complex than depicted in Figure 2.2. Figure 2.3 presents an example of a transformational design process, where the design objective is to devise a realization satisfying five properties \( P_0 \) till
2.3 Design approaches for real-time software

The design process starts from an abstract model ($M_a$). By stepwise refinement, more detailed models (such as $M_b$ and $M_c$) are created to satisfy more properties. Finally, the realization is generated based on model $M_c$. During each refinement step, different refinements can be generated based on the creativity and experience of designers. For example, the refinement of $M_a$ could be $M_b$, $M_d$, or $M_e$. Some of these models may be discarded during the design process, since they might not satisfy some of the system properties. For example, model $M_e$ may not satisfy $P_2$, while the others may evolve to different qualified realizations.

2.3.2 Compositionality

A design process can be considered as a set of transformation activities, the effectiveness of which can significantly affect the required design time and cost. This holds in particular for large-scale software systems. To reduce the design complexity, a system can be considered as consisting of a set of interacting components. Compositionality is often regarded as an important characteristic of the semantics of a design language, which can facilitate effective transformation for complex systems.

The well-known principle of compositionality [67, 78] states that

the meaning of a design description is determined by the meanings of its parts and of the syntactic rules by which they are combined.

It is originally proposed to guide the association of the semantics and the syntax of a design language and to assist designers in understanding the meaning of a complex design description in a structured way.

To illustrate this, we can use a tree structure to represent the form of a design descrip-
effective design transformations 21

Figure 2.4: Compositional meaning assignment

Figure 2.5: Semantic independence of compositionality

tion (see Figure 2.4(a)), where each leaf of the tree is a syntactic primitive and where all other nodes represent syntactic combination rules. To ensure compositionality, the assignment of meaning to such a syntax tree follows the rules listed below:

- Each syntactic primitive $e_i$ is given a meaning independently from the other primitives by some meaning assignment function $f$.
- For each syntactic combination rule $F_j$, there is a corresponding semantic operation $G_j$ that assigns meanings to the (sub-)tree rooted by $F_j$ (see Figure 2.4(b)).

A compositional meaning assignment operation $[\cdot]$ can be depicted as:

$$[F_i(D_1, ..., D_n)] = G_i([D_1], ..., [D_n])$$  \hspace{1cm} (2.1)

In this equation, $F_i(D_1, ..., D_n)$ represents a piece of design description, which is formed by its sub-descriptions $D_1, ..., D_n$ by using syntax combinator $F_i$. We can see that the meaning of $F_i(D_1, ..., D_n)$ entirely depends on semantic operation $G_i$, and the meanings of its sub-descriptions.

Since the semantics of a design description is only related to its parts and their combination rules without being interfered by parts outside the scope, compositionality assists the understanding of a complex design description in a structured way. For example, in Figure 2.5, the meaning of description $F_i(e_{1a}, e_{1b})$ entirely depends on its internal elements $F_i, e_{1a}$ and $e_{1b}$ independently of its context. It has exactly the same meaning in both descriptions depicted in Figure 2.5(a) and Figure 2.5(b).
However, due to the potential complexity of the syntax tree, the semantic interpretation of a complex design description can be far from simple. For example, the syntax tree of the description shown in Figure 2.4(a) only has three levels, while its semantic interpretation is already not straightforwardly comprehensible anymore. We can easily foresee that the interpretation of a syntax tree with hundreds of levels, which is not unusual for a complex design description, could easily grow beyond human’s understanding. Therefore, compositionality alone does not guarantee that the meaning of a recursively composed syntax tree can be easily understood.

2.3.3 Compositionality in design transformations

Although compositionality alone is not sufficient to cope with complexity, it offers many benefits to reduce design complexity and to improve design efficiency. The compositional semantics divides a complex system into isolated semantic components and ensures the semantic independency of each component in the system. Thus, the transformation of the whole system can be achieved by local transformations of its components and the mapping of combinators to corresponding ones at another level (see Figure 2.6). Furthermore, the correctness of the transformation activities can be also verified locally.

One example of design languages equipped with a compositional semantics is CCS (Calculus of Communicating Systems) [66]. Based on the compositional semantics of CCS, observational equivalence is defined, which states that two models are observational equivalent if and only if both models exhibit the same communication behavior to the external observer. The observational equivalence relation provides the theoretical basis for transformational design approaches. The model of a component can be iteratively refined or abstracted into equivalent models either with more details for implementing purposes or with less details for better comprehensibility. In this way, observational equivalence can effectively assist the transformation of a design description. More detailed discussions about transformational design approaches can be found in [54, 65, 97, 99].
In principle, the compositional semantics of a language can facilitate the accomplishment of design transformation in both top-down and bottom-up design paradigms.

In the top-down design paradigm, a design process starts at a very abstract level. The initial system is relatively simple and only includes the essential behavior w.r.t. the desired (global) properties. The correctness of the initial system can be verified with the aid of verification tools. Designers refine the initial system by adding details in each successive refinement. During each refinement step, more relevant details are added to the system at the corresponding abstraction level. At each abstraction level, the system is divided into a set of sub-systems, each of which is independently refined into a more detailed sub-system (or a set of interacting sub-systems). The compositional semantics ensures that the correctness of the refinement can be verified locally \[22\]. Such a refinement process is also called a correctness-preserving refinement, because all properties of the original system are satisfied by the newly generated system. In this way, the design space can be effectively explored at different abstraction levels and the design decisions can be made through successive refinements.

In contrast to the top-down design paradigm, the bottom-up design paradigm starts a design process from a set of detailed sub-systems. By exploring different ways to combine these sub-systems, designers expect to find a design solution satisfying desired properties. In this process, designers need to deal with property verification of a combined system based on its sub-systems, which often suffers from the state space explosion problem. The compositional semantics of a language provides an effective way to do so. First, the compositional semantics ensures that the behaviors of individual sub-systems are independent from each other. Hence, the verification of desired local properties can be carried out separately, making the formal verification feasible. Consequently, the verified properties can be considered as abstractions of the original system. This process can also be considered to be a correctness-preserving abstraction. In this way, different composed systems can be verified effectively based on the abstraction of each constituent without having to consider their detailed internal behavior.

In summary, suppose $S \overset{\text{def}}{=} P_1 \oplus P_2 \ldots \oplus P_n$ is a system expressed in a language with a compositional semantics, where $P_1, P_2 \ldots P_n$ are components of $S$ and where $\oplus$ is a combinator of components. The compositional semantics guarantees that transformations (abstractions or refinements) $P'_1, P'_2 \ldots P'_n$ of $P_1, P_2 \ldots P_n$ can be carried out independently. Therefore the transformation of $S$ can be $S' \overset{\text{def}}{=} P'_1 \odot P'_2 \ldots \odot P'_n$, where $\odot$ is the corresponding mapping of combinator $\oplus$ in $S$. In practice, $S'$ can be expressed in the same language as $S$ or in a totally different language. For example, properties of a system written in a requirement language can be abstractions of those written in a modelling language.

### 2.3.4 Composability in design transformations

The concept of compositionality is intuitively useful in achieving effective transformation during the design of complex systems. However, in practice, it may not be always as effective as expected. An important reason is that it does not put any restrictions on the assignment of meaning to combinators. As a consequence, semantic
independency can always be achieved by assigning trivial semantics to combina-
tors [105]. In practice, the semantics of combiners in abstractions and refinements
should be simple enough. For example, the semantics of the combiners $\parallel$ and $+$ in
CCS is defined in a natural way and can be understood easily. Furthermore, during
the transformation of a process from one abstraction level to another in CCS, its sub-
processes can be transformed independently from each other and the combiners
between them remain unchanged.

In the context of concurrent systems, a stronger “version” of compositionality is
sometimes called composability. Composability states that properties satisfied by
individual components of a system should be satisfied by their parallel composi-
tions [88]. In [43], this concept is also called compositional verification. For example,
assume reactive system $S$ consisting of two parallel components $P$ and $Q$ has a tim-
ing response property $\varphi$, which states that every environmental stimulus $p$ must be
followed by a response $q$ within 3 seconds. If $P$ satisfies $\varphi$ and the design language
supports composability, then $S \overset{\text{def}}{=} P \parallel Q$ should also satisfy $\varphi$.

More generally, consider a system $S \overset{\text{def}}{=} P_1 \parallel P_2 \ldots \parallel P_n$ expressed by a language sup-
porting composability, where $P_1, P_2, \ldots, P_n$ are components of $S$ and $\parallel$ is the parallel
combinator. Assume each component $P_i$ satisfies property $\varphi_i$. Composability of a
design language states that $S$ satisfies the simple logical conjunction of these indi-
vidual properties ($\varphi_1 \land \varphi_2 \ldots \land \varphi_n$). We can see that only the parallel operator ($\parallel$) and
the logic conjunction ($\land$) are used in composability and their semantics are defined
independently of the semantics of composed components.

2.4 Design approaches for real-time software

In previous subsections, we have introduced several elements that a design language
should possess to support predictable design. In this section, we investigate how
these elements are supported by existing real-time design approaches. To clarify our
illustration, we classify existing approaches into two categories, platform-dependent
approaches and platform-independent approaches, based on different timing con-
cepts adopted. This is a justifiable classification because approaches adopting the
same concept of timing often provide similar predictability support during the de-
sign process. Briefly speaking, platform-independent approaches use a system vari-
able to represent time (denoted as virtual time), while platform-dependent approaches
often adopt physical time $^3$ to represent time progress. This implies that the timing
behavior of the system depends on the underlying computing platform.

2.4.1 Platform-dependent design approaches

Platform-dependent approaches take platform computation constraints into consid-
erations during system modelling, and they use the hardware time to specify the tim-
ing behavior in their modelling languages. Examples of these languages are Rose-

$^3$Physical time can be considered as hardware time, which is counted by a hardware clock in this
chapter.
Inadequate timing semantics

In most platform-dependent design approaches, the semantics of timing expressions, such as the delay expression, are often too ambiguous to express the timing behavior of real-time software. For example, CSDL [1], a design approach using the SDL-96 language, relies on an asynchronous timer mechanism based on a global machine clock. In this time mechanism, the delay between the moment of the timer expiration and the moment at which the model reacts to this expiry is potentially unbounded [59, 100]. The unbounded delay can be caused by several factors, such as the waiting time of a timer-expiration message before it is inserted into the input message queue of a process, the time for consuming all messages that arrived before the timer message, and the interaction time between the process and its input message queue [59]. When a timer is set to \( t \) seconds, it should actually be interpreted as an arbitrary time duration \( d_t \) (\( d_t \in [t, \infty) \)). The unbounded uncertainty of \( d_t \) contradicts with the timeliness requirement of real-time software. A software clock in Example 2.1 demonstrates this prevalent problem in platform-dependent design approaches.

Example 2.1 Consider a digital clock, which issues an action at the end of each second to count the passing of time. A clock modelled in CSDL is shown in Figure 2.7, where \( t_m \) is a variable used to count the passing of time. Ideally, the \( i \)-th action \( t_m = t_m + 1 \) should be issued at the \( i \)-th second of the hardware time since the start of the clock. However, due to the unbounded uncertainty of the timer in CSDL, it is observed in Figure 2.7 that the 31-st action \( t_m = t_m + 1 \) is performed at the 51-st (\( \text{currenttime-starttime} \)) second during a single sample run.

To avoid the unbounded interpretation of timing expressions, instead of using timers, some approaches advocate to using while loops to continuously check the system time and determine whether the time-out is due [20]. It is easy to see that by using this technique the computation time is exclusively taken up by a single process, which is only applicable in sequential software. The approach is incapable of dealing with systems containing parallel processes.

Ineffective monitoring mechanism

Two approaches are commonly used to monitor actions in a software system: breakpoint based approaches and monitoring based approaches [101]. Breakpoint based approaches allow designers to set breakpoints where the software execution stops, giving them the opportunity to observe the state of the executing code. By analyzing the state changes between the breakpoints, designers may localize the fault to a
particular code block and figure out the reason causing the error. On the other hand, monitoring-based approaches only record the state of interest without stopping the execution of the system. The captured information is often transferred to a separated processing unit and analyzed off-line.

In both approaches, additional monitoring code has to be added for monitoring purposes. This code can change the timing behavior of the software, especially the relative timing between processes, since the execution of the code always takes a certain amount of hardware time. As a result, the monitoring code disturbs the localization of design errors and may introduce new errors. This problem is referred to as the Heisenberg uncertainty principle in testing in [101].

**Ineffective model transformation**

The timing semantics of modelling languages in platform-dependent approaches is often too ambiguous to support effective model transformation. This can be illustrated by Example 2.2.

**Example 2.2** *Two synchronized processes* $P$ and $Q$: Consider a simple real-time system (shown in Figure 2.8) consisting of two parallel processes $P$ and $Q$ ($P \parallel Q$), each of which comprises an iterative code segment involving timed actions. At the beginning of each iteration, $P$ and $Q$ synchronize with each other. Then process $P$ sets a timer with a 3-second delay and process $Q$ sets a timer with a 2.999-second delay. After the timer of $Q$ expires, $Q$
Design approaches for real-time software

Figure 2.8: A system with two parallel processes P and Q

sends a reply message (rpl\_sig) to P. For process P, there are two possibilities:

1. P receives the timer expiration message and outputs the message “wrong”;
2. P receives the reply message (rpl\_sig) from Q, resets its own timer and outputs the message “correct”.

Here, we use a graphical modeling language based on SDL-96 to describe the system (shown in Figure 2.8). In SDL-96, the timing semantics is given in such a way that each action takes an undefined amount of physical time [34] and the interpretation of timing expressions (such as timers) relies on an asynchronous timer mechanism provided by underlying platforms [59].

Suppose that two processes P and Q are designed separately, which is often the case in complex system design. Now, let us first look at the timing semantics of process Q depicted in Figure 2.9(a). In Figure 2.9, x is a clock used to express timing constraints on actions and x := 0 represents the setting of clock x to zero. Since the timing semantics of process Q is influenced by the underlying platform and other processes running on the same platform, it is too ambiguous to accurately illustrate this timing semantics by state diagrams. Figure 2.9(a) only shows a part of the semantics of Q, which is already sufficient to show the deficiencies of the platform-dependent semantics. Process Q first receives a syn\_sig message, which takes time duration t₁. Before the next statement (set(qtimer, now + 2.999)) is executed, the operating system might switch to other processes taking a total amount of time t₂, before it switches back to process Q. Then the timer is set and process Q is suspended (taking time t₃) to wait for the timer expiration message. Between the time that the timer expires and the time that process Q responds to the time\_out message, again the operating system might take a total amount of time t₄ for the execution of other processes.

Execution times t₁, t₃ and t₅ are neglectable w.r.t. most real-time properties of interest in a modern computing platform. In the case that process Q is the only active process running on the platform, t₂ and t₄ are zero. As a consequence, Figure 2.9(b) can be considered to
be a proper abstraction of process \( Q \). In a similar way, an abstraction for process \( P \) can be obtained, which is depicted in Figure 2.9(c). In design practice, it may be assumed that correctness is preserved between abstractions and refinements. That is, the integration of parallel processes can preserve the properties of the integration of their abstractions. Therefore, the integrated system (\( P \parallel Q \)) can be reasoned about through the abstractions. This would indicate that \( P \) should never output the "wrong" message when it is integrated with process \( Q \).

However, in certain circumstances, the platform-dependent semantics of both processes does allow process \( P \) to output the "wrong" message in the integrated system. For example, in Figure 2.9(a), when process \( Q \) is in state \( S_1 \), the underlying operating system may first activate process \( P \), then \( P \) sets the timer and suspends itself, after which the operating system switches back to \( Q \), which sets a timer with an expiration of 2.999 seconds. If one context switch, setting a timer, suspending a process together with other necessary scheduling execution take more than 0.001 seconds in total, the timer of process \( P \) might expire before that of process \( Q \). As a result, \( P \) outputs the "wrong" message. In a complex concurrent real-time (software) system, the cost can easily exceed 0.001 seconds due to frequent context switches between many processes.

From the above example, we can see that the abstraction of the integration of a set of components cannot always be correctly reasoned about from the abstraction of its components. To eliminate unexpected system behaviors, designers have to rely on ad-hoc ways to tune the behavior of each component, which often involve a tremendous number of design details of other components. As a result, the design process is often time consuming and prone to errors.

(Task-level) real-time scheduling is often adopted in practice to alleviate the problems mentioned above for platform-dependent design approaches.
**Task-level scheduling:** A system can be viewed as a set of concurrent tasks. A task-level scheduler is used to manage the activation and execution of tasks concurrently running in the system. It assigns the computation time by giving different priorities to tasks. In general, the task with a higher priority is scheduled before those with lower priorities. The goal of real-time scheduling is to devise a priority assignment scheme to ensure that every task can be accomplished in time, which can eliminate undesired interferences from other tasks, reducing the ambiguity of the timing semantics of each task. However, the task-level scheduling problem is computationally intractable in general, and task-level scheduling is only feasible for a certain class of applications. A further discussion of task-level scheduling is presented in Section 2.5.2.

### 2.4.2 Platform-independent design approaches

Contrary to platform-dependent design approaches, platform-independent design approaches often adopt a virtual timing concept, which is independent of any underlying execution platforms. In the following, we show that predictability can be well-supported by platform-independent design approaches during system modelling. However these approaches usually lack predictability support for system synthesis.

**Two-phase execution framework**

Design practice has shown that real-time system modelling is much more problematic than non real-time system modelling. This can be contributed to the fact that computation time is usually a limited and shared resource. For example, the semantics of the modelling language SDL-96 ensures that the untimed behavior of each individual process in a concurrent system remains unchanged during system integration. But the timing behavior of each individual process in the physical time domain can be interfered by other processes during the integration. Additional techniques are required to eliminate functional and timing inconsistencies caused by potential time resource conflicts during real-time system modelling.

A straightforward solution to the above problem is to assume that computation time resources are infinite, and consequently no time resource conflict exists anymore. This solution is also adopted by the two-phase execution framework (see Figure 2.10). In this framework, the state of a system can change either by asynchronously executing some actions such as communication and data computation without time passing (phase 1) or by letting time pass synchronously without any action being performed (phase 2). It is easy to see that time is not a limited resource for computations, because computations are timeless in this framework. As a consequence, the following benefits can be obtained in this framework.

- Concurrent processes do not compete for computation time with each other, and it is easier to keep their timing behaviors unchanged during system integration.

---

4The physical time can be the hardware time measured by a hardware clock.
• Monitoring code does not take up the computation time of the monitored system, and the original timing behavior of the system can remain unchanged during system analysis.

• Adding design details to or removing design details from a part of a system has no influence on the computation time of the other parts of the system, and model transformations can be performed locally.

• Last but not the least, the timing behavior of the model is not affected by techniques (such as caching and pipelining) adopted by the platform.

Obviously, the physical time cannot be adopted to represent time in this two-phase execution framework. Instead a system variable is usually used to represent time and is denoted as virtual time. Consequently, the timing behavior of the system is analyzed in the virtual time domain.

The two-phase execution framework offers many benefits to improve design predictability during system modelling, and it has been supported by many modelling languages. Typical examples in academic fields are timed automata [5], timed process algebra ATP [73] and real-time transition systems [39], all of which have been successfully applied to the modelling and the analysis of real-time systems. More examples with tool support in design fields are emerging, such as, SDL-2000 [90] supported by TAU G2 [3] and POOSL supported by SHESim [97, 100].

In the two-phase execution framework, the timing aspect of a system is treated orthogonally from the system functionality. Therefore it can be incorporated into different untimed semantic frameworks. For example, in the semantics of SDL-2000, this framework is combined with asynchronous communications to specify the timing behavior of a system, while in the semantics of POOSL, it is combined with synchronous communications. For detailed semantics of these modelling languages, readers are referred to [96, 97, 90].

The timing semantics of the design descriptions in Example 2.2 can also be given in platform-independent semantic frameworks. In these frameworks, $t_1$ till $t_5$ are all zero and we can consider the semantics for process $Q$ depicted in Figure 2.9(a) to be a proper abstraction of that in Figure 2.9(b), and the same holds for the abstraction of
process $P$ depicted in Figure 2.9(c). Consequently, the abstraction of the combined system $(P \parallel Q)$ can be captured by the combination of Figure 2.9(b) and Figure 2.9(c), in which process $P$ should never output the “wrong” message. We made the same model in TAU G2, and the behavior of the model $(P \parallel Q)$ is indeed as expected.

Ineffective system synthesis

Although most platform-independent approaches provide better support in their modelling languages for predictable design, bridging the large gap between semantics of these modelling languages and implementation languages is still not solved adequately (see Section 2.2.3, Implementation languages).

The automatic transformation of design models to realizations is a superseding technique to manual transformation, the latter of which is inefficient and prone to errors. In current practice, the automatic transformation lacks sufficient support to bridge the gap between the timing semantics of the modelling language and the implementation language. As a result, inconsistencies can be observed between the model and the realization. For example, actions are usually assumed to be instantaneous in the model, while they do take a certain amount of physical time in the realization. Without careful consideration of this semantic difference, the realization can exhibit faulty behavior. Although the model in Example 2.2 made in TAU G2 is proven to be correct, errors are observed in the automatically synthesized realization (see Figure 2.11).

The semantic gap is also observed by the accumulation of timing errors during execution. The accumulated timing errors can result in timeliness failures or even functional failures. This is demonstrated by the following two simple examples, which have been modelled in TAU G2 and synthesized automatically by the corresponding tool.

Example 2.3 Accumulated timing errors:
Consider a digital clock, the functionality of which is similar to that described in Example 2.1. The only difference is that its accuracy is one tenth of a second instead of one second.

This simple real-time clock system can be easily modelled in most platform-independent design frameworks, where each counting action is performed precisely at the expected time. We used TAU G2 to automatically transform the model into a realization. The result of the realization is shown in Figure 2.12, where the timing error of each counting action is calculated based on the deviation between its observation times in the realization and that in the model (i.e. the difference between the time points of the same action performed in the vir-
tual time domain and in the hardware time domain). To reduce the interference of additional observations, we sampled the timing error once every hundred counting actions. The small irregularity at the start of the curve is caused by the computational cost of the initialization of the realization. We can see from Figure 2.12 that the timing error is accumulated and unbounded during the execution of the realization.

Accumulated timing errors can also change the relative timing relation between concurrent processes and cause faulty behaviors. This is demonstrated by Example 2.4.

**Example 2.4 Incorrect functionality:**
Consider a controller for a flash board showing 4 consecutive letters “IEEE”, the functionality of which is described as follows. The four letters of the word are sequentially displayed on the board, and then wiped off altogether at the same time. The iteration continues unless being interrupted manually. One solution to designing the controller is to use three parallel processes. Process I emits letter I every 0.3 seconds, process E emits letter E every 0.1 seconds and process space issues four blank spaces every 0.3 seconds to erase the letters. The three processes start at 0.01,0.02 and 0.25 seconds respectively.

We made a model of the system using TAU G2, and the sequence diagram of the controller is shown in Figure 2.13. The relative timing relation between three concurrent processes ensures that the model behaves correctly w.r.t. the functionality requirement. However, in the realization generated by TAU G2, timing errors are introduced in each process and accumulated during their execution. These errors change the relative timing relation between the concurrent processes and result in unexpected outputs. Figure 2.14 gives a snapshot of the output of the realization. We can see that after some iterations, the timing errors of processes led to an incorrect output sequence.

Till now, we have investigated the merits and pitfalls of platform-dependent and platform-independent design approaches. The major drawbacks of platform dependent approaches are that they lack predictability support during model analysis,
Figure 2.13: The sequence diagram of the IEEE flashboard controller

Figure 2.14: The output of a realization of the IEEE flashboard controller
model transformation and model integration. Task-level scheduling can relieve the problem to a certain extent, but it is not a general solution due to its limitation, which is explained in the following section. In principle, platform-independent design approaches provide better support to model and analyze real-time software, but they run into trouble when constructing realizations from models. To solve this problem, a formal link should be established between the semantic domains of modelling languages and implementation languages. The next part of this thesis will be devoted to the investigation of the real-time property relations between timed systems. These relations serve as a theoretical basis for linking the timing semantics of models and realizations.

2.5 Timeliness considerations

Due to timeliness requirements, the correctness of real-time software depends not only on the sequence of actions during the execution, but also on the time at which these actions are performed. Platform-dependent and platform-independent approaches adopt different strategies to ensure that actions are performed at the correct time (interval). Most platform-dependent design approaches use task-level scheduling, while platform-independent design approaches use action-level scheduling. 

2.5.1 Orders of actions

A real-time system (a software system and its environment) is usually composed of a set of tasks. Each task consists of a set of actions performing internal computations and external communications. Accordingly, the system behavior can be formalized as a set of action sequences (or state sequences), each of which represents a possible execution of the system. To ensure the correctness of the system behavior, action sequences (state sequences) have to follow certain orders.

Causal order

Actions can be ordered causally in the sense that one action can be the “cause” of another action. Such an order is called a causal order. Causal orders help to determine which actions should happen before what others [26]. In software systems, causal orders can be defined in the following two ways.

- Causal orders within each task can be defined by the coming after relation between actions. For example, \( a_{i-1} \) always happens earlier than \( a_i \) in task \( A \) (depicted in Figure 2.15).

In certain cases, platform-independent approaches can also adopt task level scheduling. For example, in [9], the timing behavior of a system is analyzed in the framework of timed automata, where the platform constraints are not explicitly considered. This design approach can be categorized as platform-independent. However, the timing behavior of each task in the system is modelled by its WCET (worse case execution time) without considering its internal individual actions. Accordingly, the scheduling is carried out at the task level.
Causal orders between different tasks are often defined based on the communications between them. For example, in a synchronous communication model, task \( A \) and task \( B \) in Figure 2.15 synchronize through action \( a_i \) and \( b_i \). Since \( a_i \) and \( b_i \) happen simultaneously, it is easy to see that \( a_{i-1} \) happens earlier than \( b_{i+1} \), if causal orders within each task are also considered. Similarly, \( b_{i-1} \) happens earlier than \( a_{i+1} \). In an asynchronous communication model, if task \( A \) sends a message (\( a_i \)) and task \( B \) receives a message (\( b_i \)), then \( a_i \) always happens earlier than \( b_i \). It is easy to see that \( a_{i-1} \) happens earlier than \( b_{i+1} \), but no causal order can be derived between \( b_{i-1} \) and \( a_{i+1} \) in this case.

Qualitative real-time properties (such as safety and liveness properties) of a system are directly influenced by causal orders of actions, which are partially determined by the interactions (communications) between tasks. Intricate interactions in complex software often result in tremendous number of possible action sequences, which complicates the reasoning about properties.

**Temporal order**

A temporal order refers to the order of actions resulting from the mapping of the causal order of actions to their time of occurrence along a timeline [26]. As shown in Figure 2.15, actions of task \( A \) and task \( B \) are mapped to a timeline and the quantitative timing order between actions is specified. For example, \( a_{i-1} \) happens \( t_6 - t_3 \) seconds earlier than \( b_{i+1} \). It is easy to see that a causal order always implies a temporal order, but the reverse does not always hold. For example, in Figure 2.15 \( a_{i-2} \) happens before \( a_{i-1} \) in the causal order, which implies that this is still true in the temporal order. However, \( b_{i-1} \) happening before \( a_{i-1} \) in the temporal order does not imply that a causal order exists between these actions.

Quantitative real-time properties (such as deadlines) are determined by the temporal order of actions in the system. In real-time system design, a proper mechanism should be provided to map causal orders to desired temporal orders. In the sequel of this section, two different mapping strategies are investigated and their merits and pitfalls are discussed.
2.5.2 Task-level and action-level scheduling

The mapping from causal orders to temporal orders can be achieved by real-time scheduling. Based on the scheduling granularity, real-time scheduling can be divided into two categories, task-level and action-level scheduling.

Task-level scheduling

Task-level scheduling is one of the most important analytical methods in real-time system design. It can help designers to specify and analyze the timing behavior of design solutions before they are actually implemented on the target platform. In task-level scheduling, each task is assigned a set of timing parameters, such as the worse-case execution time (WCET), the priority and the invoking frequency [21]. The timing behavior of the system can be predicted by the analysis of these parameters. The goal of task-level scheduling is to devise a priority assignment scheme to ensure that every task can be accomplished in time. In practice, various scheduling schemes have been devised for different real-time applications. However, task-level scheduling is not a complete solution to the design of complex real-time software, due to the following facts:

1. **Invalid scheduling assumptions:** It is well-known that the general (task-level) scheduling problem is NP-complete and computationally intractable [21]. In order to reduce the complexity of the design so that a feasible priority assignment scheme can be devised, designers have to rely on a set of assumptions, such as fixed execution time, independent tasks, fixed priorities, no sharing of resource and periodically triggered tasks. These assumptions are only valid for a certain class of real-time applications and cannot be taken for granted in the design of a complex real-time software system.

2. **Overly complex scheduling scheme:** A complex real-time software system might consist of tens or even hundreds of tasks. For example, a multi-cast internet router can consist of more than a dozen interwoven communication protocols (such as TCP, UDP, MSDP, PIM-SM and MBGP) [102], each of which involves a number of real-time tasks. For example MBGP (Multi-Protocol extension of the Border Gateway Protocol) needs to update in real-time routing tables, maintain link state information and respond to errors. The complexity of devising a priority assignment scheme for such a system might be as complex as the design of the whole system. Furthermore, the priority assignment scheme can be easily affected by the composition of tasks and the computational time of each task. Modifying, removing or adding a task might lead to the redesign of the complex priority assignment scheme.

3. **Inefficient utilization of the timing resource:** The computation time is a crucial and limited resource in a real-time system. This requires that an ideal scheduler should introduce negligible overhead. However, due to the existence of the priority inversion problem ⁶ (discussed in the following), sophisticated priority assignment schemes have to be devised to avoid this problem. This often

---

⁶The priority inversion problem was first recognized in 1980s. It has become well-known since the failure of the Mars Pathfinder in 1997.
results in a significant overhead in real-time system execution. Furthermore, priority assignment schemes are usually devised according to the worst-case execution time of each task, which is often much longer than its actual execution time. Hence, the computation time cannot be used efficiently, causing difficulties to find a correct design solution with limited computational resources [62].

The key problem in task-level scheduling can be attributed to the potential contradiction between priority assignment schemes and causal orders derived from communications between tasks, which is directly or indirectly related to the above three problems. This potential contradiction can be illustrated by the priority inversion problem. A typical scenario of priority inversion is depicted in Figure 2.16, where a system is shown consisting of three parallel tasks. Task A has the highest priority, task B has a medium priority, and task C has the lowest priority. Assume that task A needs to synchronize with task C at action a₂ and action c₂. Task A needs to wait at a₂ until task C reaches c₂. However, since task B has a higher priority than task C, the execution of task C may be blocked by that of task B, which implies that the execution of task A is also blocked by that of task B. In this case, a task with lower priority (task B) can take precedence over a task with higher priority (task A). The order of this execution is often considered to be abnormal w.r.t. the priority assignment and is referred to as the priority inversion problem.

Now, let us have a look at the potential contradiction between priority assignment schemes and causal orders derived from communications between tasks in the priority inversion problem. According to the priority assignment, the system is assumed to have the following action order when all the three tasks are free to be scheduled at the same time.

\[ a_1 \prec a_2 \prec a_3 \prec b_1 \prec b_2 \prec b_3 \prec c_1 \prec c_2 \prec c_3. \]

However, in the case that A needs to receive a message from C (either by a synchronous or asynchronous communication) at action a₂, then c₂ should happen before a₃. Based on the priority assignment, actions b₁, b₂ and b₃ should happen earlier than c₂, therefore they should happen earlier than a₃. But at the same time, based on the priority assignment, they should also happen later than a₃. Such a contradiction leads to an abnormal situation in the priority inversion problem. To avoid the priority inversion problem, more complicated priority assignment algorithms are required, such as priority inheritance and priority ceiling protocols [81]. However, these additional mechanisms often complicate the analysis of the timing behavior of the system.
Action-level scheduling

Action-level scheduling has been adopted in existing formalizations for real-time systems, such as timed CCS [33, 104], where the invoking time of each action is specified. In general, action-level scheduling provides fine-grained scheduling schemes and gives more freedom to designers to tune the timing behavior of the system. However, it also complicates the timing analysis of the system. On one hand, it requires designers to have sophisticated knowledge of formal theories, because the timing analysis of the system often relies on formal verification techniques, such as model-checking and theorem proving. On the other hand, it is the common belief that action-level scheduling is only effective for moderate scale real-time systems, due to the state-space explosion problem in formal verification techniques. Currently, many research activities are being carried out to alleviate the problem. Some techniques have been developed, such as symbolic representations supporting the compact encoding of system states [19], on-the-fly algorithms generating system states on the fly [17] and redundancy elimination removing redundant system states [24].

In summary, task-level scheduling maps a set of actions to the timeline based on the priority of each task. For certain real-time applications (especially those consisting of independent tasks), this mapping can simplify the timing analysis of the system and promote the likelihood of success of the design. However, the priority assignment might interfere with the causal order between concurrent tasks (shown in the priority inversion problem). On the other hand, action-level scheduling maps actions to the timeline based on a semantic framework, such as the two-phase execution framework. Due to the finer granularity of the action-level scheduling, it is more effective than the task-level scheduling during design of interaction-intensive real-time applications. In Part III of this thesis, we adopt an action-level scheduling, which is based on the two-phase execution framework, to design real-time systems.

2.6 Summary

In this chapter, we discussed predictability support for real-time software design.

- We defined predictability in a multi-stage system design. The importance of predictability for system design was illustrated on the basis of correctness-preservation and compositionality, which are two inter-dependent forms of predictability.

- We analyzed basic requirements for the semantics of design languages (requirement languages, modelling languages and implementation languages) to obtain predictable design. In short, these design languages should have adequate expressive power to specify the timing behavior of a software system. Due to the potential complexity of real-time software, additional requirements are put on the semantics of modelling languages. First, it should facilitate effective timing analysis of models, where the analysis (monitoring) code should not affect the original timing behavior of software. Second, it should assist effective transformation between different abstraction levels. To this end, composability should be supported by the semantics, which allows model trans-
formations to be carried out locally and the correctness of the model to be verified locally. Finally, a formal link should be established between the semantics of both modelling and implementation languages, in order to achieve correctness-preserving software synthesis.

- The timing concept adopted by modelling languages has a direct impact on their satisfaction of the above mentioned requirements. This is illustrated by the comparison of platform-dependent approaches (adopting the hardware time concept) and platform-independent approaches (adopting the virtual time concept). Platform-dependent design approaches lack sufficient predictability support for modelling and analysis of the timing behavior of software, while platform-independent design approaches usually lack a correctness-preserving transformation between models and realizations.

- The correctness of real-time software depends not only on the order of actions, but also on the quantitative time distance between actions. From this point of view, we investigate potential limitations of both task-level scheduling (usually applied in platform-dependent design approaches) and action-level scheduling (usually applied in platform-independent design approaches). In task-level scheduling, there is a potential contradiction between the action order defined by a priority assignment and the action order derived from communications between tasks. On the contrary, action-level scheduling may suffer from the state-space explosion problem during timing analysis of many large-scale systems. Action-level scheduling can provide fine-grained scheduling schemes, which is more effective than task-level scheduling during design of interaction-intensive real-time applications.
Part II

Real-time property-preservation between timed systems

In Part I of this thesis, we classified existing real-time system design approaches into two categories (platform-dependent and platform-independent approaches). Furthermore, we investigated the merits and pitfalls of these approaches w.r.t. their support for design predictability. In short, the semantics of modelling languages adopted by platform-dependent approaches often lack sufficient support for specifying and analyzing concurrent timing behavior. In addition, design predictability is not well-supported during system modelling. The platform-independent approaches offer solutions to the problems encountered by platform-dependent approaches. However, they often fail to guarantee the (timing) behavior consistency between a model and its realization, due to the large gap between the semantics of their modelling and implementation languages.

In platform-independent design approaches, it is difficult to ensure that corresponding actions in a model and in its realization are executed at exactly the same time. In other words, timing inconsistencies always exist between the model and the realization. As a consequence, properties verified in the model may not hold in the realization.

In this part of the thesis, we quantitatively measure the timing inconsistencies between a model and its realization by introducing two proximity measures between timed systems. Based on these measures, property relations can be established between timed systems (e.g. a model and its realization). This can be illustrated by
the diagram shown in Figure 2.17. In this diagram, timed system $S_2$ is “close” to $S_1$ under a certain proximity measure. If it can be verified that $S_1$ satisfies a (quantitative or qualitative) real-time property $P$, we can predict a corresponding property satisfied by $S_2$. These property relations between timed systems can serve as a theoretical basis for automatic and correctness-preserving system synthesis in platform-independent design, which will be illustrated in Part III of this thesis.

![Diagram of property preservation](image)

Figure 2.17: The objective of real-time property-preservation

In this part of the thesis, the diagram in Figure 2.17 will be completed based on different proximity measures between timing behaviors. Chapter 3 introduces some notations and preliminaries for later discussions and proofs. Chapter 4 gives two proximity measures between timing behaviors, which are based on absolute timing differences and relative timing differences respectively. Chapter 5 discusses weakening relations between real-time properties. In Chapter 6, we define a class of functions called “tube functions” between two approximate time interval sequences. The properties of tube functions facilitate the proof of property-preservation between timed systems. In Chapter 7, we prove real-time property-preservation between timed systems for both the absolute timing difference case and the relative timing difference case. Furthermore, we show that the same property-preservation results hold for both sequential and concurrent timed systems.
Chapter 3

Preliminaries

In this chapter, we give a brief introduction of the mathematical framework employed in this part. A graph structure with various annotations for describing the timing behavior of a system is presented in Section 3.1. Temporal logics, which are often used to express real-time properties, are introduced in Section 3.2. Based on the notions presented in this chapter, formalizations for sequential and concurrent timing behaviors and real-time properties will be given in the following chapters, which serve as the basis for the discussion and proof of real-time property-preservation between timed systems.

3.1 Representations of system behaviors

A computing system can be considered as a discrete-event system, the behavior of which is formed by the asynchronous occurrence of discrete events (or actions). A common representation of a discrete event system is a graph structure, where the nodes of the graph represent system states and the arcs represent system actions (see Figure 3.1(a)). The occurrence of an action triggers a transition from one state to another. A graph structure defines a set of traces, each of which represents an execution (or a run) of the system (see Figure 3.1(b)). The nodes and arcs of a graph structure are often annotated with more specific information, in order to facilitate the analysis of the system behavior [68]. In the following, the graph annotations of untimed behaviors are first presented, which can be further extended with timing constraints to represent timing behaviors. Based on these graph annotations, we give in the next chapter the formalizations of timing behaviors for both sequential and concurrent systems.

3.1.1 Graph representations of untimed behaviors

Two ways exist to annotate a graph structure for the untimed behavior of a system. One is to annotate the nodes (observable states) with atomic propositions. The other is to annotate the arcs (state transitions) with actions. Both annotations can be ap-
plied to the traces defined by a graph structure. However, these annotations lead to different formalizations to describe untimed behaviors. A graph structure with node annotations is often formalized as a Kripke structure \[58\], and a graph structure with arc annotations is formalized as a labelled transition system \[66\]. The definition of both formalizations can be found in literature (e.g. \[71, 68\]). Example 3.1 illustrates how these annotations can be used to represent the untimed behavior of a system.

**Example 3.1** Consider a coffee machine which only accepts 5-cent coins. After putting a 5-cent coin, the customer can get a small cup of coffee by pressing the button, or he/she can put another 5-cent coin. In the latter case, he/she can get a medium cup of coffee by pressing the button, or put another coin again to get a large cup of coffee.

**Node annotation** Let \( P \) be a set of atomic propositions. Subsets of \( P \) denote observable states of a system. Proposition set \( P \) for the coffee machine in Example 3.1 consists of three propositions \( S, M \) and \( L \), which have the following meaning:

- \( S \): The coffee machine is ready to produce a small cup of coffee.
- \( M \): The coffee machine is ready to produce a medium cup of coffee.
- \( L \): The coffee machine is ready to produce a large cup of coffee.

The behavior of the coffee machine can be depicted by the graph structure shown in Figure 3.2(a), where each state is a (possibly empty) set consisting of atomic propositions that hold in that state. Correspondingly, Figure 3.2(b) shows the beginning of an infinite (state) trace defined by Figure 3.2(a).
Representations of system behaviors

Figure 3.3: The graph structure of a coffee machine with arc annotations

Figure 3.4: The graph structure of a coffee machine with both node and arc annotations

Arc annotation Let \( E \) be a set of actions. Each action triggers a state change. An example of action set \( E \) for the coffee machine in Example 3.1 is \( \{ \text{Put}_5c, \text{PushBtn} \} \), the elements of which have the following meaning:

- \( \text{Put}_5c \): Put in 5 cents.
- \( \text{PushBtn} \): Push the button.

Using arc annotations, the untimed behavior of the coffee machine and one of its (action) traces are shown in Figures 3.3(a) and 3.3(b) respectively.

Intuitively speaking, the node annotation is more suitable for expressing the static properties of a system and the arc annotation is more suitable for describing the dynamics of a system. For example, the coffee machine system in Example 3.1 is expressed more naturally by the node annotation as depicted in Figure 3.2(a). Another example is a mutually exclusive lock scheme, in which a lock can be acquired and released by different processes. The lock scheme can be expressed more naturally by lock and unlock actions. There are various ways to encode a node annotation into an arc annotation and vice versa [71]. In principle, both annotations can provide sufficient information for expressing the untimed behavior of a system. In practice, developers can choose any one of them according to specific needs. We can also use both annotations on the same graph structure and its corresponding traces for representation convenience, as shown in Figure 3.4 for the coffee machine.
3.1.2 Graph representations of timing behaviors

The graph structure (and its traces) introduced above specifies a sequence of states (or actions) but not the actual time at which states are observed (or actions occur). To investigate the timing behavior of a system based on a graph structure (or its traces), timing constraints have to be attached to each node (or arc) in the graph structure. In general, timing constraints attached to the graph structure are treated as an orthogonal aspect to the node (or arc) annotation. In principle, depending on the choice of annotations (on nodes, arcs, or both) and the placement of timing information (on nodes, arcs, or both), many possible ways exist to express the timing behavior of a system. Typical examples are: timed Muller automata [5], which annotate arcs and put timing constraints on arcs; timed automata [7], which annotate nodes and put timing constraints on nodes, and timed labelled transition systems, which consider timing progress as special actions and annotate arcs. An example of timed labelled transition systems is timed CCS.

In the following, a graph structure for representing the timing behavior of a system is illustrated by an example of an intelligent light controller, where mutually exclusive nodes are annotated with atomic propositions and arcs are labelled with timing constraints.

Example 3.2 Consider an intelligent light controller, which can adjust light intensity according to different input action sequences. If a click action occurs at the initial state, the controller goes into a temporary state and a timer is activated. If a second click occurs within 2 seconds, the intensity of the light is set to high. On the other hand, if no click occurs within 2 seconds after the first click, the intensity of the light is set to normal. When the light is on (with either normal or high intensity), another click sets the light off.

An annotation for timing behaviors Let \( P \) be a set of atomic propositions, and let \( C \) be a set of clocks, which are associated to the arcs of a graph structure to specify timing constraints on a system. During a transition from one observable state to another, some clocks in \( C \) might be reset. In Example 3.2, \( P \) is defined as the set \{Wait, Normal, High\} and \( C \) only contains one clock \( x \). The propositions in \( P \) have the following meaning:

- \textbf{Wait}: The light intensity is ready to be increased.
- \textbf{Normal}: The intensity of the light is normal.
- \textbf{High}: The intensity of the light is high.

The timing behavior of the intelligent light controller in Example 3.2 is shown in Figure 3.5(a). The system has four different observable states and one clock \( x \). It starts at observable state \( \emptyset \) where the light is off. As soon as one click occurs, clock \( x \) is reset and the system enters a temporary observable state \{Wait\}. Clock \( x \) is used to measure the elapsed time since the first click. If a second click occurs within 2 seconds, the system gives high intensity light. Otherwise, the system gives normal intensity light. In both observable states \{Normal\} and \{High\}, if a click occurs, the system goes into state \( \emptyset \). An execution of the system can be represented by a trace of pairs, where each pair consists of an observable state and its duration interval. Figure 3.5(b) gives an example of such a trace.
3.2 Representations of real-time properties

In system design, a target system should possess certain desired properties to meet the requirements. For example, a traffic light controller should have the following properties: (1) it should guarantee that no collision ever happens if the traffic rules are obeyed, (2) each traffic direction should eventually be granted access, and (3) different directions should be treated fairly. The correctness of a target system largely depends on whether these system properties are satisfied. In formal verification (especially model-checking), these properties are often formalized and analyzed within certain mathematical frameworks. Temporal logics are a class of frameworks widely applied in the formalization of qualitative and quantitative real-time properties (also called temporal properties in literature) [13]. In this section, we give an informal introduction of the application of temporal logics for expressing real-time properties.

3.2.1 Logics for qualitative real-time properties

(Traditional) temporal logics cover a family of various logics. In computer science, temporal logics are classified into linear-time (such as PLTL and LTL [32, 52]) and branching-time (temporal) logics (such as CTL [29] and CTL* [23]). The classification is based on the trace structures on which these temporal logics are interpreted.

Linear-time logics are interpreted over linear traces, such as state traces or action traces mentioned in Section 3.1. A typical example of linear-time logics is LTL (Linear Temporal Logic), which is interpreted over state traces. Its formulas are constructed by the following syntax:

$$\varphi ::= p \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \text{O} \varphi \mid \varphi_1 U \varphi_2$$

Intuitively, if a state trace satisfies atomic proposition $p$, then it is required that $p$ is observed at its first state. The interpretation of $\neg$ ("not") and $\lor$ ("or") is the same as that used by most logics. Instead of giving a precise interpretation of basic temporal operators $\text{O} \varphi$ ("next $\varphi$") and $\varphi U \psi$ ("$\varphi$ until $\psi$"), we illustrate some of them visually in Figure 3.6. Next to the above basic syntax structures, several commonly used structures are given in the following example.
Preliminaries

\begin{figure}[h]
\begin{center}
\begin{tikzpicture}
  \node[state, initial, accepting] (1) {$\phi$};
  \node[state] (2) [below of=1] {$\phi$};
  \node[state] (3) [below of=2] {$\phi$};
  \node[state] (4) [below of=3] {$\phi$};
  \node[state] (5) [below of=4] {$\phi$};
  \node[accepting] (6) [below of=5] {$\cdots$};
  \draw (1) -- (2); \draw (2) -- (3); \draw (3) -- (4); \draw (4) -- (5); \draw (5) -- (6);
\end{tikzpicture}
\end{center}
\caption{Illustration of LTL operators}
\end{figure}

\begin{figure}[h]
\begin{center}
\begin{tikzpicture}
  \node[state] (1) {$\emptyset$};
  \node[state, accepting] (2) [below of=1] {$S$};
  \node[state] (3) [below of=2] {$\emptyset$};
  \node[state] (4) [below of=3] {$\emptyset$};
  \node[state] (5) [below of=4] {$\emptyset$};
  \node[state] (6) [below of=5] {$\emptyset$};
  \node[accepting] (7) [below of=6] {$L$};
  \draw (1) -- (2); \draw (2) -- (3); \draw (3) -- (4); \draw (4) -- (5); \draw (5) -- (6); \draw (6) -- (7);
\end{tikzpicture}
\end{center}
\caption{Computation trees}
\end{figure}

\textbf{Example 3.3} Commonly used syntax structures:

- $\varphi_1 \land \varphi_2 \equiv \neg(\neg \varphi_1 \lor \neg \varphi_2)$;
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg \varphi_1 \lor \varphi_2$;
- $true \equiv p \lor \neg p$;
- $\Diamond \varphi \equiv true \lor \varphi$ ("eventually $\varphi$");
- $\Box \varphi \equiv \neg \Diamond \neg \varphi$ ("always $\varphi$").

Most qualitative properties can be formally expressed based on LTL syntax. For example, a response property that "every stimulus req is always followed by response ack", can be defined as $\Box(req \rightarrow \Diamond ack)$. This indicates that in any possible state trace of a system, whenever req is observed in state $s$, ack will be observed in another state later than $s$.

Branching-time logics are interpreted over tree-like state (or action) traces. For example, the untimed behavior of the coffee machine system given in Example 3.1 can be expressed by a computation tree, which is formed by unrolling its graph structure.
Representations of real-time properties

(as shown in Figure 3.7(a)). In contrast to linear-time logics, branching-time logics incorporate path (trace) quantifiers into their syntax, such as $A$ (all paths) and $E$ (for some path) in CTL. Figure 3.7(b) gives an interpretation of formula $A \bigotimes \varphi$ on a tree structure. Readers are referred to [7] for a more comprehensive overview of temporal logics that are used for specifying reactive systems.

### 3.2.2 Logics for quantitative real-time properties

When timeliness becomes a major concern, it is necessary to incorporate quantitative timing constraints into temporal logics to express desired quantitative real-time properties. A common way to do so is to extend qualitative temporal logics by attaching time-bounds to their temporal operators. Both linear temporal logics and branching temporal logics can be extended with time bounds. Typical examples include metric temporal logic MITL (a time-bounded extension of LTL) [7] and branching time-bounded temporal logic RTCTL (real-time CTL) [30]. More detailed surveys of time-bounded extensions of temporal logics can be found in [7, 8].

To better understand how time-bounds are incorporated into logics to express real-time properties, here we briefly introduce a simple, yet widely used temporal logic, MTL [57]. It has the capability of specifying quantitative real-time properties. Its syntax is given as follows:

$$\varphi ::= p \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \varphi_1 \mathsf{U}_I \varphi_2,$$

where $p$ is an atomic proposition and time-bound $I$ is a non-singular interval with integer end-points. A $+$ operator is often defined for time-bound $I$. $I + t$ represents interval $\{t + t_1 \mid t_1 \in I\}$. Time-bounded formula $\varphi_1 \mathsf{U}_I \varphi_2$ holds at time $t$ of a timed state trace iff there is a $t' \in I + t$, where $\varphi_2$ holds and $\varphi_1$ holds in time interval $[t, t')$. Two commonly used temporal operators $\diamond_I \varphi$ (time bounded eventually) and $\Box_I \varphi$ (time bounded always) can be expressed in terms of the until operator $\mathsf{U}_I$ in a similar way as shown in Example 3.3.

#### Example 3.4 Specification of a login program

Login programs are widely embedded in various security applications. In these programs, multiple attempts are usually allowed for password authentication. For security reasons, if the first password attempt fails, a minimal waiting time ($t_1$) is required before a second attempt is allowed. At the same time, the waiting time between the two attempts should not be too long to cause inconvenience for legal users. Correspondingly, if the first password attempt fails, a login program should prompt to users again within $t_2$ seconds, where $t_2 \geq t_1$.

Now we define two atomic propositions as follows.

- $p$: the login program is ready to receive a password.
- $q$: the login program receives an incorrect password.

A correct login program should satisfy the following property:

$$\Box_{[0, \infty)} (q \rightarrow (\Box_{[0,t_1]} \neg p \land \Diamond_{[t_1, t_2]} p)).$$
3.3 Summary

In this chapter, we have given an introduction to various decorations put on a graph structure to express the functional and timing behavior of a system. All of these representations can be interpreted as a set of timed state (or action) traces, which is employed in this thesis to formalize system behaviors. The detailed formalization of the timing behavior can be found in the next chapter.

In the second part of this chapter, we have given a brief overview of temporal logics and illustrated their application to express (quantitative and qualitative) real-time properties. In Chapter 5, we further discuss the linear-time (temporal) logic $MTL$, which is used in this thesis to specify real-time properties.
Chapter 4

Proximity measures between timing behaviors

To investigate real-time property relations between timed systems, we need to formalize their timing behaviors, and define the proximity (or nearness) between timing behaviors. These are discussed in this chapter. In Section 4.1, the timing behavior of a system is represented by a set of timed state sequences. Two formalizations are given for sequential and concurrent systems respectively. In Section 4.2, we define two displacement functions to measure the nearness between timed state sequences. One displacement function measures the nearness between timed state sequences based on the absolute timing difference between the observation times of corresponding state transitions. The other displacement function is based on the relative timing difference between the durations of corresponding observable states. In addition, these displacement functions are defined for both sequential and concurrent cases on the basis of corresponding formalizations of timed state sequences. Each displacement function gives a different nearness measure between timed state sequences, which can lead to different property-preservation results in Chapter 7. In Section 4.3, we discuss whether the two displacement functions define an "equivalent" nearness between timed state sequences. Sections 4.4 and 4.5 discuss related work and summarize this chapter.

4.1 Formalization of timed systems

Based on decorations put on the graph structures (see Section 3.1.2, Graph representations of timing behaviors), various behavior formalizations of timed systems have been proposed in literature, such as timed automata and timed labelled transition systems. The timing behavior expressed by these formal frameworks can be interpreted by (or encoded into) a unified semantic framework of state traces, which is employed in this thesis. In the sequel, we present the formalizations of timing behaviors for sequential and concurrent systems respectively.
4.1.1 Sequential timed systems

Basic concepts

The timing behavior of a sequential system can be formalized as a set of timed state sequences, whose related definitions are given as follows.

**Proposition set** \( \text{Prop} \) is a set of atomic propositions. An observable state of a system can be interpreted by a subset of \( \text{Prop} \) which contains all propositions that hold in that state.

**State sequence** \( \delta = \delta_0 \delta_1 \delta_2 \ldots \) over proposition set \( \text{Prop} \) is a finite or countably infinite sequence of states, where \( \delta_i \subseteq \text{Prop} \) for \( i \in \mathbb{N} \). We use \( \delta(i) \) to denote observable state \( \delta_i \) in the sequence, and use \( n(\delta) \) to represent the length (number of states) of the sequence.

**(Sequential) time interval** \( I \) is a convex set of time points over a certain time domain \( T \). Commonly used time domains are \( \mathbb{R}^\geq 0 \), \( \mathbb{Q}^\geq 0 \) and \( \mathbb{N}^2 \). In this thesis, time intervals are defined over \( \mathbb{R}^\geq 0 \). \( I \) has one of the following forms: \([a, b)\) or \([a, \infty)\), where \( a < b \) and \( a, b \in \mathbb{R}^\geq 0 \). The lower (upper) bound of the interval is represented by \( l(I) \) (\( r(I) \)). \( |I| = r(I) - l(I) \) denotes the length of a bounded (\( b < \infty \)) time interval \( I \). If time interval \( I \) is unbounded, we write \( |I| = \infty \).

**Sequential) time interval sequence** \( I = I_0 I_1 I_2 \ldots \) is a finite or infinite sequence of time intervals. The length (number of time intervals) of sequence \( I \) is represented by \( n(I) \), which can be finite or countable infinite. \( I \) has the following properties:

- \( I \) is **adjacent**, i.e. \( r(I_{i-1}) = l(I_i) \) for every \( 0 < i < n(I) \).
- \( I \) is **diverging**, i.e. for any \( t \geq l(I_0) \), there exists some \( i \in \mathbb{N} \), such that \( t \in I_i \). Hence, a finite time interval sequence always ends with an unbounded interval.
- \( I \) is **strictly monotonic**, i.e. \( l(I_i) < l(I_{i+1}) \) for every \( i < n(I) \). This can be easily concluded from the adjacency property and the definition of time interval.

**(Sequential) timed state sequence** \( \tau \) is a sequence of duration states, where a time interval \( I_i \) is attached to each state \( \delta_i \) to represent its duration. An example of a sequential timed state sequence \( \tau \) is:

\[
(\delta_0, I_0), (\delta_1, I_1), (\delta_2, I_2), \ldots
\]

All \( \delta_i \) together form a state sequence \( \delta \), and all \( I_i \) together form a time interval sequence \( I \). It is easy to see that the lengths of \( \delta \) and \( I \) are identical (i.e. \( n(\delta) = n(I) \)).

In the sequel, we use a sequence of pairs and a pair of sequences interchangeably for convenience to represent a timed state sequence. We call \( |I_i| \) the **time-duration** of state.

---

1 We use symbols \( N, Q, R \) to represent the set of natural numbers, rational numbers and real numbers respectively. Furthermore, \( R^\geq t \) represents the set of reals which are no less than \( t \).

2 In this thesis, we use \( T \) to denote a space, which consists of a set of elements \( T \) and a structure \( S \) over its elements. Structure \( S \) specifies the nearness relation between the elements. A typical example of \( S \) is the metric function, which defines the distance between elements. Furthermore, \( R^\geq 0 \) indicates that its element set is \( R^\geq 0 \).
Formalization of timed systems

\( \delta_i \) and \( l(I_i) \) the time-stamp of the state transition from \( \delta_{i-1} \) to \( \delta_i \). We assume that the initial state of the sequence is also triggered by a state transition. \( \tau = (\delta, T) \) is a timed state sequence over proposition set \( \text{Prop} \), if \( \delta \) is a state sequence over \( \text{Prop} \). In the sequel, we denote \( S^S_{\text{Prop}} \) as the set of all timed state sequences in sequential systems over \( \text{Prop} \).

(Sequential) timed system \( S \) is a set of timed state sequences, each of which represents a possible timed execution.

Due to the (well-)known problem of \textit{until} operator in dense-time temporal logics (see Section 5.1.2), we require that time intervals are left-closed when they are used to express the duration of a state.

In this formalization framework, it is assumed that the duration of a state (i.e. the length of a time interval) is always larger than zero. Based on this assumption, we exploit real-time property preservation between sequential timed systems, which is much simpler than the concurrent case. The major concepts of real-time property-preservation are introduced in a relative simple mathematical framework. The use of instantaneous states (which durations are zero) makes the proof of property-preservation between timed systems much more complex, and is addressed separately in the concurrent case.

Normalization of timed state sequences

Recall that a timed state sequence \( \tau \) consists of two sequences, state sequence \( \delta \) and time interval sequence \( I \). The relation between these two sequences can be viewed as a function from time domain \( R^{\geq l(I_0)} \) to state space \( 2^{\text{Prop}} \) [6, 79]. By this function, every time instant \( t \) along the time line is assigned a state \( \tau(t) \). For each \( t \in I, \) we have that \( \tau(t) = \delta_i \). From this point of view, timed state sequences with different state sequences and/or time interval sequences may imply the same function from \( R^{\geq l(I_0)} \) to set \( \Delta = \{ \delta_i \mid i < n(T) \} \).

Example 4.1 Define two timed state sequences \( \tau \) and \( \tau' \) as follows. \( \tau = (\delta, T) \) where for all \( i \in N, \delta(i) = \delta_i \) and \( T(i) = [2i, 2i + 2] \). \( \tau' = (\delta', T') \) where for all \( i \in N, \delta'(2i) = \delta_i \) and \( T'(i) = [i, i + 1] \). It is easy to see that for any \( t \in R^{\geq l(I_0)} \), \( \tau(t) = \tau'(t) \).

Definition 4.1 Two timed state sequences \( \tau = (\delta, T) \) and \( \tau' = (\delta', T') \) are equivalent (\( \tau \equiv \tau' \)) iff \( l(I_0) = l(I'_0) \) and for all \( t \in R^{\geq l(I_0)} \), \( \tau(t) = \tau'(t) \).

Any timed state sequence \( \tau \) can be normalized to an equivalent timed state sequence \( \tau^* \) of a particular form, which is called the normal form of \( \tau \). The normalization of \( \tau \) can be performed by the following operations. Along the state sequence of \( \tau \), successive identical states are replaced by one single state, and their corresponding time intervals are also merged into one single time interval. It can be shown that \( \tau \equiv \tau^* \) iff they have the same normal form. In Example 4.1, if no two subsequent states \( \delta_i \) and \( \delta_{i+1} \) \( (i \in N) \) are identical, we can conclude that \( \tau \) is in normal form. We assume in the sequel that sequential timed state sequences are in normal form, unless explicitly stated otherwise.
4.1.2 Concurrent timed systems

The main difference between the formalizations of concurrent timed systems and sequential timed systems is that a singular time interval (e.g., \([t, t]\)) is allowed to be attached to a state in the concurrent case. That is, the duration of a state can be zero in the concurrent case. As a consequence, a finite state sequence can be observed at one single time moment. This relaxation in definition enables timed state sequences to express concurrent system activities in an interleaved way.

Basic concepts

We first introduce some related basic concepts for the formalization of concurrent timed systems. Most of these concepts are extended versions of those introduced in the previous subsection.

The definition of **Propositions** and **State sequences** are identical to those in the sequential case.

(Concurrent) **time intervals**: Different from the sequential case, concurrent time interval \(I\) has one of the following forms: \([a, a]\), \([a, b]\) and \([a, \infty)\), where \(a < b\). A time interval is **singular** iff it takes the form of \([a, a]\). Two time intervals \(I\) and \(I'\) are adjacent iff \(r(I) = l(I')\). For example, the following interval pairs are adjacent, \([1, 2, 3)\) and \([1, 2, 1, 2]\) and \([1, 2, 1, 3]\).

(Concurrent) **time interval sequences**: Time interval sequence \(\mathcal{T} = I_0I_1I_2...\) is a sequence of time intervals such that \(\mathcal{T}\) is adjacent and diverging. It is easy to see that \(\mathcal{T}\) is monotonic (i.e. \(l(I_i) \leq l(I_{i+1})\)).

(Concurrent) **timed state sequences**: Different from the sequential case, a concurrent timed state sequence can pass through multiple states at one (time) moment. We use \(\tau(t)\) to denote the sequence of states observed at time \(t\). This state sequence should be finite, because of the divergence property of time interval sequences (also referred to as the Non-Zeno property \([4]\)). In the sequel, we denote \(S^C_{Prop}\) as the set of all timed state sequences in concurrent systems over \(Prop\).

(Concurrent) **timed system** \(S\) is a set of concurrent timed state sequences.

**Interleaving semantics**

Interleaving semantics have been adopted in many formal frameworks, where simultaneous actions are modelled by action (or state) sequences. For example, the parallel execution \((a \parallel b)\) of actions \(a\) and \(b\) can be represented by a nondeterministic choice of two sequential action sequences \(a.b\) and \(b.a\). This interleaving of concurrent actions facilitates sequential computation of concurrent behaviors \([12]\).

Under the interleaving semantics, the timing behavior of a concurrent system is often formalized by a two-phase execution framework \([72]\). One phase is the advance of time (the abstraction of the physical time), and no action takes place during this phase. The other phase is the sequential execution of simultaneous actions (in an interleaved way). The two-phase model can be expressed by timed state sequences as defined above, in which a singular time interval is attached to every instantaneous
state. Simultaneous actions can be modelled as sequentialized instantaneous (state) transition sequences, where the duration of intermediate states of these transitions are zero.

Example 4.2 Assume that a state transition system performs two concurrent actions \( p \) and \( q \) at time point 3. One possible action sequence is \( p.q \), which can be represented as follows:

\[
\ldots \rightarrow (\delta_0, [2, 3]) \xrightarrow{p} (\delta_1, [3, 3]) \xrightarrow{q} (\delta_2, [3, 5]) \rightarrow \ldots,
\]

where a singular time interval \( [3, 3] \) is attached to state \( \delta_1 \).

A timed state sequence can be considered as a function, which maps each time point \( t \in \mathbb{R} \geq l(I_0) \) to a finite state sequence. There can be multiple representations of timed state sequences (with different state sequences and time interval sequences) for the same timed execution trace of a system. Based on a similar notion as the sequential case, here we use the same procedure to define the normal form of a timed state sequence for the concurrent case.

Definition 4.2 Two timed state sequences \( \tau = (\delta, I) \) and \( \tau' = (\delta', I') \) are equivalent (\( \equiv \)) iff for all \( t \in \mathbb{R} \geq l(I_0) \), \( \tau(t) = \tau'(t) \).

Any timed state sequence \( \tau \) in the concurrent case can be normalized to a timed state sequence \( \tau^* \), the normal form of \( \tau \), by performing merging operations (see Section 4.1.1). It is easy to see that two equivalent timed state sequences have the same normal form. We assume in the sequel that concurrent timed state sequences are in normal form, unless explicitly stated otherwise.

Labelling time interval sequences

As mentioned before, a timed state sequence \( (\delta, I) \) is viewed as a mapping from each time point \( t \in \mathbb{R} \geq l(I_0) \) to a finite state sequence. As a consequence, the order of occurrence of interleaved simultaneous actions can not be discriminated only by their observed time points. For instance, in the sequence in Example 4.2, two states \( (\delta_1 \text{ and } \delta_2) \) are both observed at time point 3, and this time point alone is not sufficient to distinguish their order. To solve this problem, a time-point labelling algorithm is applied to time interval sequences in timed state sequences to distinguish the order of interleaved simultaneous actions.

We use \( m_T^t \) to denote the number of intervals \( I \) such that \( t \in I \) and \( I \) is an interval in the time interval sequence of \( \tau \). In other words, \( m_T^t \) represents the number of states observed at time instant \( t \) in timed state sequence \( \tau \). Note that \( m_T^t \) is finite. The issuing time of an action is represented by a time pair \( \langle t, i \rangle \), in which \( i \) is a label of \( t \) and represents the interleaving order of observable states at time \( t \). In other words, a time pair \( \langle t, i \rangle \) can be seen as the integration of a “macro-time” \( t \in \mathbb{R} \geq l(I_0) \) and a linearly ordered discrete “micro-time” \( 1 \leq i \leq m_T^t \) [7]. The “macro-time” refers to the real time and the “micro-time” specifies the interleaving order of states observed at the same macro-time point. Correspondingly, a time interval sequence \( I \) in a timed state sequence is extended to a labelled time interval sequence \( I^* \). All time points in a time interval sequence are labelled with 1 except for the following two cases.


- **End-points of a singular time interval**: For a singular time interval, the right end point is always labelled with the same number as the left end point. For instance, \([7, 7]\) is labelled as \([⟨7, i⟩, ⟨7, i⟩]\).

- **Adjacent closed end-points**: For two adjacent time intervals, if the first time interval is right-closed and the second is left-closed, the right end point of the first is labelled with \(i\), and the label of the left end point of the second interval is \(i + 1\). For instance, time interval sequence \([3, 5][5, 5][5, ∞)\) is labelled as \([⟨3, 1⟩, ⟨5, 1⟩][⟨5, 1⟩, ⟨5, 1⟩][⟨5, 2⟩, ∞)\). The remaining time points inside the time interval are labelled with 1. For instance, time point 6 in the above time interval sequence is labelled as \((6, 1)\).

A timed state sequence can be labelled in a similar way as the above cases. For instance, the timed state sequence in Example 4.2 is labelled as:

\[... → (δ₀, [(2, 1), (3, 1)]) → (δ₁, [(3, 1), (3, 1)]) → (δ₂, [(3, 2), (5, 1)]) → ...\]

where we assume that only one action is observed at time point 2. By applying this labelling method to the time interval sequence of a timed state sequence, we can view a timed state sequence \(τ\) as a function that maps every time pair in the labelled time set \(T^{τ} = \{⟨t, i⟩ | t ∈ R^{≥(I₀)} ∧ 1 ≤ i ≤ m^{τ}\}\) to a state in \(\mathbb{P}^{Prop}\).

In \([72]\), a time domain is called an additive group \((T, +, 0)\), where the least element 0 exists and addition operation \(+\) specifies a total order on \(T\). In this thesis, the addition operation on \(T^{τ}\) is not of our major concern, while a total order between time pairs is of more interest. A total order \(<\) on set \(T^{τ}\) is defined as follows. Let \((t, i)\) and \((t', j)\) be two time pairs in \(T^{τ}\). \((t, i) < (t', j)\) iff \((1) t < t'\) or \((2) t = t'\) and \(i < j\). A subtraction operation for time pairs is defined as \((t, i) − (t', j) = t − t'\). Similar to time intervals, the greatest lower (least upper) bound of labelled time interval \(I\) is represented by \(l(I) \ (r(I))\). \(|I| = r(I) − l(I)\) denotes the length of \(I\).

According to the definition of the total order \(<\) on \(T^{τ}\), the order between any two time points in \(R^{≥(I₀)}\) does not change during the labelling. Furthermore, their quantitative distance also retains after the labelling. The labelling algorithm on a time interval sequence merely adds an ordering of interleaved simultaneous actions.

**Proposition 4.1** Let \(T\) be a time interval sequence and let \(T^{l}\) be its labelled time interval sequence. For any \(k < n(T)\), then \(G_k\), defined as \(G_k(t, i) = t\), is a bijection from labelled time interval \(I_k\) to \(I_k\).

In the following formalization of concurrent behaviors, whenever we use \(T\), we assume tacitly that \(T\) is also defined (and vice versa).

### 4.2 Timed state sequence space

In the previous section, we used timed state sequences to formalize the timing behavior of a system. To investigate property relations between these timed state sequences, we need first define their nearness (or closeness) relations. Usually, the
nearness between elements of a set can be specified by a particular structure over the set. Commonly used structures include metrics and uniform structures [10], which specify an order of the nearness between elements. For example, given metric space \((X, d)\), metric function \(d\) specifies the nearness between all elements as a total order. That is, for any \(x_1, x_2, y_1 \text{ and } y_2 \in X\),

\[
either d(x_1, x_2) \leq d(y_1, y_2) \text{ or } d(x_1, x_2) \geq d(y_1, y_2).
\]

Given uniform space \((X, U)\), function \(U\) specifies the nearness between all elements as a partial order, and the nearness from different elements to the same element as a total order. That is, for any \(x_1, x_2 \text{ and } y \in X\),

\[
either U(x_1, y) \leq U(x_2, y) \text{ or } U(x_1, y) \geq U(x_2, y).
\]

In this section, we define two displacement functions to measure the nearness from one timed state sequence to another. Each displacement function specifies the nearness from different elements to the same element as a partial order. So, it is neither a metric nor a uniform structure.

The displacement functions proposed in this section only measure the nearness between two timed state sequences, which share the same state sequence. The nearness between timed state sequences with different state sequences is not of our concern. Intuitively, the nearness between two timed state sequences is determined by the nearness between their time interval sequences.

For notational simplicity, the following abbreviations are employed throughout this part. We use subscript \(a\) to represent the absolute timing difference and \(r\) to represent the relative timing difference, and we use superscript \(i\) to represent time interval sequences and \(s\) to represent timed state sequences. For example, function \(D^a_{ia}\) is used to specify the proximity between two time interval sequences based on their absolute timing differences.

### 4.2.1 Proximity measure based on absolute timing differences

In this subsection, we define a displacement function to measure the nearness from one timed state sequence to another based on their absolute timing difference. More specifically, the function computes the displacement from one timed state sequence to another based on the deviations of their corresponding time-stamps. This function is first defined for set \(S^s_{Prop}\), and then is generalized to set \(S^c_{Prop}\).

**Proximity measure between timed state sequences in sequential systems**

We define displacement function \(D^a_{ia}\) over \(S^s_{Prop} \times S^s_{Prop}\) to measure the nearness between timed state sequences. The \(D^a_{ia}\)-displacement between two timed state sequences \(\tau \text{ and } \tau'\) is calculated based on absolute differences between their corresponding time-stamps. Before we formally give the definition for \(D^a_{ia}\), three auxiliary functions are first defined for time interval sequences.

---

3Notice that most proximity functions in the thesis do not specify the nearness between elements in a symmetric way, which is different from many commonly used proximity functions, such as metrics and uniform structures.
Definition 4.3 For time interval sequences $\bar{T}$ and $\bar{T}'$ with the same length (i.e. $n(\bar{T}) = n(\bar{T}')$), function $d^i_{\sup}(\bar{T}, \bar{T}')$ is defined as follows.

$$d^i_{\sup}(\bar{T}, \bar{T}') = \sup\{l(I'_i) - l(I_i) \mid i < n(\bar{T})\}.$$ 

In $R$, the supremum (sup) of set $S$ only exists when $S$ is bounded from above. In this definition, we define $\sup S = \infty$ when $S \subseteq R$ and unbounded from above.

Given two timed state sequences $\tau = (\delta, I)$ and $\tau' = (\delta', I')$, $\sup (l(I'_i) - l(I_i))$ describes the extent to which the state transition from state $\delta_{i-1}$ to $\delta_i$ in $\tau'$ lags behind that in $\tau$. The larger the value of $l(I'_i) - l(I_i)$ is, the later the corresponding transition occurs in $\tau'$. Furthermore, if $l(I'_i) - l(I_i) < 0$, the corresponding transition in $\tau'$ occurs earlier than that in $\tau$. Therefore, function $d^i_{\sup}(\bar{T}, \bar{T}')$ gives the least upper bound of the delay of corresponding transitions in $\tau'$ w.r.t. the time of state transitions in $\tau$. Similarly, we can also define function $d^i_{\inf}$ by the greatest lower bound of the delay of corresponding state transitions.

Definition 4.4 For two time interval sequences $\bar{T}$ and $\bar{T}'$ with the same length, function $d^i_{\inf}(\bar{T}, \bar{T}')$ is defined as follows.

$$d^i_{\inf}(\bar{T}, \bar{T}') = \inf\{l(I'_i) - l(I_i) \mid i < n(\bar{T})\}.$$ 

Similar to the supremum case, we define $\inf S = -\infty$ when $S \subseteq R$ and unbounded from below. Now, we define the displacement from one time interval sequence to another.

Definition 4.5 For time interval sequences $\bar{T}$ and $\bar{T}'$ with the same length, displacement function $D^i_\alpha$ is given as follows.

$$D^i_\alpha(\bar{T}, \bar{T}') = [d^i_{\inf}(\bar{T}, \bar{T}'), d^i_{\sup}(\bar{T}, \bar{T}')].$$ 

Function $D^i_\alpha$ defines the displacement from one time interval sequence to another as a non-empty closed interval in $R$. Several examples are:

- $D^i_\alpha(\bar{T}, \bar{T}') = [0, 0]$;
- $D^i_\alpha(\bar{T}, \bar{T}') = [2, 2]$, if $I = [0, 1)\ldots[i, i + 1)\ldots$ and $I' = [2, 3)\ldots[i + 2, i + 3)\ldots$ ($i \geq 0$).
- $D^i_\alpha(\bar{T}, \bar{T}') = [-2, -1.5]$, where $I = [2, 3)\ldots[2i + 2, 2i + 3)\ldots[2i + 3, 2i + 4)\ldots$ and $I' = [0, 1.5)\ldots[2i, 2i + 1.5)\ldots[2i + 1.5, 2(i + 1))\ldots$ ($i \geq 0$).
- $D^i_\alpha(\bar{T}', \bar{T}) = [1.5, 2]$, where $\bar{T}$ and $\bar{T}'$ are the same as in the above case.

Now, we can directly derive $D^\alpha_\nu$ between timed state sequences from $D^i_\nu$.

Definition 4.6 Let $\tau$ and $\tau'$ be two timed state sequences with the same state sequence, and let $\bar{T}$ and $\bar{T}'$ be time interval sequence of $\tau$ and $\tau'$ respectively. Displacement function $D^\alpha_\nu$ over $S^S_{\prop} \times S^S_{\prop}$ is defined as follows.

$$D^\alpha_\nu(\tau, \tau') = D^i_\nu(\bar{T}, \bar{T}').$$
\( D_a \) defines the displacement from \( \tau \) to \( \nu \) as a closed time interval \([x, y]\), where \( x \) represents the greatest lower bound of the delay of transitions in \( \tau \) w.r.t. their corresponding transitions in \( \nu \), and \( y \) represents the least upper bound of the delay of corresponding transitions. It is easy to see that \( x > 0 \) indicates that state transitions in \( \tau \) are always observed later than their corresponding state transitions in \( \nu \). Similarly, \( y < 0 \) indicates that transitions in \( \nu \) are always earlier than those in \( \tau \). Since \( x \leq y \), we have the following proposition.

**Proposition 4.2** For any two timed state sequences \( \tau \) and \( \nu \) with the same state sequence, \( D_a(\tau, \nu) \neq \emptyset \).

**Example 4.3** The two timed state sequences \( \tau_1 \) and \( \tau_2 \) shown in Figure 4.1 have the same state sequence. There are five state transitions \(^4\) in each timed state sequence. It is easy to calculate that \( d^s_{\min}(\tau_2, \tau_1) = \sup\{0 - 0, 1.1 - 1.2, 2.3 - 2.2, 3.3 - 3.4, 4.4 - 4.2\} = 0.2 \) and \( d^s_{\min}(\tau_2, \tau_1) = -0.1 \). Therefore, the \( D_a^s \)-displacement from \( \tau_2 \) to \( \tau_1 \) is \([-0.1, 0.2]\). Since \( d^s_{\min}(\tau_1, \tau_2) = 0.1 \) and \( d^s_{\min}(\tau_1, \tau_2) = -0.2 \), the \( D_a^s \)-displacement from \( \tau_1 \) to \( \tau_2 \) is \([-0.2, 0.1]\).

It is worth noticing that \( D_a^s \)-displacements are directional. That is, \( D_a^s(\tau, \nu) \) is not always equal to \( D_a^s(\nu, \tau) \). However, we can easily calculate the \( D_a^s \)-displacement from \( \tau \) to \( \nu \) based on that from \( \nu \) to \( \tau \). This follows from Proposition 4.3.

**Proposition 4.3** For any two timed state sequences \( \tau \) and \( \nu \) with the same state sequence, \( D_a^s(\tau, \nu) = [x, y] \) implies that \( D_a^s(\nu, \tau) = [-y, -x] \).

Now, let’s take a closer look at the notion of “nearness” specified by the \( D_a^s \) function. Intuitively speaking, the nearness of timed state sequences to \( \tau \) can be ordered by the inclusive relation between their \( D_a^s \)-displacements. More precisely, \( \tau_1 \) is closer to \( \tau \) than \( \tau_2 \) to \( \tau \) iff \( D_a^s(\tau_1, \tau) \subseteq D_a^s(\tau_2, \tau) \). Obviously, the nearness relation specified by \( D_a^s \) is different from that specified by uniform structures (or distance metrics), which allows the comparison of the nearness from any two elements to one specific element. \( D_a^s \) only specifies a partial order for the nearness from different elements to the same element, instead of a total order. For example, when \( D_a^s(\tau_1, \tau) = [2, 3] \) and \( D_a^s(\tau_2, \tau) = [-4, 1] \), it is not of our interest to compare \( D_a^s(\tau_1, \tau) \) and \( D_a^s(\tau_2, \tau) \).

---

\(^4\)We assume that the initial state is also triggered by a state transition.
**Definition 4.7** For any sequential timed state sequence $\tau$, $x, y \in \mathbb{R}$ and $x \leq y$, we define an absolute $[x, y]$-tube of $\tau$ as

$$\{ \tau' \in S_{Prop}^S \mid D^s_a(\tau, \tau') \subseteq [x, y] \},$$

and denote it as $\tau_a^{[x, y]}$. Furthermore, we call $\tau'$ is absolute $[x, y]$-close to $\tau$ if and only if $\tau'$ belongs to $\tau_a^{[x, y]}$.

$\tau_a^{[x, y]}$ defines a set including all timed state sequences $\tau'$ that satisfy the requirement that the $D^s_a$-displacement from $\tau$ to $\tau'$ is a subset of $[x, y]$. It is easy to conclude that $\tau$ is inside its absolute $[x, y]$-tube iff $x \leq 0$ and $y \geq 0$, since $D^s_a(\tau, \tau) = [0, 0]$.

An absolute $[x, y]$-tube of $\tau$ can also be visualized in a graph as shown in Figure 4.2. The $x$-axis represents the states of $\tau$ and the $y$-axis represents time. Each polyline in the figure represents a timed state sequence. Since time interval sequences in the sequential case are strictly monotonic, the slope of each segment in the polyline should be positive. In the concurrent case, the slope of each segment should be non-negative.

**Proximity measure between timed state sequences in concurrent systems**

Similar to the sequential case, we can define the $D^s_a$ function for the concurrent case. In the concurrent case, time intervals and time interval sequences in the relevant definitions are replaced by labelled time intervals and labelled time interval sequences. Here we only give the definition of function $d^s_{a, \text{no}}$. The other functions can be defined in a similar manner to those in the sequential case.
Definition 4.8 For two labelled time interval sequences $\mathcal{T}$ and $\mathcal{T}'$ with the same length, function $d_{\text{sup}}^i(\mathcal{T}, \mathcal{T}')$ is defined as follows.

$$d_{\text{sup}}^i(\mathcal{T}, \mathcal{T}') = \sup \{ l(I'_i) - l(I_i) \mid i < n(\mathcal{T}) \}.$$ 

Similar to the sequential case, we can also define an absolute $[x, y]$-tube for a timed state sequence $\tau$ in $S^C_{\text{Prop}}$ as $\{ \tau' \in S^C_{\text{Prop}} \mid D^a(\tau, \tau') \subseteq [x, y] \}$.

### 4.2.2 Proximity measure based on relative timing differences

In the previous subsection, we defined displacement function $D^a$ to measure the nearness between timed state sequences based on absolute differences between their corresponding time-stamps. In this subsection, we present a different displacement function $D^r$ to measure the nearness between timed state sequences, which is based on relative differences between their corresponding time-durations. More specifically, in this subsection we consider the case that the displacement between two timed state sequences is caused by the inconsistent change rates of clocks that are used to measure the time-stamps of state transitions (or the time-durations of states).

Now, reconsider state sequence $\delta$ in Figure 4.1. Assume we use two clocks $A$ and $B$ to measure the time duration of each state in $\delta$. Due to the inconsistent change rates of the clocks, we obtain two different timed state sequences $\tau_1$ and $\tau_2$ correspondingly. In Figure 4.1, we can see that the average change rate of clock $A$ is $\frac{1}{12}$ faster than that of clock $B$ during the observation of state $\delta_0$.

### Proximity measure between timed state sequences in sequential systems

Now we give the relevant definitions for $D^r$ in the sequential case.

Definition 4.9 For two time interval sequences $\mathcal{T}$ and $\mathcal{T}'$ with the same length, function $d^i_{\text{sup}}(\mathcal{T}, \mathcal{T}')$ is defined as follows.

$$d^i_{\text{sup}}(\mathcal{T}, \mathcal{T}') = \sup \{ \frac{|I'_i|}{|I_i|} \mid i < n(\mathcal{T}) \}.$$ 

In the case that both $|I_i|$ and $|I'_i|$ are $\infty$, we leave $d^i_{\text{sup}}(\mathcal{T}, \mathcal{T}')$ undefined.

In the above definition, $\frac{|I'_i|}{|I_i|}$ gives the ratio between the durations of the $i$-th time intervals in $\mathcal{T}$ and $\mathcal{T}'$. Assume that both $\mathcal{T}$ and $\mathcal{T}'$ are attached to the same state sequence, and clocks $A$ and $B$ are used to measure $I_i$ and $I'_i$ respectively. If $\frac{|I'_i|}{|I_i|} > 1$, the average change rate of clock $B$ is faster than that of clock $A$ during the observation of the $i$-th state. In other words, the ratio between the durations of two corresponding intervals in two timed state sequences can also be considered as the ratio between the average change rates of the clocks during the observation of the same state.

Similar to the absolute timing difference case, we can easily define functions $d^i_{\text{inf}}$, $D^i$, and $D^r$ for the relative timing difference case. Here we skip their detailed definitions for brevity.
Example 4.4 If we use two clocks (A and B) to measure the time-duration of each state in state sequence $\bar{\delta}$, two timed state sequences $(\bar{\delta}, T_A)$ and $(\bar{\delta}, T_B)$ can be derived from those measured durations. If clock B is always twice as fast as clock A, then each time-duration in $T_B$ should be twice as long as its corresponding time-duration in $T_A$. In this case, $d^i_{\text{rup}}(T_A, T_B) = d^i_{\text{inf}}(T_A, T_B) = 2$, and $D^r_\tau(T_A, T_B) = D^r_\tau(T_A, T_B) = [2, 2]$.

Let $r_i$ be the ratio between the average change rates of two clocks during the observation of the $i$-th state in sequence $\bar{\delta}$. The $D^r_\tau$ function defines the nearness between two timed state sequences as a non-empty closed interval in $\mathbb{R}^2$, the left-end of which represents the greatest lower bound over all $r_i$ ($0 \leq i < n(\bar{\delta})$), and the right-end of which represents the least upper bound over all $r_i$.

In reality, the change rates of clocks may be variable during the observation of each state. One can question whether we can use the ratio between the change rates of clocks to approximate $D^r_\tau(T, \bar{\tau})$. This is discussed in the following example.

Example 4.5 Given two clocks A and B, and assume that the change rate of clock B can deviate (faster or slower) from that of clock A by at most $a\%$ ($a < 100$). If we use clocks A and B to measure the duration of each state in an untimed execution $\bar{\delta}$, two timed state sequences $\bar{\tau}$ and $\bar{\bar{\tau}}$ can be derived from these measured durations.

Assume $\delta_i$ is the $i$-th state in $\bar{\delta}$, and $t_l(t'_l)$ and $t_r(t'_r)$ are the left-end and the right-end of the duration of state $\delta_i$ counted by clock A (clock B). Since the difference between the change rate of clock B and that of A is less than $a\%$, we know that $(1 - \frac{a}{100}) (t_r - t_l) \leq (t'_r - t'_l) \leq (1 + \frac{a}{100}) (t_r - t_l)$. By the definition of function $D^r_\tau$, we can see that $D^r_\tau(\bar{\tau}, \bar{\bar{\tau}}) \subseteq [1 - \frac{a}{100}, 1 + \frac{a}{100}]$.

Assume two clocks are used to measure the duration of each state in sequence $\bar{\delta}$ and two timed state sequences are derived from the duration measurements. If the ratio between the change rates of two clocks is bounded, we can see from the above example that the ratio between corresponding time-durations (the lengths of time intervals) in two timed state sequences is bounded too. The other way around, however, is not true. For example, we can use a digital clock and an analog clock to time the duration of each state in a state sequence. The ratio between corresponding time-durations can be bounded, but the ratio between the change rates of both clocks is unbounded.

Example 4.6 Consider the two timed state sequences as shown in Figure 4.1. We can calculate

$$d^i_{\text{rup}}(T_1, T_2) = \sup\{\frac{1.2}{1.1}, \frac{1}{1.2}, \frac{1.2}{1}, \frac{0.8}{1.1}\} = \frac{6}{5}.$$  

Similarly, we can compute that $d^i_{\text{inf}}(T_1, T_2) = \frac{8}{11}$, $d^i_{\text{rup}}(T_2, T_1) = \frac{11}{8}$, and $d^i_{\text{inf}}(T_2, T_1) = \frac{5}{6}$. The $D^r_\tau$-displacements between $T_1$ and $T_2$ are:

$$D^r_\tau(T_1, T_2) = [d^i_{\text{inf}}(T_1, T_2), d^i_{\text{rup}}(T_1, T_2)] = \left[\frac{8}{11}, \frac{6}{5}\right],$$  

$$D^r_\tau(T_2, T_1) = [d^i_{\text{inf}}(T_2, T_1), d^i_{\text{rup}}(T_2, T_1)] = \left[\frac{5}{6}, \frac{11}{8}\right].$$
In the relative timing difference case, the $D^s_r$-displacement from $\tau$ to $\tau'$ can be directly derived from the $D^s_r$-displacement from $\tau'$ to $\tau$. This is formally stated in the following proposition.

**Proposition 4.4** For any two timed state sequences $\tau$ and $\tau'$ with the same state sequence, $D^s_r(\tau, \tau') = [x, y]$ implies that $D^s_r(\tau', \tau) = [1/y, 1/x]$.

Similar to the absolute timing difference case, we also define a relative $[x, y]$-tube $(x \leq y$ and $x, y \in \mathbb{R}^\geq 0$) for a timed state sequence $\tau$. This tube can be visualized in a state-rate graph of $\tau$ as shown in Figure 4.3. The $x$-axis represents the states of $\tau$ and the $y$-axis represents the relative change rates of clocks w.r.t. the change rates of the clock used in $\tau$ during the observation of each state. A relative $[x, y]$-tube of $\tau$ contains all timed state sequences $\tau'$, whose $D^s_r$-displacement from $\tau$ is a subset of $[x, y]$. Each timed state sequence in the tube can be represented as a sequence of horizontal line segments in the graph.

**Definition 4.10** For any sequential timed state sequence $\tau$, $x, y \in \mathbb{R}^\geq 0$ and $x \leq y$, we define a relative $[x, y]$-tube of $\tau$ as follows.

$$\tau^r_{[x,y]} = \{ \tau' \in \mathcal{S}^S_{Prop} \mid D^s_r(\tau, \tau') \subseteq [x, y] \}.$$ 

Furthermore, we call $\tau'$ is relative $[x, y]$-close to $\tau$ if and only if $\tau'$ belongs to $\tau^r_{[x,y]}$.

**Proximity measure between timed state sequences in concurrent systems**

Similar to the absolute case, we can also redefine the $D^s_r$ function for the concurrent case, where time intervals and time interval sequences in the relevant definitions...
Proximity measures between timing behaviors

Figure 4.4: Counter examples

are replaced by labelled time intervals and labelled time interval sequences. Here we only give the definition of function \( d_{r_{\text{sup}}} \). The other functions can be defined correspondingly.

\textbf{Definition 4.11} For two time interval sequences \( \mathcal{I} \) and \( \mathcal{I}' \) with the same length, function \( d_{r_{\text{sup}}} (\mathcal{I}, \mathcal{I}') \) is defined as follows.

\[
d_{r_{\text{sup}}} (\mathcal{I}, \mathcal{I}') = \sup \{|I'_i| / |I_i| \mid i < n(\mathcal{I})\}.
\]

In the case that both \( |I_i| \) and \( |I'_i| \) are \( \infty \) or both \( |I_i| \) and \( |I'_i| \) are zero, we leave \( |I'_i| / |I_i| \) undefined.

\section{4.3 Discussions}

In a metric space, the nearness between elements in a set is determined by a metric function. Different metric functions can map the distance of two elements to different positive reals. However, these metric functions may still imply the same metric topology. From this point of view, different metrics are considered equivalent iff they define the same metric topology.

More formally, given set \( S \), metric \( g \) defined for \( S \), element \( s \in S \) and a positive real \( \epsilon \in R_{>0} \), an \( \epsilon \)-ball is defined to be the set \( B_g(s, \epsilon) = \{s' \mid g(s, s') < \epsilon\} \). Two metrics \( g \) and \( g' \) on set \( S \) are equivalent iff for any \( \epsilon \in R_{>0} \) and \( s \in S \), there exist positive reals \( \epsilon_1 \) and \( \epsilon_2 \) such that \( B_g(s, \epsilon_1) \subseteq B_{g'}(s, \epsilon) \) and \( B_{g'}(s, \epsilon_2) \subseteq B_g(s, \epsilon) \). The intuition behind this definition is that whenever two elements are close in one metric space, they are also close in an equivalent metric space.

In the previous section, we have proposed two displacement functions \( D_s^a \) and \( D_s^p \) to measure the nearness between timed state sequences. Now it is natural to question whether these functions are “equivalent” in a similar sense to the metric equivalence. Note that a timed state sequence set with a displacement function is neither a metric space nor a topology space. Here we borrow the metric equivalence concept from the metric space to address the difference between the two displacement functions. Let us first look at two simple examples.
Example 4.7 Consider the two timed state sequences \( \tau = (\delta, \bar{T}) \) and \( \tau' = (\delta, \bar{T}') \) as shown in Figure 4.4(a). Time interval sequences \( \bar{T} \) and \( \bar{T}' \) are defined as follows:

\[
\bar{T} = [0, 2)(2, 2\frac{1}{2})(2\frac{1}{2}, 4)\ldots [2k, 2k + \frac{1}{2k}, 2k + 2)\ldots \quad k \geq 1
\]

\[
\bar{T}' = [0, 2)(2, 3)(3, 4)\ldots [2k, 2k + 1, 2k + 2)\ldots \quad k \geq 1
\]

It is easy to derive that \( D^s_a(\tau, \tau') = [0, 1] \) and \( D^s_p(\tau, \tau') = [2, 2] \).

Example 4.8 Consider the two timed state sequences \( \tau = (\delta, \bar{T}) \) and \( \tau' = (\delta, \bar{T}') \) as shown in Figure 4.4(b). Time interval sequences \( \bar{T} \) and \( \bar{T}' \) are defined as follows:

\[
\bar{T} = [0, 1)(1, 2)\ldots [k, k + 1)\ldots \quad k \geq 0
\]

\[
\bar{T}' = [0, 2)(2, 4)\ldots [2k, 2k + 2)\ldots \quad k \geq 0
\]

It is easy to derive that \( D^s_a(\tau, \tau') = [0, \infty] \) and \( D^s_p(\tau, \tau') = [2, 2] \).

Example 4.7 shows that timed state sequences \( \tau \) and \( \tau' \) have finite \( D^s_a \)-displacement, but infinite \( D^s_p \)-displacement. Example 4.8 illustrates that timed state sequences \( \tau \) and \( \tau' \) have finite \( D^s_a \)-displacement, but infinite \( D^s_p \)-displacement. These examples give an indication that two close timed state sequences in a space with displacement function \( D^s_a \) may not be close in another space with displacement function \( D^s_p \), and conversely. Now we give a brief proof of this statement.

First, we define an \((x, y)\)-ball \((x < y)\) for a displacement function \( D \) on a set of timed state sequences \( S \). It is given by

\[
B_D(\tau, x, y) = \{ \tau' \mid D(\tau, \tau') \subset (x, y) \}.
\]

Consider the timed state sequence \( \tau \) in Example 4.7. For any \( 0 < \epsilon < 0.5 \) and \( B_{D^s_a}(\tau, -\epsilon, \epsilon) \), we can always find a timed state sequence \( \tau' \) which has a time interval sequence as follows:

\[
\bar{T}' = [0, 2)\ldots [2k, 2k + \frac{1}{2k}, 2k + 2)\ldots \kappa \geq 1.
\]

If \( \tau \) and \( \tau' \) share the same time sequence, it is easy to see that \( \tau' \) is in \( B_{D^s_a}(\tau, -\epsilon, \epsilon) \), but \( D^s_a(\tau, \tau') = \infty \). Similarly, consider timed state sequence \( \tau \) in Example 4.8. For any \( 1 < \epsilon < 2 \) and \( B_{D^s_p}(\tau, \frac{1}{2}, \epsilon) \), we can always find a timed state sequence \( \tau' \) in \( B_{D^s_p}(\tau, \frac{1}{2}, \epsilon) \), which has an infinite \( D^s_a \)-displacement from \( \tau \). For example, the time interval sequence of \( \tau' \) can be given by

\[
\bar{T}' = [0, \frac{\epsilon + 1}{2}, \frac{\epsilon + 1}{2} \ldots [k\frac{\epsilon + 1}{2}, (k + 1)\frac{\epsilon + 1}{2}) \ldots \kappa \geq 0.
\]

Now we can see that two spaces with displacement function \( D^s_a \) and \( D^s_p \) do not always induce the same nearness relations between two timed state sequences. In other words, \( D^s_a \) and \( D^s_p \) define different proximity measures between timed state sequences and are not interchangeable ("equivalent").
4.4 Related work

In [36], several equivalent distance metrics were proposed for finite timed action sequences (called trajectories). These metrics were used to define robust timed automata, which is discussed in Section 7.6. Here we mainly concentrate on the characteristics of these metrics.

To facilitate the following discussion, we first show that a trajectory can be encoded into a finite timed state sequence. As a consequence, metrics for trajectories can also be applied to measure the nearness between timed state sequences.

Denote a trajectory as \( \tau_a = (a_1, t_1)(a_2, t_2)...(a_i, t_i)...(a_n, t_n) \), where \( a_i \) represents the \( i \)-th event, \( t_i \) represents the time-stamp of \( a_i \) and the length of \( \tau_a \) is \( n \). The corresponding timed state sequence \( \tau_\delta \) can be as follows:

\[
\tau_\delta = (S_{a_1}, [t_1, t_2]) (S_{a_2}, [t_2, t_3])... (S_{a_i}, [t_i, t_{i+1}])... (S_{a_n}, [t_n, \infty]),
\]

where \( S_{a_i} \) is a state triggered by event \( a_i \) at state \( S_{a_{i-1}} \) and where \([t_i, t_{i+1}]\) represents the time-duration of \( S_{a_i} \). Assume that the time-stamps of events in a trajectory are actually the time-stamps of state transitions in its corresponding timed state sequence, then it is trivial to show that the metrics in [36] can also be applied to measure the nearness between timed state sequences correspondingly. In the following, we restrict our discussion of nearness measurements to time state sequences.

Among the proposed metrics in [36], \( d_{\text{max}} \) is used to measure the nearness of two timed state sequences based on the maximal absolute value of the absolute differences between the corresponding timing stamps. In [49], metric \( d_{\text{sup}} \) is defined to measure the nearness of two timed state sequences, which is based on the same concept as \( d_{\text{max}} \). However, \( d_{\text{sup}} \) differs from \( d_{\text{max}} \) in that it is defined for both finite and infinite sequences.

Metric \( d_{\text{max}} \) (or \( d_{\text{sup}} \)) uses a single variable to capture the bound of the absolute difference between two timed state sequences. For example, \( d_{\text{max}}(\tau, \tau') = 3 \) indicates that the absolute timing differences of corresponding time-stamps in \( \tau \) and \( \tau' \) are always within interval \([-3, 3]\). To give a more precise measurement of the absolute timing difference between timed state sequences, we generalize those metrics to a \( D_\alpha \) displacement function in this chapter, which uses an interval \([x, y]\) to capture the greatest lower (least upper) bound of the absolute timing differences between corresponding time stamps in two timed state sequences. In Chapter 7, we show that such a metric can lead to an improved real-time property-preservation result between timed systems.

In addition to the \( d_{\text{max}} \) metric, "metric" \( d_{\text{drift}} \) was also proposed to measure the relative timing difference between two finite timed state sequences in [36]. Metric \( d_{\text{drift}} \) assumes that the relative timing difference between timed state sequences is caused by the drift of the clocks that are used for counting time-stamps. This difference is measured by the maximal ratio between the total elapsed time. For example,

\[\text{Strictly speaking, } d_{\text{drift}} \text{ does not completely comply with the mathematical definition of metric.}\]
we use clocks $x$ and $x'$ to count time-stamps of the same state sequence $\overline{\delta} = \delta_1 \delta_2 \delta_3 \delta_4$ respectively and obtain two timed state sequences $\overline{\tau}$ and $\overline{\tau'}$ as follows:

\[
\overline{\tau} = (\delta_1, [1, 100])(\delta_2, [100, 101])(\delta_3, [101, 106])(\delta_4, [106, \infty))
\]

\[
\overline{\tau'} = (\delta_1, [1, 100])(\delta_2, [100, 110])(\delta_3, [110, 111])(\delta_4, [111, \infty)).
\]

The $d_{drift}$ between $\overline{\tau}$ and $\overline{\tau'}$ can be calculated as:

\[
\max\{\frac{1}{1}, \frac{100}{101}, \frac{101}{100}, \frac{110}{101}, \frac{106}{111}, \frac{111}{106}\} - 1 = \frac{9}{101}
\]

Since the $d_{drift}$ metric measures the distance based on the total elapsed time, the “local” drift of clocks cannot be captured by the metric. In the above example, clock $x'$ is at least 10 times faster than clock $x$ during the observation of state $\delta_2$. However, under the $d_{drift}$ metric, the relative timing difference between $\overline{\tau}$ and $\overline{\tau'}$ is less than 0.1. The $d_{drift}$ metric is not suitable for our later discussion on the property preservation, because real-time property-preservation is closely related to the “local” timing drift between two timed state sequences. In this chapter, we proposed displacement function $D^*_s$ to measure the relative timing difference, which is based on the ratio between time durations of corresponding states in two timed state sequences. In the above example, the $D^*_s$-displacement from $\overline{\tau}$ to $\overline{\tau'}$ is $[\frac{9}{101}, 10]$, which indicates that the average change rate of clock $x'$ at the observation of each state is at most 10 times faster and at least $\frac{9}{101}$ faster than that of clock $x$.

### 4.5 Summary

In this chapter, we formalized real-time systems for the sequential case and the concurrent case respectively. Based on these formalizations, we defined two displacement functions to measure the nearness between timed state sequences. Displacement function $D^*_s$ measures the nearness between timed state sequences based on their absolute timing differences, while function $D^*_r$ is based on their relative timing differences. These functions measure the nearness between timed state sequences from different perspectives and can lead to different preservation results between their real-time properties (see Chapter 7).
Chapter 5

Weakening real-time properties

In this chapter, we investigate real-time properties of a system and their weakening relations. In Section 5.1, we introduce the MTL logic, which is used to formalize both qualitative and quantitative real-time properties in this thesis. MTL formulas can be used to express real-time properties for both sequential and concurrent systems, when interpreted in the corresponding domains. In Section 5.2, we investigate a weakening relation between MTL formulas. Specifically, we prove that this weakening relation between MTL formulas can be derived from their sub-formulas and time-bounds. In Section 5.3, we propose several weakening functions over MTL formulas as special cases of the weakening relation. These functions are used to establish the property relations between timed systems in Chapter 7.

5.1 Formalization of real-time properties

Real-time properties of a system can be represented by linear-time or branching-time temporal logics (see Section 3.2.1). Since the reasoning of property-preservation between timed systems in this thesis is based on timed state sequences without considering branching structures, we choose a linear-time temporal logic to formalize real-time properties of timed systems. More specifically, MTL is adopted in this thesis.

5.1.1 MTL logic

MTL formulas have the following syntactic forms:

\[ \varphi ::= p \mid \neg p \mid \varphi_1 \lor \varphi_2 \mid \varphi_1 \land \varphi_2 \mid \varphi_1 [t_1,t_2] \mid \varphi_1 [t_1,t_2) \mid \varphi_1 (t_1,t_2] \mid \varphi_1 (t_1,t_2)) , \]

where time-bound \( I \) is an interval of nonnegative reals. It takes one of the following forms: \( \emptyset, [a,a], [a,b], (a,b), (a,b], [a,\infty) \) and \( (a,\infty) \), where \( a < b \) for \( a, b \in R^{\geq 0} \).
Interpretation of MTL formulas in sequential timed systems

The interpretation of MTL formulas over timed state sequences in sequential systems is given in Definition 5.1. If $\tau$ is a timed state sequence and $t \in I$, then $(\tau, t)$ represents a suffix of $\tau$, the interval sequence of which starts with interval $[t, r(I))$ and the state sequence of which starts with state $\delta_i$.

**Definition 5.1** Let $\tau \in S^S_{prop}$ and $t \in R^{\geq l(I_o)}$. For any MTL formula $\varphi$, the interpretation of $\varphi$ over $(\tau, t)$ is given as follows:

- $(\tau, t) \models p$ iff $p \in \tau(t)$;
- $(\tau, t) \models \neg p$ iff $p \notin \tau(t)$;
- $(\tau, t) \models \varphi_1 \lor \varphi_2$ iff $(\tau, t) \models \varphi_1$ or $(\tau, t) \models \varphi_2$;
- $(\tau, t) \models \varphi_1 \land \varphi_2$ iff $(\tau, t) \models \varphi_1$ and $(\tau, t) \models \varphi_2$;
- $(\tau, t) \models \varphi_1 U_I \varphi_2$ iff there is some $t_2 \in I$, such that $(\tau, t + t_2) \models \varphi_2$ and for all $0 \leq t_1 < t_2$, $(\tau, t + t_1) \models \varphi_1$;
- $(\tau, t) \models \varphi_1 V_I \varphi_2$ iff for all $t_2 \in I$, $(\tau, t + t_2) \models \varphi_2$ or there is some $0 \leq t_1 < t_2$, $(\tau, t + t_1) \models \varphi_1$.

In case that $I$ is empty, $(\tau, t) \models \varphi_1 U_I \varphi_2$ is always false and $(\tau, t) \models \varphi_1 V_I \varphi_2$ is always true. We use $(\tau, l(I_o)) \models \varphi$ (in short) to denote that sequential timed state sequence $\tau$ satisfies MTL formula $\varphi$.

Interpretation of MTL formulas in concurrent timed systems

MTL formulas can also be interpreted over timed state sequences in concurrent systems. The interpretation for the concurrent case is more complex than that for the sequential case, but both interpretations originate from the same inspiration.

**Definition 5.2** Let $\varphi$ be an MTL formula. Further let $\tau \in S^C_{prop}$. For any $(t, i) \in T^\tau$, the interpretation of $\varphi$ over $(\tau, (t, i))$ is given as follows:

- $(\tau, (t, i)) \models p$ iff $p \in \tau(t, i)$;
- $(\tau, (t, i)) \models \neg p$ iff $p \notin \tau(t, i)$;
- $(\tau, (t, i)) \models \varphi_1 \lor \varphi_2$ iff $(\tau, (t, i)) \models \varphi_1$ or $(\tau, (t, i)) \models \varphi_2$;
- $(\tau, (t, i)) \models \varphi_1 \land \varphi_2$ iff $(\tau, (t, i)) \models \varphi_1$ and $(\tau, (t, i)) \models \varphi_2$;
- $(\tau, (t, i)) \models \varphi_1 U_I \varphi_2$ iff there exist $t_2 \in I$, $j (1 \leq j \leq m_{t+1,j})$ and $(t, i) \leq (t + t_2, j)$, such that $(\tau, (t + t_2, j)) \models \varphi_2$ and for all $t_1$ and $k (1 \leq k \leq m_{t_1,i})$ that satisfy $(t, i) \leq \langle t_1, k \rangle < \langle t + t_2, j \rangle$, $(\tau, (t_1, k)) \models \varphi_1$;
- $(\tau, (t, i)) \models \varphi_1 V_I \varphi_2$ iff for all $t_2 \in I$, $j (1 \leq j \leq m_{t+1,j})$ and $(t, i) \leq \langle t + t_2, j \rangle$, either $(\tau, (t + t_2, j)) \models \varphi_2$ or there exist $t_1$ and $k (1 \leq k \leq m_{t_1,i})$, such that $(t, i) \leq \langle t_1, k \rangle < \langle t + t_2, j \rangle$ and $(\tau, (t_1, k)) \models \varphi_1$. 
Table 5.1: Rules to move negation to atomic propositions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg\neg \varphi \Rightarrow \varphi$</td>
<td>$\neg(\varphi_1 \lor \varphi_2) \Rightarrow \neg\varphi_1 \land \neg\varphi_2$</td>
</tr>
<tr>
<td>$\neg(\varphi_1 \land \varphi_2) \Rightarrow \neg\varphi_1 \lor \neg\varphi_2$</td>
<td>$\neg(\varphi_1 V_I \varphi_2) \Rightarrow \neg\varphi_1 U_I \neg\varphi_2$</td>
</tr>
<tr>
<td>$\neg(\varphi_1 U_I \varphi_2) \Rightarrow \neg\varphi_1 V_I \neg\varphi_2$</td>
<td>$\neg(\varphi_1 U_I \varphi_2) \Rightarrow \neg\varphi_1 V_I \neg\varphi_2$</td>
</tr>
</tbody>
</table>

Table 5.2: Syntactic abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>$T \equiv p \lor \neg p$</td>
</tr>
<tr>
<td>false</td>
<td>$F \equiv p \land \neg p$</td>
</tr>
<tr>
<td>eventually</td>
<td>$\diamond_I \varphi \equiv TU_I \varphi$</td>
</tr>
<tr>
<td>always</td>
<td>$\square_I \varphi \equiv FV_I \varphi$</td>
</tr>
<tr>
<td>weakly until</td>
<td>$\varphi_1 U_I \varphi_2 \equiv (\varphi_1 U_I \varphi_2) \lor \square_I \varphi_1$</td>
</tr>
<tr>
<td>implication</td>
<td>$\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \lor \varphi_2$</td>
</tr>
</tbody>
</table>

We use $(\tau, \{l(I_0), 1\}) \models \varphi$ ($\models \varphi$, in short) to denote that concurrent timed state sequence $\tau$ satisfies MTL formula $\varphi$.

In Section 3.2.2, we choose $\neg$ ("not"), $\land$ ("and") and $U$ ("until") as the basic operators of the MTL logic. Here we choose another set of basic operators: $\land$, $\lor$ ("or"), $U$ and $V$ ("unless")\(^1\) to define a "negation-free" MTL logic, where negation only appears in front of atomic propositions. However, both logics have the same expressive power. For example, any negation formula can be represented as a "negation-free" formula by applying the rules given in Table 5.1. On the other hand, any unless formula can be expressed by an until formula with negations (i.e. $\varphi_1 V_I \varphi_2 = \neg(\neg\varphi_1 U_I \neg\varphi_2)$).

In the case that $I$ is $[0, \infty)$, we omit time-bound $I$ of temporal operators. Several useful abbreviations are defined in Table 5.2.

Consider one time-bounded version of a response property, which states that every stimulus $req$ is always followed by response $ack$ within 2 to 3 seconds. This real-time property can be formalized in MTL logic as follows:

$$\square(req \rightarrow \diamond_{[2,3]}ack).$$

Since a timed system can be viewed as a set of timed state sequences, its real-time property satisfaction can be formally defined using the linear trace interpretation of MTL. The formal definition is as follows.

\(^1\)Unless operator $V$ defined in this section is different from the conventional unless operator $U$, which is also called the weakly until operator (see Table 5.2).
Weakening real-time properties

5.1 The semantics of the until operator

In this subsection, we give a discussion on the choice of left-closed time intervals in Section 4.1, which closely relates to the semantics of the until operator in dense time domain.

The semantics of the until operator states that formula $\varphi_1 U \varphi_2$ holds at certain time $t$ in a timed state sequence $\tau$ iff there exists a $t'$ at which $\varphi_2$ is true, and $\varphi_1$ is true in interval $[t, t')$ (as shown in Figure 5.1(a)). Given any timed state sequence $\tau$ (as shown in Figure 5.1(b)), $\varphi_1 \land \neg \varphi_2$ and $\varphi_2 \land \neg \varphi_1$ hold respectively in time interval $[t_1, t_2]$ and $(t_2, t_3]$. However, $\varphi_1 U \varphi_2$ does not hold at time $t_1$, which is counter-intuitive. This is because for any $t \in (t_2, t_3]$, there always exists a $t'$ for which $t_2 < t' < t$ and $\varphi_1$ does not hold at $t'$. The semantics of $\varphi_1 U \varphi_2$ shows that the validation of until formulas in general depends on whether the second time interval is left-closed (referred to as the until operator problem). Therefore, we require that time intervals in (sequential and concurrent) timed state sequences are left-closed. A more detailed discussion about the semantics of the until operator can be found in [18].

5.2 Weakening formulas

A weakening relation exists between MTL formulas. Formula $\varphi$ is weaker than $\varphi'$, if the satisfaction of $\varphi'$ by a timed state sequence always implies the satisfaction of $\varphi$. Conversely, $\varphi'$ can be considered to be a stronger formula than $\varphi$. For example, formula $p \lor q$ is weaker than formula $p$.

The following definitions formally define the weakening relation between MTL formulas which are interpreted over sequential and concurrent timed state sequences respectively.

Definition 5.4 An MTL formula $\varphi'$ is a weaker formula than $\varphi$, iff for any timed state sequence $\tau \in S_{\text{seq}}^\infty$ and $t \in R^{\geq l(t_0)}$, $(\tau, t) \models \varphi$ implies $(\tau, t) \models \varphi'$.

Definition 5.5 An MTL formula $\varphi'$ is a weaker formula than $\varphi$, iff for any timed state sequence $\tau \in S_{\text{con}}^\infty$ and $(t, i) \in T^\tau$, $(\tau, (t, i)) \models \varphi$ implies $(\tau, (t, i)) \models \varphi'$.

It is easy to see that the weakening relation between MTL formulas is reflexive and
transitive. Some examples are: \( \varphi \) is weaker than \( \varphi \land \psi \), \( \varphi \lor \psi \) is weaker than \( \varphi \) and \( pU_{[0,2.5]}q \) is weaker than \( pU_{[0,2]}q \).

In the sequel of this chapter, we assume that the interpretation of MTL formulas can be over either sequential or concurrent timed state sequences, unless explicitly stated otherwise.

The weakening relation between MTL formulas can be derived by exploiting the weakening relation between their sub-formulas and/or the inclusion relation between their time-bounds. This is discussed in the following subsection.

5.2.1 Weakening relation derived from sub-formulas

Since each non-atomic MTL formula \( \varphi \) is recursively formed by its sub-formulas, it is reasonable to believe that the weakening of its sub-formulas should have a direct impact on formula \( \varphi \). This intuition is formalized in Proposition 5.1, which shows that the weakening of a sub-formula of \( \varphi \) generates a weaker formula than \( \varphi \). For example, let \( p, q, r \) be atomic propositions, and let \( \varphi = pU_{[2,2.5]}r \). By weakening a sub-formula \( p \) of \( \varphi \) to \( p \lor q \), the newly generated formula \( \varphi' = (p \lor q)U_{[2,2.5]}r \) is weaker than \( \varphi \).

**Proposition 5.1** For any MTL formula \( \varphi \), let \( \phi \) be a sub-formula of \( \varphi \). If \( \phi' \) is a weaker formula than \( \phi \) and \( \varphi' \) is the formula obtained by replacing one occurrence of \( \phi \) with \( \phi' \) in \( \varphi \), then \( \varphi' \) is a weaker formula than \( \varphi \).

**Proof** If \( \phi = \varphi \), it is easy to see that \( \varphi' = \phi' \). Hence \( \varphi' \) is a weaker formula than \( \varphi \). If \( \phi \neq \varphi \), \( \varphi \) can be decomposed into two sub-formulas, denoted as \( \varphi_1 \) and \( \varphi_2 \), and \( \phi \) is a sub-formula of either one of them. We can prove the theorem by induction on the structure of \( \varphi \). The proof is not difficult, therefore, we only give one case as an example.

**Case 1:** \( \varphi = \varphi_1 \lor \varphi_2 \). Without loss of generality, assume that \( \phi \) belongs to \( \varphi_1 \). Let \( \varphi'_1 \) be the formula obtained by replacing \( \phi \) with \( \phi' \). By induction we have that \( \varphi'_1 \) is a weaker formula than \( \varphi_1 \). Then, by the interpretation of MTL formulas, it is easy to see that \( \varphi' = \varphi'_1 \lor \varphi_2 \) is a weaker formula than \( \varphi \).

Corollary 5.2 follows directly from Proposition 5.1.

**Corollary 5.2** Given an MTL formula \( \varphi, \varphi' \) can be formed by replacing at least one sub-formula \( \phi \) of \( \varphi \) with a weaker formula than \( \varphi \). Then \( \varphi' \) is a weaker formula than \( \varphi \).

Proposition 5.1 and Corollary 5.2 show that the weakening relation based on sub-formulas behaves compositionally in the “negation-free” MTL logic. However, such a sub-formula-weakening compositionality does not hold for all forms of MTL formulas, which can have operators like negation and implication. For example, \( p \) is weaker than \( p \land q \), but \( p \to r \) is not weaker than \( p \land q \to r \). One solution to the problem is to convert a given MTL formula to its corresponding “negation-free” formula using the rules given in Table 5.1 and 5.2. For example, \( p \to r \) is converted to \( \neg p \lor r \) and \( p \land q \to r \) is converted to \( \neg p \lor \neg q \lor r \). Since \( \neg p \lor \neg q \) is weaker than \( \neg p \), we can derive that \( p \land q \to r \) is weaker than \( p \to r \).
5.2.2 Weakening relation derived from time-bounds

MTL incorporates quantitative timing constraints in its operators, enabling the expression of quantitative real-time properties. These quantitative timing constraints on formulas (properties) can also be used to derive the weakening relation between formulas. For example, in MTL, $pU_{[0,\mu]}q$ specifies a real-time property that $q$ happens within $\mu$ time units after $p$. For different values of $\mu$, formulas have different quantitative timing constraints. Formula $pU_{[0,2.5]}q$ has a weaker constraint on the issuing time of $q$ than formula $pU_{[0,2]}q$. The following lemmas formally state the weakening relation between MTL formulas based on their time-bounds.

**Lemma 5.3** Let $I$ and $I'$ be two time-bounds. Further let $\varphi_1, \varphi_2$ be two MTL formulas. If $I \subseteq I'$, then $\varphi_1 U_{I'} \varphi_2$ is a weaker formula than $\varphi_1 U_I \varphi_2$.

**Proof** By the interpretation of MTL formulas over timed state sequences, the proof is straightforward.

**Lemma 5.4** Let $I$ and $I'$ be two time-bounds. Further let $\varphi_1, \varphi_2$ be two MTL formulas. If $I' \subseteq I$, then $\varphi_1 V_{I'} \varphi_2$ is a weaker formula than $\varphi_1 V_I \varphi_2$.

**Proof** By the interpretation of MTL formulas over timed state sequences, the proof is straightforward.

Here several pairs of MTL formulas are listed where the second formula is weaker than the first formula.

**Example 5.1** Weakening formulas:

- $\varphi \Rightarrow \varphi \lor \psi$;
- $\varphi \land \psi \Rightarrow \varphi$;
- $\varphi V_{[3,2,9]} \psi \Rightarrow \varphi V_{[3,2,7]} \psi$;
- $\varphi U_{[3,2,7]} \psi \Rightarrow \varphi U_{[3,2,9]} \psi$;
- $(\varphi_1 U_{[3,2,7]} \psi_1) \lor (\varphi_2 U_{[2,5]} \psi_2) \Rightarrow (\varphi_1 U_{[3,2,7]} \psi_1) \lor (\varphi_2 U_{[2,5]} \psi_2)$;
- $(\varphi_1 U_{[3,2,7]} \psi_1) \lor (\varphi_2 U_{[2,5]} \psi_2) \Rightarrow (\varphi_1 U_{[3,2,7]} \psi_1) \lor (\varphi_2 U_{[2,5]} (\psi_2 \lor \psi_3))$.

5.3 Weakening functions

In this section, we first define several operators for modifying the size of time-bounds. Following that, we introduce several specific weakening functions over MTL formulas by modifying their time-bounds.
5.3.1 Operators over time-bounds

In the previous section, we have proven that the inclusion relation between time-bounds implies a weakening relation between formulas. Now, we define two pairs of operators over time-bounds for modifying the size of time-bounds.

Operators \( \oplus \) and \( \ominus \)

Informally speaking, for any (non-empty) closed interval \([x, y]\) \(\subseteq\) \(\mathbb{R}\) and time-bound \(I\), \(I \oplus [x, y]\) represents an interval, which has the same form as \(I\) (e.g. both \(I\) and \(I \oplus [x, y]\) are left-closed and right-open) and has the left end-point \(l(I) + x\) and the right end-point \(r(I) + y\) respectively. \(I \ominus [x, y]\) is an interval, which has the same form as \(I\) and has the left end-point \(l(I) - x\) and the right end-point \(r(I) - y\) respectively. Figure 5.2 gives an illustration of both operators, which are formally defined as follows.

**Definition 5.6** For any non-empty interval \([x, y]\) \(\subseteq\) \(\mathbb{R}\) and time-bound \(I\),

\[
I \oplus [x, y] = \bigcup_{t \in I} [t + x, t + y] \\
I \ominus [x, y] = \{ t \mid [t + x, t + y] \subseteq I \}
\]

Several examples are given as follows.

**Example 5.2** Operators \( \oplus \) and \( \ominus \):

- if \(I = [2\pi, 4\pi]\) and \([x, y] = [\pi, 2\pi]\), then \(I \oplus [x, y] = [3\pi, 6\pi]\);
- if \(I = (3, 9)\) and \([x, y] = [-4, 3]\), then \(I \oplus [x, y] = (-1, 12)\);
- if \(I = [1.1, 3]\) and \([x, y] = [-1.5, -0.5]\), then \(I \ominus [x, y] = [2.6, 3.5]\);
- if \(I = [1.5, 3]\) and \([x, y] = [-2, 1]\), then \(I \ominus [x, y] = [3.5, 2] = \emptyset\).

If \(x \leq 0\) and \(y \geq 0\), by the definition of operators \(\otimes\) and \(\ominus\) we can easily prove that for any time-bound \(I\), \(I \subseteq I \oplus [x, y]\) and \(I \ominus [x, y] \subseteq I\).

The following propositions and lemmas reveal the relation between \(I\), \(I \ominus [x, y] \oplus [x, y]\) and \(I \ominus [x, y] \oplus [x, y]\). We first show that \(I = I \ominus [x, y] \oplus [x, y]\).
there are four possibilities:

- non-empty interval $[t', x, t' + y] \subseteq R$, if $I \oplus [x, y] \neq \emptyset$ then there exists a $t' \in R$ such that $[t' + x, t' + y] \subseteq I$ and $t \in [t' + x, t' + y]$.

**Proposition 5.5** Let $I$ be an interval of non-negative reals. For any $t \in I$ and for any non-empty interval $[x, y] \subseteq R$, if $I \oplus [x, y] \neq \emptyset$ then there exists a $t' \in R$ such that $[t' + x, t' + y] \subseteq I$ and $t \in [t' + x, t' + y]$.

**Proof** There are four cases depending on the form of $I$. Here we prove the case that $I$ is left-closed and right-open. The proofs of the other cases are similar.

We know that

$$
\exists t' \in R : ([t' + x, t' + y] \subseteq I) \wedge (t \in [t' + x, t' + y]) \quad (5.1)
$$

$$
\Leftrightarrow \exists t' \in R : (t' + x \geq l(I)) \wedge (t' + y < r(I)) \wedge (t' + x \leq t) \wedge (t' + y \geq t)
\Leftrightarrow \exists t' \in R : (l(I) - x \leq t' < r(I) - y) \wedge (t - x \leq t' \leq t - y)
\Leftrightarrow [l(I) - x, r(I) - y] \cap [t - y, t - x] \neq \emptyset
\quad (5.2)
$$

To check the existence of $t'$ for equation 5.1, we need only check the validation of equation 5.2. Now we assume $[l(I) - x, r(I) - y] \cap [t - y, t - x] \neq \emptyset$. It is easy to see there are four possibilities:

- $[l(I) - x, r(I) - y] = \emptyset$: This implies that $|I| < y - x$. By the definition of $\ominus$ operator, we can see that $I \oplus [x, y] = \emptyset$, which contradicts the fact that $I \oplus [x, y] \neq \emptyset$.
- $[t - y, t - x] = \emptyset$: This contradicts the fact that $x \leq y$.
- $t - y \geq r(I) - y$: This contradicts the fact that $t \in I$.
- $t - x < l(I) - x$: This contradicts the fact that $t \in I$.

Therefore $[l(I) - x, r(I) - y] \cap [t - y, t - x] \neq \emptyset$. Hence, there exists a $t' \in R$ such that $[t' + x, t' + y] \subseteq I$ and $t \in [t' + x, t' + y]$.

**Lemma 5.6** Let $I$ be an interval of non-negative reals. For any non-empty interval $[x, y] \subseteq R$, if $I \oplus [x, y] \neq \emptyset$, then $I \oplus [x, y] \oplus [x, y] = I$.

**Proof** 1) We first prove that $I \subseteq I \oplus [x, y] \oplus [x, y]$. For any $t \in I$, by Proposition 5.5, we know that there exists a $t' \in R$ such that $[t' + x, t' + y] \subseteq I$ and $t \in [t' + x, t' + y]$. By the definition of $\ominus$, we know that $t' \in I \ominus [x, y]$. Furthermore, by the definition of $\oplus$, we know that $[t' + x, t' + y] \subseteq I \oplus [x, y] \ominus [x, y]$. It is easy to see that $t \in I \ominus [x, y] \oplus [x, y]$. Hence $I \subseteq I \ominus [x, y] \oplus [x, y]$. 2) Now we prove that $I \ominus [x, y] \oplus [x, y] \subseteq I$. For any $t \in I \ominus [x, y] \oplus [x, y]$, by the definition of $\ominus$, there exists a $t' \in I \ominus [x, y]$ such that $t \in [t' + x, t' + y]$. At the same time, since $t' \in I \ominus [x, y]$, by the definition of $\ominus$, we know that $[t' + x, t' + y] \subseteq I$. Hence, $t \in I$ and $I \ominus [x, y] \oplus [x, y] \subseteq I$.

Hence, $I = I \ominus [x, y] \oplus [x, y]$.

Now we show that $I = I \oplus [x, y] \ominus [x, y]$.

**Proposition 5.7** Let $I$ be an interval of non-negative reals. For any $t \in R$ and for any non-empty interval $[x, y] \subseteq R$, if $[t + x, t + y] \subseteq I \ominus [x, y]$, then $t \in I$. 
The following propositions and lemmas reveal the relation between similar to operators \( \oplus \), we know that there exist \( t_1, t_2 \in I \) such that \( t_1 + x \leq t + x \) and \( t + x \leq t_2 + x \). That is, \( t_1 \leq t \) and \( t \leq t_2 \). Hence, \( t \in I \).

**Lemma 5.8** Let \( I \) be an interval of non-negative reals. For any non-empty interval \( [x, y] \subseteq R \), \( I \oplus [x, y] \ominus [x, y] = I \).

**Proof** By the definition of operators \( \ominus \) and \( \ominus \), we can easily prove that \( I \subseteq I \oplus [x, y] \ominus [x, y] \). By Proposition 5.7 and the definition of \( \ominus \), it is easy to prove that \( I \ominus [x, y] \oplus [x, y] \subseteq I \).

**Operators \( \otimes \) and \( \ominus \)**

Informally speaking, for any (non-empty) closed interval \( [x, y] \subseteq R^{>0} \) and time-bound \( I \), \( I \otimes [x, y] \) represents an interval, which has the same form as \( I \) and has the left end-point \( (I) \cdot x \) and the right end-point \( r(I) \cdot y \) respectively. \( I \otimes [x, y] \) represents an interval, which has the same form as \( I \) and has the left end-point \( (I) \cdot x \) and the right end-point \( \frac{r(I)}{y} \) respectively.

**Definition 5.7** For any non-empty interval \( [x, y] \subseteq R^{>0} \) and time-bound \( I \),

\[
I \otimes [x, y] = \bigcup_{t \in I} [t \cdot x, t \cdot y]
\]

\[
I \ominus [x, y] = \{ t \mid [t \cdot x, t \cdot y] \subseteq I \}
\]

Several examples are given as follows.

**Example 5.3** Operators \( \otimes \) and \( \ominus \):

- if \( I = [2\pi, 4\pi] \) and \( [x, y] = [1, 2] \), then \( I \otimes [x, y] = [2\pi, 8\pi] \);
- if \( I = (3, \infty) \) and \( [x, y] = [3, 4] \), then \( I \otimes [x, y] = (9, \infty) \);
- if \( I = [1.1, 4.5] \) and \( [x, y] = [0.5, 2] \), then \( I \otimes [x, y] = [2.2, 2.25] \);
- if \( I = [1.5, 3) \) and \( [x, y] = [1, 2] \), then \( I \otimes [x, y] = [1.5, 1.5] = \emptyset \).

Similar to operators \( \oplus \) and \( \ominus \), we can prove that: for any \( x \leq 1 \), \( y \geq 1 \) (\( x, y \in R^{>0} \)), and any time-bound \( I \), \( I \subseteq I \otimes [x, y] \) and \( I \otimes [x, y] \subseteq I \).

The following propositions and lemmas reveal the relation between \( I \), \( I \otimes [x, y] \otimes [x, y] \) and \( I \otimes [x, y] \otimes [x, y] \). The proofs of the propositions and the lemmas are similar to the proofs of Propositions 5.5, 5.7 and Lemmas 5.6, 5.8 and are therefore omitted.

**Proposition 5.9** Let \( I \) be an interval of non-negative reals. For any \( t \in I \) and non-empty interval \( [x, y] \subseteq R^{>0} \), if \( I \otimes [x, y] \not= \emptyset \), then there exists \( t' \in R \) such that \( [t' \cdot x, t' \cdot y] \subseteq I \) and \( t \in [t' \cdot x, t' \cdot y] \).
In the absolute $\epsilon$-weakening function $
abla$.

Lemma 5.10 Let I be an interval of non-negative reals. For any non-empty interval $[x, y] \subseteq R_{\geq 0}$, if $I \otimes [x, y] \neq \emptyset$, then $I \otimes [x, y] \otimes [x, y] = I$.

Proposition 5.11 Let I be an interval of non-negative reals. For any $t \in R$ and for any non-empty interval $[x, y] \subseteq R_{\geq 0}$, if $[t \cdot x, t \cdot y] \subseteq I \otimes [x, y]$, then $t \in I$.

Lemma 5.12 Let I be an interval of non-negative reals. For any non-empty interval $[x, y] \subseteq R_{\geq 0}$, $I \otimes [x, y] \otimes [x, y] = I$.

5.3.2 Absolute $\epsilon$-weakening function

In this subsection, we define a specific weakening function (called the absolute $\epsilon$-weakening function) over MTL formulas.

Definition 5.8 Weakening function $R_a^\epsilon (\epsilon \in R_{\geq 0})$: MTL $\rightarrow$ MTL is recursively defined as

- $R_a^\epsilon(p) = p$
- $R_a^\epsilon(-p) = -p$
- $R_a^\epsilon(\varphi \lor \varphi_2) = R_a^\epsilon(\varphi) \lor R_a^\epsilon(\varphi_2)$
- $R_a^\epsilon(\varphi \land \varphi_2) = R_a^\epsilon(\varphi) \land R_a^\epsilon(\varphi_2)$
- $R_a^\epsilon(\varphi \lor I \varphi_2) = R_a^\epsilon(\varphi) \lor (\varphi_2)$
- $R_a^\epsilon(\varphi \land I \varphi_2) = R_a^\epsilon(\varphi) \land (\varphi_2)$

In the absolute $\epsilon$-weakening function $R_a^\epsilon$, $\epsilon$ is a parameter giving the extent to which a time-bound is elongated (or shrunk). For every $\epsilon \in R_{\geq 0}$, $R_a^\epsilon$ defines a function over MTL formulas. $R_a^\epsilon(\varphi)$ relaxes the quantitative timing constraints in formula $\varphi$ and is called the absolute $\epsilon$-weakened formula of $\varphi$. Since $I \otimes [\epsilon, \epsilon]$ may elongate time-bound $I$ to negative reals, we avoid this by using the intersection of $I \otimes [\epsilon, \epsilon]$ and $[0, \infty)$ in the definition of function $R_a^\epsilon$. Several examples are given in the following.

Example 5.4 Absolute $\epsilon$-weakening functions:

- $R_a^{0.01}(p \lor q) = p \lor q$
- $R_a^{0.5}(p U [1.2, 5]) = p U [1.2, 5] \otimes [0.5, 0.5] = p U [0.7, 5.5]$
- $R_a^{0.5}(p V [1.2, 5]) = p V [1.2, 5] \otimes [0.5, 0.5] = p V [1.7, 4.5]$
- $R_a^{2}(p U [0.2, 5]) = (p U [0.2, 5] \otimes (q V [0.2, \infty] r)) = (p U [0.7] \otimes (q V [2.2, \infty] r))$

Now, we show that formula $R_a^\epsilon(\varphi)$ is indeed weaker than formula $\varphi$.

Theorem 5.13 For any $\epsilon \in R_{\geq 0}$ and $\varphi \in MTL$, $R_a^\epsilon(\varphi)$ is a weaker formula than $\varphi$.

Proof Note that time bound $I$ is an interval of non-negative reals. It is easy to see that $I \subseteq (I \otimes [\epsilon, \epsilon]) \cap [0, \infty)$ and $I \otimes [\epsilon, \epsilon] \subseteq I$. The theorem is proven by induction on the structure of formula $R_a^\epsilon(\varphi)$ and by using Lemma 5.3, Lemma 5.4 and Proposition 5.1.
In general, the larger the value of \( \epsilon \) is, the weaker is formula \( R^\epsilon_a(\varphi) \). For example, for any \( \varphi \in MTL \), \( R^{0.2}_a(\varphi) \) is a weaker formula than \( R^{0.1}_a(\varphi) \). This is formally stated as follows.

**Lemma 5.14** For any \( \epsilon_1, \epsilon_2 \in \mathbb{R}^\geq 0 \) and \( \varphi \in MTL \), \( \epsilon_1 \leq \epsilon_2 \) implies that \( R^{\epsilon_2}_a(\varphi) = R^{\epsilon_2-\epsilon_1}_a(R^{\epsilon_1}_a(\varphi)) \).

**Proof** This is easy to prove by induction on the structure of formula \( R^\epsilon_a(\varphi) \).

**Theorem 5.15** For any \( \epsilon_1, \epsilon_2 \in \mathbb{R}^\geq 0 \) with \( \epsilon_1 \leq \epsilon_2 \) and \( \varphi \in MTL \), \( R^{\epsilon_2}_a(\varphi) \) is a weaker formula than \( R^{\epsilon_1}_a(\varphi) \).

**Proof** Follows directly from Lemma 5.14 and Theorem 5.13.

The absolute \( \epsilon \)-weakening function defines a special relation between real-time properties, which is used to establish property relations between two approximate timed systems in later parts of the thesis. However, we also observe that function \( R^\epsilon_a \) cannot exactly preserve qualitative real-time properties that contain the unless operator. For example, assume \( \varphi = p \lor q \), then \( R^\epsilon_a(\varphi) = p \lor \text{V}_{[0, \infty)} q \). This contradicts the intuition that qualitative properties should be preserved between two timed state sequences, when they have bounded absolute timing differences. To avoid this problem, we propose the absolute \( \epsilon \Box \)-weakening function in the following.

### 5.3.3 Absolute \( \epsilon \Box \)-weakening function

Here, we define another weakening function \( R^\epsilon_a \Box \) over MTL formulas, which relaxes unless (or always \(^2\)) formulas to a less extent than \( R^\epsilon_a \) does. This results in better preservation results for unless (or always) formulas (see Section 7.1.4). The only difference between the definitions of \( R^\epsilon_a \Box \) and \( R^\epsilon_a \) is in the case of unless formulas. The definition is as follows.

**Definition 5.9** Weakening function \( R^\epsilon_a \Box (\epsilon \in \mathbb{R}^\geq 0) \): MTL \( \rightarrow \) MTL is recursively defined as:

\[
R^\epsilon_a \Box (p) = p; \quad R^\epsilon_a \Box (\neg p) = \neg p; \quad R^\epsilon_a \Box (\varphi_1 \lor \varphi_2) = R^\epsilon_a \Box (\varphi_1) \lor R^\epsilon_a \Box (\varphi_2); \quad R^\epsilon_a \Box (\varphi_1 \land \varphi_2) = R^\epsilon_a \Box (\varphi_1) \land R^\epsilon_a \Box (\varphi_2); \quad R^\epsilon_a \Box (\varphi_1 U I \varphi_2) = R^\epsilon_a \Box (\varphi_1) U I_{\Box([-\epsilon, \epsilon]) \cap [0, \infty)} R^{\epsilon} \Box (\varphi_2); \quad R^\epsilon_a \Box (\varphi_1 V I \varphi_2) = \begin{cases} R^\epsilon_a \Box (\varphi_1) V I_{\Box([-\epsilon, \epsilon])} R^\epsilon_a \Box (\varphi_2) & l(I) > 0; \\ R^\epsilon_a \Box (\varphi_1) V I_{\Box[0, \epsilon]} R^\epsilon_a \Box (\varphi_2) & l(I) = 0. \end{cases}
\]

Similar to the absolute \( \epsilon \)-weakening function, we can prove that \( R^\epsilon_a \Box (\varphi) \) is always weaker than \( \varphi \), and that the larger the value of \( \epsilon \) is, the weaker formula \( R^\epsilon_a \Box (\varphi) \) is.

\(^2\)Note that always formulas can be expressed in terms of unless formulas.
Furthermore, we can also prove that formula $R_e^\bullet(\varphi)$ is always weaker than formula $R_a^\bullet(\varphi)$, and that $\square R_e^\bullet(\varphi)$ is weaker than $\square R_a^\bullet(\varphi)$. These are stated in the following proposition and corollary.

**Proposition 5.16** For any $\epsilon \in R^\geq 0$ and $\varphi \in \text{MTL}$, $R_e(\varphi)$ is a weaker formula than $R_a^\bullet(\varphi)$.

**Corollary 5.17** For any $\epsilon \in R^\geq 0$ and $\square \varphi \in \text{MTL}$, $\square R_e^\bullet(\varphi)$ is weaker than $\square R_a^\bullet(\varphi)$.

**Proof** Note that $R_e^\bullet(\varphi)$ is weaker than $R_a^\bullet(\varphi)$. By using Proposition 5.1, the remainder of the proof is straightforward.

### 5.3.4 Relative $[x, y]$-stretching function

The above weakening functions are used to establish property relations between two timed state sequences (timed systems), where the nearness is measured based on the absolute timing difference. For the relative timing difference case, we use a relative $[x, y]$-stretching function presented in this subsection. We define a relative $[x, y]$-stretching function over formulas based on the $\otimes$ and $\odot$ operators.

**Definition 5.10** Let interval $[x, y] \subseteq R^\geq 0$. Relative $[x, y]$-stretching function $R_r^{[x,y]} : \text{MTL} \rightarrow \text{MTL}$ is defined as:

- $R_r^{[x,y]}(p) = p$;
- $R_r^{[x,y]}(\neg p) = \neg p$;
- $R_r^{[x,y]}(\varphi_1 \lor \varphi_2) = R_r^{[x,y]}(\varphi_1) \lor R_r^{[x,y]}(\varphi_2)$;
- $R_r^{[x,y]}(\varphi_1 \land \varphi_2) = R_r^{[x,y]}(\varphi_1) \land R_r^{[x,y]}(\varphi_2)$;
- $R_r^{[x,y]}(\varphi_1 U [x,y] \varphi_2) = R_r^{[x,y]}(\varphi_1) U [x,y] R_r^{[x,y]}(\varphi_2)$;
- $R_r^{[x,y]}(\varphi_1 V [x,y] \varphi_2) = R_r^{[x,y]}(\varphi_1) V [x,y] R_r^{[x,y]}(\varphi_2)$.

Several examples of relative $[x, y]$-stretching formulas are given.

**Example 5.5** Relative $[x, y]$-stretching functions:

- $R_r^{[2,3]}(p \lor q) = p \lor q$;
- $R_r^{[0.5,2]}(p U_{[1.2,5]} q) = p U_{[1,2.5] \otimes [0.5,2]} q = p U_{[0,6,10]} q$;
- $R_r^{[0.5,2]}(p V_{[1,2.5]} q) = p V_{[1,2.5] \odot [\frac{1}{2},\frac{3}{4}]} q = p V_{[2,4,2.5]} q$;
- $R_r^{[1,2]}((p U_{[0.2,5]} q) \land (q V_{[0.2,\infty]} r)) = (p U_{[0,2,10]} q) \land (q V_{[0,4,\infty]} r)$.
It is easy to see that $R_{t}^{[x,y]}(\varphi)$ is not always weaker than $\varphi$. For example, if $\varphi = pU_{(2,4)}q$ and $[x, y] = [1.5, 2]$, then $R_{t}^{[x,y]}(\varphi) = pU_{(3,8)}q$. There is no weakening relation between formulas $pU_{(2,4)}q$ and $pU_{(3,8)}q$. However, under certain conditions, a weakening relation between relative $[x, y]$-stretched formulas does exist. This fact is explained by the following theorem and corollary.

**Theorem 5.18** For any $[x, y] \subseteq R_{t}^{\geq 0}$, $[x', y'] \subseteq R_{t}^{\geq 0}$ and $\varphi \in MTL$, if $[x, y] \subseteq [x', y']$, then $R_{t}^{[x', y']}(\varphi)$ is a weaker formula than $R_{t}^{[x,y]}(\varphi)$.

**Proof** For any time-bound $I$, it is easy to see that $I \otimes [x, y] \subseteq I \otimes [x', y']$ and $I \otimes [x', y'] \subseteq I \otimes [x, y]$. The remainder can be easily proven by induction on the structure of formula $R_{t}^{[x,y]}(\varphi)$ and by using Lemma 5.3, Lemma 5.4 and Proposition 5.1.

**Corollary 5.19** For any $[x, y] \subseteq R_{t}^{\geq 0}$ and $\varphi \in MTL$, if $[1, 1] \subseteq [x, y]$, then $R_{t}^{[x,y]}(\varphi)$ is a weaker formula than $\varphi$.

**Proof** Notice that $\varphi = R_{t}^{[1,1]}(\varphi)$. The proof of the corollary follows directly from Theorem 5.18.

### 5.4 Discussion and summary

In [40], a subclass of MTL formulas (called weakly constrained formulas) was defined with special constraints on the form of the time-bounds of the operators. For example, every until operator in the formula is constrained by an open interval, and every unless operator by a closed interval. An MTL formula can be weakened (or strengthened) to a weakly constrained formula by changing the form of its time-bounds. The weakening and strengthening relation defined in [40] can be considered as a special case of the weakening relation proposed in this chapter.

In this chapter, we formalized real-time properties of a system by the MTL logic. Based on this formalization, we further investigated the weakening relation between MTL formulas, which can be derived from their sub-formulas or their time-bounds. Furthermore, several special weakening functions were also proposed, which can be used to establish property relations between two timed systems in the Chapter 7.
Chapter 6

Tube functions

In Chapter 5 we gave the interpretation of a formula over a timed state sequence. These formulas are used to represent real-time properties of a timed state sequence (timed system). From the interpretation of these formulas, we can see that the satisfaction of a formula by a timed state sequence is completely determined by the timed states in the sequence.

If we are able to establish a quantitative relation between corresponding timed states in two timed state sequences (timed systems), then we can build up a quantitative property relation between these timed state sequences (timed systems). To this end, we propose two parameterized functions in this chapter, each of which establishes a mapping from a time instant \( t \) in \( T \) to a time instant \( t' \) in \( T' \). The parameters of the functions are closely related to the corresponding displacement (\( D'_a \) or \( D'_r \)) from \( T' \) to \( T \) (see Chapter 4). If \( T \) and \( T' \) are used to describe a different timing of the same state sequence, then these functions can also be considered to establish quantitative relations between corresponding timed states in two timed state sequences. We call these functions tube functions. In this chapter, we illustrate that the existence of the tube functions actually gives another characterization of the displacement function defined in Chapter 4, and the properties of the tube functions facilitate the proofs of property-preservation in Chapter 7.

In this chapter, we first introduce an absolute tube function, which establishes a quantitative relation between two sequential time interval sequences based on their absolute timing differences (Section 6.1). This function is called absolute \([x, y]\)-tube function, where parameters \( x \) and \( y \) represent a lower and upper bounds of the absolute timing differences between corresponding time instants in two time interval sequences. Then, the absolute \([x, y]\)-tube function is extended for concurrent time interval sequences (Section 6.2). Similarly, we first introduce relative \([x, y]\)-tube functions for two approximate sequential time interval sequences based on their relative timing differences in Section 6.3. The relative \([x, y]\)-tube functions are also extended for concurrent time interval sequences in Section 6.4. Section 6.5 summarizes this chapter.

\[1\] In a sequential timed system, a timed state is a pair \((\delta, t)\), where \(\delta\) is the observable state at time \(t\). In a concurrent timed system, a timed state is a pair \((\delta, (t, i))\), where \(\delta\) is the observable state at time \((t, i)\).
6.1 Absolute \([x, y]\)-tube function for sequential timed systems

We define an absolute \([x, y]\)-tube function between two sequential time interval sequences as follows.

**Definition 6.1** let \([x, y]\) be a non-empty interval. Further let \(T\) and \(T'\) be two sequential time interval sequences. A function \(F : R^{\geq l(I_0)} \rightarrow R^{\geq l(I'_0)}\) is called an absolute \([x, y]\)-tube function from time interval sequence \(I\) to \(I'\), if and only if it has all of the following properties.

- **Interval consistency:** For every \(t \in R^{\geq l(I_0)}\), \(t \in I_k\) implies \(F(t) \in I'_{k'}\).
- **Bijection:** For all \(t_1, t_2 \in R^{\geq l(I_0)}\), \(t_1 \neq t_2\) implies \(F(t_1) \neq F(t_2)\). Furthermore, for every \(t' \in R^{\geq l(I'_0)}\), there exists a \(t \in R^{\geq l(I_0)}\), such that \(t' = F(t)\).
- **Monotone increasing:** For any \(t_1, t_2 \in R^{\geq l(I_0)}\), \(t_1 > t_2\) implies \(F(t_1) > F(t_2)\).
- **\([x, y]\)-boundedness:** For every \(t \in R^{\geq l(I_0)}\), \(x \leq F(t) - t \leq y\).

Assume that two timed state sequences \(\tau\) and \(\tau'\) share the same state sequence. If an absolute \([x, y]\)-tube function \(F\) can be established between their time interval sequences, then a corresponding relation can also be established between their timed states. This relation has the following characteristics.

- The interval consistency (of \(F\)) guarantees that corresponding timed states in both sequences share the same state (i.e. \(\tau(t) = \tau'(F(t))\)).
- The bijection (of \(F\)) guarantees that for any timed state in one sequence, we can always find its corresponding timed state in the other sequence under function \(F\).
- The monotone increasing (of \(F\)) guarantees that state \(\tau(t_1)\) is observed later than \(\tau(t_2)\) iff state \(\tau(F(t_1))\) is observed later than \(\tau(F(t_2))\).
- The \([x, y]\)-boundedness (of \(F\)) gives a quantitative bound on the “distance” between two timed states \((\tau(t), t)\) and \((\tau(F(t)), F(t))\).

In the following, we investigate the conditions for the existence of the absolute \([x, y]\)-tube function.

**Lemma 6.1** If \(F\) is an absolute \([x, y]\)-tube function from \(T\) to \(T'\), then \(F^{-1}\) is an absolute \([−y, −x]\)-tube function from \(T'\) to \(T\).

**Proof** This is easy to prove by the definition of the absolute \([x, y]\)-tube function.
Example 6.1 Given two time interval sequences $T = I_0 I_1 I_2 \ldots$, where $I_k = [k + 1, k + 2]$ and $k \geq 0$, and $T' = I'_0 I'_1 I'_2 \ldots$, where $I'_{2k} = [2k+0.8, 2k+2.3], I'_{2k+1} = [2k+2.3, 2k+2.8]$, and $k \geq 0$ (shown in Figure 6.1). We can construct a function $F$ (see Figure 6.1) as follows:

$$F(t) = \begin{cases} (2k + 0.8) + \frac{3(t - 2k - 1)}{2}, & \text{if } t \in [2k + 1, 2k + 2); \\ (2k + 2.3) + \frac{t - 2k - 2}{2}, & \text{if } t \in [2k + 2, 2k + 3), \end{cases}$$

where $k \geq 0$. It is easy to see that $F$ is a mapping from $T$ to $T'$. Since $F$ is linear and monotonic for each time interval of $T$, it is not hard to see that $F$ is an absolute $[-0.2, 0.3]$-tube function from $T$ to $T'$. Furthermore,

$$F^{-1}(t) = \begin{cases} (2k + 1) + \frac{2(t - 2k - 0.8)}{3}, & \text{if } t \in [2k + 0.8, 2k + 2.3); \\ (2k + 2) + 2 \cdot (t - 2k - 2.3), & \text{if } t \in [2k + 2.3, 2k + 2.8), \end{cases}$$

where $k \geq 0$. $F^{-1}(t)$ is an absolute $[-0.3, 0.2]$-tube function from $T'$ to $T$.

It should be noticed that for any two time interval sequences and any $x, y \in R (x \leq y)$, an absolute $[x, y]$-tube function does not always exist. For example, there is no absolute $[0.4, 1]$-tube function from $T$ to $T'$ in Example 6.1. The following theorem establishes the necessary and sufficient conditions for the existence of an absolute $[x, y]$-tube function from one time interval sequence to another. First we introduce Lemma 6.2, in which an absolute $[x, y]$-tube function establishes a correspondence between the left-end points of corresponding intervals in both sequences.

**Lemma 6.2** If $F$ is an absolute $[x, y]$-tube function from $T$ to $T'$, then $F(l(I_i)) = l(I'_i)$, for all $i < n(T)$.

**Proof** Suppose $F(l(I_i)) = t$ and $t \neq l(I'_i)$. By the interval consistency property of $F$, it is easy to see that $t > l(I'_i)$. By the bijection property of $F$, for any $l(I'_i) < t' < t$, there exists a $t''$ such that $F(t'') = t'$. By the monotone increasing property of $F$, it is easy to see that $t'' < l(I_i)$. This contradicts the interval consistency property of $F$. 

![Figure 6.1: A $[-0.2, 0.3]$-tube function from $T$ to $T'$](image-url)
and the adjacent property of time interval sequence.

As shown in Figure 6.2, function $G$ satisfies the first three properties of the absolute $[x, y]$-tube function.

Next we show that $G$ also satisfies the $[x, y]_a$-boundedness property.

For any $t \in R^{\geq l(I_k)}$, there exists a time interval $I_k$ and $k \in N$ such that $t \in I_k$. Construct a function $\text{diff}(t) = G(t) - t$. Since $\text{diff}(t)$ is a linear function within interval $I_k$, $\text{diff}(t) \leq \max\{\text{diff}(l(I_k)), \lim_{r(I_k) \to r(I_k)} \text{diff}(t)\}$. By the continuity of $\text{diff}(t)$ and the adjacent property of time interval sequence $T$, it is not hard to see that

$$\min\{\text{diff}(l(I_k)), \text{diff}(l(I_{k+1}))\} \leq \text{diff}(t) \leq \max\{\text{diff}(l(I_k)), \text{diff}(l(I_{k+1}))\}.$$ 

Since $\text{diff}(l(I_k)) = l(I_k') - l(I_k)$ and $D^n_a(T, T') \subseteq [x, y]$, by the definition of $D^n_a$, we can see that $x \leq \text{diff}(t) \leq y$. Hence, $G$ satisfies the $[x, y]_a$-boundedness property.

We have shown that $G$ is an absolute $[x, y]$-tube function from $T$ to $T'$. This completes the proof of Theorem 6.3.
From Theorem 6.3, we can see that the existence of an absolute \([x, y]\)-tube function from time interval sequence \(\mathcal{I}\) to \(\mathcal{I}'\) gives another characterization of the \(D_a\) displacement from \(\mathcal{I}\) to \(\mathcal{I}'\). Since the displacement from timed state sequence \(\tau (\tau = (\mathcal{I}, T'))\) to \(\tau' (\tau' = (\mathcal{I}', T'))\) is equal to the displacement from \(\mathcal{I}\) to \(\mathcal{I}'\), we can see that timed state sequence \(\tau\) is absolute \([x, y]\)-close to \(\tau'\) when \(D_a(\mathcal{I}, \mathcal{I}')\) is a subset of \([x, y]\). This also indicates that if \(\tau\) is absolute \([x, y]\)-close to \(\tau'\) and \([x, y] \subseteq [x', y']\), then \(\tau\) is absolute \([x', y']\)-close to \(\tau'\) too. For example, timed state sequence \(\tau\) is absolute \([-0.1, 0.2]\)-close to \(\tau'\). But \(\tau\) can also be called absolute \([-0.3, 1]\)-close to \(\tau'\).

Based on the definition of the absolute \([x, y]\)-tube of a timed state sequence (Definition 4.7) and Theorem 6.3, it is easy to derive the following corollary.

**Corollary 6.4** For any two timed state sequences \(\tau\) and \(\tau'\) with the same state sequence, \(\tau'\) is absolute \([x, y]\)-close to \(\tau\) iff there exists an absolute \([x, y]\)-tube function from \(\mathcal{I}\) to \(\mathcal{I}'\).

### 6.2 Absolute \([x, y]\)-tube function for concurrent timed systems

For sequential timed systems, we showed in the previous section that the existence of absolute \([x, y]\)-tube functions between time interval sequences implies the same displacement measure as function \(D_a\). In this section, we define the absolute \([x, y]\)-tube function for concurrent timed systems.

In the formalization of concurrent timed systems, due to the need to express simultaneous actions, a singular time interval is allowed to be attached to a state in a timed state sequence. However, a one-to-one mapping cannot be established between a singular interval and a non-singular interval, which is required as one property in the definition for the absolute \([x, y]\)-tube function in the sequential case. Therefore, the absolute \([x, y]\)-tube function defined for the sequential case is not directly applicable for the concurrent case. We need to consider the influence of concurrent behaviors on the constitution of the absolute \([x, y]\)-tube function and propose an alternative definition.

**Definition 6.2** Let \(x, y \in \mathbb{R}\), and further let \(\mathcal{I}\) and \(\mathcal{I}'\) be two labelled time interval sequences, which correspond to time interval sequences of \(\tau\) and \(\tau'\) respectively. \(\mathcal{F}\) is a multi-valued function \(^2\) from \(\mathcal{I}\) to \(\mathcal{I}'\) and \(\mathcal{F}(t, i)\) represents the set of images of time \(\langle t, i \rangle\) \((\langle t, i \rangle \in T^\tau)\). \(\mathcal{F}\) is called an absolute \([x, y]\)-tube function from \(\mathcal{I}\) to \(\mathcal{I}'\), iff the following properties are satisfied.

- **Interval consistency**: For every \(\langle t, i \rangle \in T^\tau, \langle t', i' \rangle \in I_k\) implies that \(\mathcal{F}(t, i) \subseteq I_k'\).
- **Surjection**: For any \(\langle t', i' \rangle \in T^{\tau'}\), there exists some \(\langle t, i \rangle \in T^\tau\), such that \(\langle t', i' \rangle \in \mathcal{F}(t, i)\). That is, the image set of \(T^\tau\) under \(\mathcal{F}\) is \(T^{\tau'}\).

\(^2\) A multi-valued function is a total relation, i.e. every input is associated with one or more outputs [53].
sequences establishes a mapping between their corresponding left-end points. Next, \([t, i) \in \mathcal{F}(t, i_1) < \langle t, i_2 \rangle\) implies that either \(\mathcal{F}(t, i_1) = \mathcal{F}(t, i_2)\) is singular or \(\mathcal{F}(t, i_1) < \mathcal{F}(t, i_2)\) (i.e. for any \((t, i) \in \mathcal{F}(t, i_1)\) and \((t', i') \in \mathcal{F}(t, i_2)\), \((t, i) < (t', i')\)).

- **[x, y]**-boundedness: For every \((t, i) \in \mathcal{T}\) and every \((t', i') \in \mathcal{F}(t, i)\), \(x \leq (t', i') - (t, i) \leq y\).

Following the definition, we can prove that \(\mathcal{F}^{-1}\) is a \([-y, -x]\)-tube function from \(\mathcal{T}^\prime\) to \(\mathcal{T}\).

Before giving the condition for the existence of an absolute \([x, y]\)-tube function between two time interval sequences, we first introduce three lemmas.

**Lemma 6.5** Let \(\mathcal{I}\) and \(\mathcal{I}'\) be two labelled time intervals, and one of them is singular. If \(l(\mathcal{I}') - l(\mathcal{I}) \in [x, y]\) and \(r(\mathcal{I}') - r(\mathcal{I}) \in [x, y]\), then for any \((t', i') \in \mathcal{I}'\) and any \((t, i) \in \mathcal{I}\), \((t', i') - (t, i) \in [x, y]\).

This lemma is based on the fact that the extrema (both maximum and minimum) of the differences between a point and any point in a closed interval are obtained at both ends of the interval.

**Lemma 6.6** Let \(\mathcal{T}\) and \(\mathcal{T}'\) be two labelled time interval sequences. \(\mathcal{D}_k^i(\mathcal{T}, \mathcal{T}') \subseteq [x, y]\) implies that for any \(k < n(\mathcal{T})\), \(r(I_k) < \infty\) implies \(r(I'_k) - r(I_k) \in [x, y]\).

**Proof** For any \(k < n(\mathcal{T})\), let \(I_k\) and \(I'_k\) be unlabelled time intervals of \(\mathcal{I}_k\) and \(\mathcal{I}'_k\) respectively. Notice that for any \(k < n(\mathcal{T})\), \(r(I_k) = l(I_{k+1})\), \(r(I'_k) = l(I'_{k+1})\) and \(r(I'_k) - r(I_k) = r(I'_k) - r(I_k)\). The remainder of the proof is straightforward by the definition of \(\mathcal{D}_k^i\).

This lemma shows that in two time interval sequences, the differences between corresponding left-end points and the differences between corresponding right-end points are always in the same closed interval.

**Lemma 6.7** If \(\mathcal{F}\) is an absolute \([x, y]\)-tube function from \(\mathcal{T}\) to \(\mathcal{T}'\), then \(l(I'_i) \in \mathcal{F}(l(I_i))\) for all \(i < n(\mathcal{T})\) (i.e. for \(i \in N\).

**Proof** Since any form of labelled time intervals is left-closed, it is easy to see that \(l(I'_i) \in \mathcal{I}'_i\) and \(l(I_i) \in \mathcal{I}_i\). Suppose \(l(I'_i) \notin \mathcal{F}(l(I_i))\). By the surjection property and the interval consistency property of \(\mathcal{F}\), there is some \((t, k) \in \mathcal{I}_i\), we know that \(l(I'_i) \notin \mathcal{F}(l(t, k))\). It is easy to see that \((t, k) > l(I_i)\). Furthermore, by the interval consistency property of \(\mathcal{F}\), for any \((t', k') \in \mathcal{F}(l(I_i))\), such that \((t', k') \in \mathcal{I}'_i\), it is easy to see that \((t', k') > l(I'_i)\), which contradicts the weakly monotonic increasing property of \(\mathcal{F}\).

This lemma reveals that an absolute \([x, y]\)-tube function between two time interval sequences establishes a mapping between their corresponding left-end points. Next, we give the condition for the existence of an absolute \([x, y]\)-tube function between two time interval sequences.
**Theorem 6.8** Let $\mathcal{I}$ and $\mathcal{I}'$ be two labelled time interval sequences. $\mathcal{D}^t_\alpha(\mathcal{I}, \mathcal{I}') \subseteq [x, y]$ iff there is an absolute $[x, y]$-tube function from $\mathcal{I}$ to $\mathcal{I}'$.

**Proof** $(\Leftarrow)$ This is not hard to prove by Lemma 6.7 and the $[x, y]$-boundedness property of the absolute $[x, y]$-tube function.

$(\Rightarrow)$ We can first construct function $F_{\mathcal{I}k}$ between corresponding intervals $\mathcal{I}_k$ and $\mathcal{I}_k'$. By enumerating the forms of $\mathcal{I}_k$ and $\mathcal{I}_k'$, we derive three different cases for $F_{\mathcal{I}k}$.

**Case 1:** In this case, both intervals ($\mathcal{I}_k$ and $\mathcal{I}_k'$) are singular. $F_{\mathcal{I}k}$ maps the element of $\mathcal{I}_k$ to that of $\mathcal{I}_k'$. It is easy to see that $F_{\mathcal{I}k}$ is a surjection and weakly monotonic increasing mapping. Furthermore, for every $\langle t, i \rangle \in \mathcal{I}_k$ and every $\langle t', i' \rangle \in F_{\mathcal{I}k}(t, i)$, $x \leq \langle t', i' \rangle - \langle t, i \rangle \leq y$.

**Case 2:** In this case, only one of the intervals ($\mathcal{I}_k$ or $\mathcal{I}_k'$) is singular, while the other is a non-singular left-closed and right-open interval. $F_{\mathcal{I}k}$ can be established by letting the elements of the non-singular interval be the source (or the destination) of the element of the singular interval (as shown in Figure 6.3). It is not hard to check that such a multi-valued function is an absolute $[x, y]$-tube function. It is easy to see that $F_{\mathcal{I}k}$ is a surjection and weakly monotonic increasing mapping. Furthermore, by Lemma 6.6 and 6.5, it is easy to prove that for every $\langle t, i \rangle \in \mathcal{I}_k$ and every $\langle t', i' \rangle \in F_{\mathcal{I}k}(t, i)$, $x \leq \langle t', i' \rangle - \langle t, i \rangle \leq y$.

**Case 3:** In this case, both intervals ($\mathcal{I}_k$ and $\mathcal{I}_k'$) are left-closed, right-open and non-singular. Let $\mathcal{I}_k$ and $\mathcal{I}_k'$ be the unlabelled intervals of $\mathcal{I}_k$ and $\mathcal{I}_k'$ respectively. We can construct a function $F : I_k \rightarrow I'_k$ (as shown in Figure 6.4):

$$F(t) = (t - l(I_k)) \frac{|I'_k|}{|I_k|} + l(I'_k), \quad t \in I_k.$$ 

By Proposition 4.1, we know there are bijections $G_{I_k}$ (from $\mathcal{I}_k$ to $I_k$) and $G_{I'_k}$ (from $\mathcal{I}'_k$ to $I'_k$). Construct a function $F_{\mathcal{I}_k}$ as follows:

$$F_{\mathcal{I}_k}(G_{I_k}^{-1}(t)) = G_{I'_k}^{-1}(F(t)), \quad t \in I_k.$$
It is easy to prove that $F_{\mathcal{I}_k}$ is a surjection and a weakly monotonic increasing mapping. Furthermore, for every $\langle t, i \rangle \in \mathcal{I}_k$ and every $\langle t', i' \rangle \in F_{\mathcal{I}_k}(t, i)$, $x \leq \langle t', i' \rangle - \langle t, i \rangle \leq y$.

Now construct function $F$ as follows:

$$F(i, t) = F_{\mathcal{I}_k}(i, t), \text{ if } \langle i, t \rangle \in \mathcal{I}_k.$$  

It is easy to prove that $F$ is a multi-valued function from $\overline{T}$ to $\overline{T}'$ and that it satisfies the properties of the absolute $[x, y]$-tube function in Definition 6.2.

From Theorem 6.8, we can see that the existence of an absolute $[x, y]$-tube function gives another characterization of the $D^A_k$ displacement from one labelled time interval sequence to another in the concurrent case, which is similar to the sequential case.

### 6.3 Relative $[x, y]$-tube function for sequential timed systems

In the previous sections, we showed that in the absolute timing difference case, an absolute $[x, y]$-tube function exists from the time interval of $\mathcal{T}$ to that of $\mathcal{T}'$, if $\mathcal{T}'$ is absolute $[x, y]$-close to $\mathcal{T}$. In this section, we show that a relative $[x, y]$-tube function can also be constructed from $\mathcal{T}$ to that of $\mathcal{T}'$, if $\mathcal{T}'$ is relative $[x, y]$-close to $\mathcal{T}$.

**Definition 6.3** Let $x, y \in R^{\geq 0}$. Function $F : R^{\geq l(I_0)} \rightarrow R^{\geq l(I'_0)}$ is called a relative $[x, y]$-tube function from time interval sequence $\mathcal{T}$ to $\mathcal{T}'$, iff it has all of the following properties:

- **Interval consistency:** For every $t \in R^{\geq l(I_0)}$, $t \in I_k$ implies $F(t) \in I'_k$.
- **Bijection:** For all $t_1, t_2 \in R^{\geq l(I_0)}$, $t_1 \neq t_2$ implies $F(t_1) \neq F(t_2)$. Furthermore, for every $t' \in R^{\geq l(I'_0)}$, there exists a $t \in R^{\geq l(I_0)}$, such that $t' = F(t)$.
- **Monotone increasing:** For any $t_1, t_2 \in R^{\geq l(I_0)}$, $t_1 > t_2$ implies $F(t_1) > F(t_2)$.
- **$[x, y]_r$-boundedness:** For any $t_1, t_2 \in R^{\geq l(I_0)}$, $t_1 < t_2$ implies $x \cdot (t_2 - t_1) \leq F(t_2) - F(t_1) \leq y \cdot (t_2 - t_1)$.

If $F$ is a relative $[x, y]$-tube function from $\mathcal{T}$ to $\mathcal{T}'$, we can easily prove that $F^{-1}$ is a $[\frac{1}{y}, \frac{1}{x}]$-tube function from $\mathcal{T}'$ to $\mathcal{T}$. Before we introduce the necessary and sufficient conditions for the existence of a relative $[x, y]$-tube function, we give Lemma 6.9 which shows that a relative $[x, y]$-tube function between two time interval sequences establishes a mapping between the left-end points of their corresponding time intervals. The proof of the lemma is similar to its corresponding lemma (Lemma 6.2) in the absolute timing difference case.

**Lemma 6.9** If $F$ is a relative $[x, y]$-tube function from $\mathcal{T}$ to $\mathcal{T}'$, then $F(l(I_i)) = l(I'_i)$, for all $i < n(\mathcal{T})$. 
Relative $[x, y]$-tube function for concurrent timed systems

Theorem 6.10 Let $\bar{T}$ and $\bar{T}'$ be two time interval sequences. There exists a relative $[x, y]$-tube function from $\bar{T}$ to $\bar{T}'$ iff $D^r_r(\bar{T}, \bar{T}') \subseteq [x, y]$.

Proof ($\Rightarrow$) For any $k < n(\bar{T}) - 1$, let $t_1 = l(I_{k+1})$ and $t_2 = l(I_k)$. By Lemma 6.9 and the $[x, y]_r$-boundedness property of the relative $[x, y]$-tube function, it is not hard to prove that $D^r_r(\bar{T}, \bar{T}') \subseteq [x, y]$.

($\Leftarrow$) Construct a function $G : R^{\geq(l_i)} \to R^{\geq(l_i')}$ as follows.

\[ G(t) = (t - l(I_k))\frac{|I_k|}{|I_k|} + l(I_k), \quad t \in I_k. \]

In the case that both $|I_k|$ and $|I_k'|$ are $\infty$, $\frac{|I_k|}{|I_k|}$ is defined as $\frac{x+y}{2}$. It is easy to see that $G$ satisfies the first three properties of the relative $[x, y]$-tube function.

Next we briefly prove that the $[x, y]_r$-boundedness property is also satisfied in this case.

For any $t_1, t_2 \in R^{\geq(l_i)}$ with $t_1 < t_2$, there exist $i$ and $j$ such that $t_1 \in I_i$ and $t_2 \in I_j$. There are two cases.

Case 1: $i = j$. Then, $\frac{G(t_2) - G(t_1)}{t_2 - t_1} = \frac{|I_i|}{|I_i|}$. By the definition of $D^r_r$, $x \leq \frac{G(t_2) - G(t_1)}{t_2 - t_1} \leq y$.

That is, $x \cdot (t_2 - t_1) \leq G(t_2) - G(t_1) \leq y \cdot (t_2 - t_1)$.

Case 2: $i < j$. Then,

\[ G(t_2) - G(t_1) = G(t_2) - G(l(I_j)) + G(l(I_j)) - G(r(I_i)) + G(r(I_i)) - G(t_1). \]

It is easy to prove that:

\[ x \cdot (t_2 - l(I_j)) \leq G(t_2) - G(l(I_j)) \leq y \cdot (t_2 - l(I_j)), \]

\[ x \cdot (l(I_j) - r(I_i)) \leq G(l(I_j)) - G(r(I_i)) \leq y \cdot (l(I_j) - r(I_i)) \]

and

\[ x \cdot (r(I_i) - t_1) \leq G(r(I_i)) - G(t_1) \leq y \cdot (r(I_i) - t_1). \]

By adding up the above three inequations, we have that $x \cdot (t_2 - t_1) \leq G(t_2) - G(t_1) \leq y \cdot (t_2 - t_1)$. This completes the proof.

Theorem 6.10 also shows that the existence of a relative $[x, y]$-tube function from one time interval sequence to another gives another characterization of their $D^r_r$ displacement. Consequently, we can always establish a relative $[x, y]$-tube function from the time interval sequence of $\tau$ to that of $\tau'$, when their $D^r_r$ displacement is a subset of $[x, y]$.

6.4 Relative $[x, y]$-tube function for concurrent timed systems

Similar to the absolute case, in this section, we extend the relative $[x, y]$-tube function for concurrent timed systems. Different from the absolute case, if two labelled time interval sequence are relative $[x, y]$-close ($[x, y] \subset R^{\geq0}$), then any corresponding intervals in both sequences have the same form. That is, either both intervals are singular or they are both left-closed and right-open. If a pair of corresponding intervals in two sequences have different forms (e.g. one is singular, the other is not), it
is easy to see that the relative timing differences between two sequences have either the greatest lower bound 0 or the least upper bound $\infty$.

**Lemma 6.11** For any $x, y \in R^+ \text{ with } x < y$, if $D^i_r(\mathcal{T}, \mathcal{T}') \subseteq [x, y]$, then for any $k < n(\mathcal{T})$ \(|I_k| = 0 \text{ iff } |I'_k| = 0\).

**Proof** Follows from the Definition of $D^i_r$.

Based on Lemma 6.11, different from the absolute case, a one-to-one mapping can be established between any corresponding labelled time intervals in the relative case. The properties of the relative $[x, y]$-tube function defined for the sequential case are also applicable for the concurrent case.

**Definition 6.4** Let non-empty interval $[x, y] \subset R^+ \text{, and further let } \mathcal{T} \text{ and } \mathcal{T}' \text{ be two labelled time interval sequences, which correspond to time interval sequences of } \mathcal{T} \text{ and } \mathcal{T}' \text{ respectively. Let } F \text{ be a function from } \mathcal{T} \text{ to } \mathcal{T}'. F \text{ is called a relative } [x, y] \text{-tube function from } \mathcal{T} \text{ to } \mathcal{T}' \text{ iff the following properties are satisfied.}

- **Interval consistency:** For every $(t, i) \in T^\mathcal{T}, (t, i) \in I_k$ implies that $F(t, i) \in T'_k$.
- **Bijection:** For any $(t_1, i), (t_2, j) \in T^\mathcal{T}, (t_1, i) \neq (t_2, j)$ implies that $F(t_1, i) \neq F(t_2, j)$. Furthermore, for any $(t', i') \in T^\mathcal{T}',$ there exists some $(t, i) \in T^\mathcal{T}$, such that $(t', i') = F(t, i)$.
- **Monotonic increasing:** For any $(t_1, i), (t_2, j) \in T^\mathcal{T}, (t_1, i) < (t_2, j)$ implies that $F(t_1, i) < F(t_2, j)$.
- **$[x, y]_r$-boundedness:** For any $(t_1, i), (t_2, j) \in T^\mathcal{T}$ with $(t_1, i) < (t_2, j), x \cdot (t_2, j) - (t_1, i) \leq F(t_2, j) - F(t_1, i) \leq y \cdot (t_2, j) - (t_1, i)$.

Similar to the sequential case, we have the following lemma and theorem. The proof of the lemma is similar to the sequential case, and is therefore omitted.

**Lemma 6.12** If $F$ is a relative $[x, y]$-tube function from $\mathcal{T}$ to $\mathcal{T}'$, then $l(I'_k) = F(l(I_k))$, for all $i < n(\mathcal{T}) (i \in N)$.

**Theorem 6.13** Let $\mathcal{T}$ and $\mathcal{T}'$ be two labelled time interval sequences. Further let $x, y \in R^+$ with $x < y$. Then $D^i_r(\mathcal{T}, \mathcal{T}') \subseteq [x, y]$ iff there is a relative $[x, y]$-tube function from $\mathcal{T}$ to $\mathcal{T}'$.

**Proof** ($\Rightarrow$) For any $k < n(\mathcal{T}) - 1$, let $(t_1, i) = l(I_{k+1})$ and $(t_2, j) = l(I_k)$. By Lemma 6.11, it is easy to see there are two cases:

- Both $|I_k| = 0$ and $|I'_k| = 0$. This case is not defined in $D^i_r$.
- Both $|I_k| > 0$ and $|I'_k| > 0$. By Lemma 6.12 and the $[x, y]_r$-boundedness property of the relative $[x, y]$-tube function, we then know that $x \cdot |I_k| < |I'_k| < y \cdot |I_k|$.
By the definition of $D_i$ for concurrent timed systems, it is easy to see that $D_i(I, I') \subseteq [x, y]$.

($\Leftarrow$) For any $k < n(\mathcal{T})$, let $I_k$ and $I'_k$ be the unlabelled intervals of $\mathcal{I}_k$ and $\mathcal{I}'_k$ respectively. We can construct a function $F : I_k \rightarrow I'_k$ as follows:

$$F(t) = (t - l(I_k)) \frac{|I'_k|}{|I_k|} + l(I'_k), \quad t \in I_k.$$ 

In the case that both $|I_k|$ and $|I'_k|$ are $\infty$, $\frac{|I'_k|}{|I_k|}$ is defined as $\frac{x+y}{2}$. In the case that both $|I'_k|$ and $|I_k|$ are zero, $\frac{|I'_k|}{|I_k|}$ is defined as 0.

By Proposition 4.1, we know there are bijections $G_{I_k}$ (from $I_k$ to $I_k$) and $G_{I'_k}$ (from $\mathcal{I}'_k$ to $\mathcal{I}'_k$). Construct a function $F_{I_k}$ as follows:

$$F_{I_k}(G_{I_k}^{-1}(t)) = G_{I'_k}^{-1}(F(t)), \quad t \in I_k.$$ 

Since $D_i(I, I') \subseteq [x, y]$, by Lemma 6.11, we know that $\mathcal{I}_k$ and $\mathcal{I}'_k$ have the same form. Based on that, it is easy to prove that $F_{I_k}$ is a bijection and monotonic increasing mapping.

Now construct function $\mathcal{F}$ as follows:

$$\mathcal{F}(i, t) = F_{I_k}(i, t), \quad \text{if} \langle i, t \rangle \in I_k.$$ 

It is easy to prove that $\mathcal{F}$ is a function from $\mathcal{I}$ to $\mathcal{I}'$. Similar to the sequential case, it satisfies the properties of the relative $[x, y]$-tube function in Definition 6.4. This ends the proof of theorem 6.13.

### 6.5 Summary

In this chapter, we introduced two parameterized tube functions between time interval sequences. More specifically, the existence of the absolute $[x, y]$-tube function between the time interval sequences of timed state sequences characterizes the same nearness relationship between timed state sequences as implied by displacement function $D_s$. Different from $D_s$, which establishes a quantitative relation between left end-points of corresponding intervals in two sequences, the absolute $[x, y]$-tube function establishes a quantitative relation between all time points of corresponding intervals. Such a relation facilitates the proof of property-preservation between sequential (or concurrent) timed state sequences (timed systems) in the next chapter. The same conclusion also holds for the relative case.
Chapter 7

Real-time property-preservation between timed systems

In previous chapters, the timing behaviors of (sequential or concurrent) timed systems are formalized as a set of timed state sequences, and the real-time properties of timed systems are formalized as $MTL$ formulas. We investigated nearness relations between timed state sequences and weakening relations between $MTL$ formulas. These results are briefly summarized as follows.

- The nearness between timed state sequences is specified by displacement functions. The displacement from one timed state sequence to another can be measured on the basis of their absolute timing differences or relative timing differences.

- The weakening relation between $MTL$ formulas can be derived from their subformulas or their timing bounds. Several specific functions are defined over $MTL$ formulas that change the size of timing bounds of $MTL$ formulas.

- Tube functions give another characterization of the displacement measure between timed state sequences, which facilitate the proof of the real-time property-preservation in this chapter.

In this chapter, based on the nearness relation between timing behaviors and the weakening relation between real-time properties, we address property-preservation between timed systems. In Section 7.1, we first address property-preservation between sequential timed systems, where their nearness is measured by displacement function $D_a$. In Section 7.2, we extend the property-preservation result of the sequential case to concurrent systems, where the influence of the interleaving semantics on property-preservation is illustrated. In Sections 7.3 and 7.4, we address property-preservation on the basis of relative timing differences for sequential and concurrent timed systems respectively. An example is given in Section 7.5 to demonstrate the application of the property-preservation results in real-time system design. Section 7.6 and Section 7.7 present related work and conclusions.
7.1 Property-preservation based on absolute timing differences: sequential case

In Section 4.2.1, we defined displacement function $D^a_\delta$ to measure the nearness between timed state sequences based on absolute timing differences. More specifically, function $D^a_\delta$ defines the nearness (from timed state sequence $\tau$ to $\tau'$) as a closed interval $[x,y]$, where $x$ (or $y$) represents the greatest lower bound (or the least upper bound) of the delay of state transitions in $\tau$ w.r.t. their corresponding ones in $\tau'$. During the investigation of property-preservation based on absolute timing differences, it is more convenient to use the \textit{span} of the $D^a_\delta$ displacement as a measurement for the nearness between timed state sequences. In the following, we first define the absolute $\epsilon$-spanning distance.

7.1.1 Absolute $\epsilon$-spanning distance

We define an absolute $\epsilon$-spanning distance over timed state sequences, which measures the nearness between two timed state sequences based on the span of absolute timing differences between their corresponding time-stamps. In Example 4.3, $D^a_\delta(\tau_1,\tau_2) = [-0.2, 0.1]$, and we call $\tau_1$ and $\tau_2$ absolute $0.3$-spanning. Now, we define the absolute $\epsilon$-spanning function on set $S^S_{Prop} \times S^S_{Prop}$.

\textbf{Definition 7.1} Given two sequential timed state sequences $\tau$ and $\tau'$, if $D^a_\delta(\tau,\tau') = [x,y]$ then $S^a_\delta : S^S_{Prop} \to S^S_{Prop}$ is defined as follows.

$$S^a_\delta(\tau,\tau') = y - x.$$ 

It is easy to see that $S^a_\delta$ defines the nearness between two time interval sequences as a non-negative real number. Furthermore, we can prove that $S^a_\delta$ is a pseudometric function over set $S^S_{Prop}$. That is, for any $\tau, \tau'$ and $\tau''$ sharing the same state sequence, function $S^a_\delta$ satisfies the following conditions:

- $S^a_\delta(\tau,\tau) = 0$;
- $S^a_\delta(\tau,\tau') \geq 0$;
- $S^a_\delta(\tau,\tau') = S^a_\delta(\tau',\tau)$ (symmetry);
- $S^a_\delta(\tau,\tau'') \leq S^a_\delta(\tau,\tau') + S^a_\delta(\tau',\tau'')$ (the triangle inequality).

In the following, we call two timed state sequences absolute $\epsilon$-spanning iff $S^a_\delta(\tau,\tau') \leq \epsilon$. The relation between the absolute $\epsilon$-spanning distance and the $D^a_\delta$ displacement is given in Lemma 7.1.

\textbf{Lemma 7.1} Let $\tau, \tau' \in S^S_{Prop}$. $\tau$ and $\tau'$ are absolute $\epsilon$-spanning iff for some $[x,y] \subset R$, $D^a_\delta(\tau,\tau') \subseteq [x,y]$ and $y - x \leq \epsilon$.

\footnote{A pseudometric is a generalization of a metric. It can map the distance of two distinct elements to 0. For example, for two distinct elements $\tau$ and $\tau'$, if $D^a_\delta(\tau,\tau') = [1.5, 1.5]$, then $S^a_\delta(\tau,\tau') = 0$.}
Property-preservation based on absolute timing differences: sequential case

Figure 7.1: Property-preservation between timed state sequences based on absolute timing differences

The proof of the lemma is straightforward by the definitions of the $D^a$ function and the absolute $\epsilon$-spanning function. Furthermore, by Corollary 6.4, we can easily prove the following proposition.

**Proposition 7.2** For any two sequential timed state sequences $\tau = (T, T')$ and $\tau' = (T', T')$, $\tau$ and $\tau'$ are absolute $\epsilon$-spanning iff there exists an absolute $[x, y]$-tube function from $T$ to $T'$ and $y - x \leq \epsilon$.

### 7.1.2 Property-preservation between timed state sequences

Since a sequential timed system can be viewed as a set of timed state sequences, we first investigate real-time property-preservation between timed state sequences, the result of which can be easily extended to sequential timed systems.

Based on the absolute $\epsilon$-spanning relation between timed state sequences and the absolute $\epsilon$-weakening relation between MTL formulas, we prove property-preservation between timed state sequences. This can be illustrated in Figure 7.1. Assume $\tau_1$ and $\tau_2$ are absolute $\epsilon$-spanning. If $\tau_1$ satisfies property $P$, then $\tau_2$ satisfies an absolute $\epsilon$-weakened property of $P$. This intuition is made precise in the following lemma and theorem.

Given two time points $t_1$ and $t_2$ in a time interval sequence $T$, the following lemma illustrates the relation between the distance $t_2 - t_1$ and the distance $G(t_2) - G(t_1)$, where $G$ is an absolute $[x, y]$-tube function.

**Lemma 7.3** Let $T$ and $T'$ be two time interval sequences, and let $G$ be an absolute $[x, y]$-tube function from $T$ to $T'$. Let $I$ be an interval of non-negative reals. For any $t_1, t_2 \in R^{\geq l(I)}$, (a) if $t_2 - t_1 \in I$ then $G(t_2) - G(t_1) \in I \oplus [x - y, y - x] \cap [0, \infty)$; (b) if $G(t_2) - G(t_1) \in I \oplus [x - y, y - x]$ then $t_2 - t_1 \in I$.

**Proof** (a) $G(t_2) - G(t_1) = (G(t_2) - t_2) - (G(t_1) - t_1) + (t_2 - t_1)$. By the $[x, y]_a$-boundedness property of $G$, it is easy to see that

$$x - y + (t_2 - t_1) \leq G(t_2) - G(t_1) \leq y - x + (t_2 - t_1).$$

Since $(t_2 - t_1) \in I$, by the definition of the $\oplus$ operator, it is not hard to see that $G(t_2) - G(t_1) \in I \oplus [x - y, y - x]$. By the monotonic increasing property of $G$, we know that $G(t_2) - G(t_1) \geq 0$. Hence, $G(t_2) - G(t_1) \in I \oplus [x - y, y - x] \cap [0, \infty)$. 

}
(b) By Lemma 5.6, we know that for any non-empty set $I \cap [x - y, y - x]$, $I \cap [x - y, y - x] = I$. Notice that $G^{-1}$ is an absolute $[-y, x]$-tube function from $\overline{T}$ to $\overline{T}$. The remainder of the proof is similar to the previous case.

**Lemma 7.4** Let $\tau$ and $\tau'$ be two sequential timed state sequences. Further let $t \in R^{x}_{y}(l_{n})$, $\varphi \in MTL$, and let $G$ be an absolute $[x, y]$-tube function from the time interval sequence of $\tau$ to that of $\tau'$. Then $(\tau, t) \models \varphi$ implies $(\tau', G(t)) \models R_{a}^{u - x}(\varphi)$.

**Proof** We show that $(\tau', G(t)) \models R_{a}^{u - x}(\varphi)$ by induction on the structure of formula $\varphi$.

**Case 1:** $\varphi = p$.

Then, by the definition of function $R_{a}^{u - x}$, $R_{a}^{u - x}(\varphi) = p$. By the interpretation of MTL formulas over timed state sequences, $p \in \tau(t)$. Since $G$ is an absolute $[x, y]$-tube function from the time interval sequence of $\tau$ to that of $\tau'$, and $\tau$ and $\tau'$ share the same state sequence, we know that $\tau(t) = \tau(G(t))$ by the interval consistency property of $G$. Hence, $p \in \tau(G(t))$, and $(\tau', G(t)) \models R_{a}^{u - x}(\varphi)$.

**Case 2:** $\varphi = \neg p$.

The proof is identical to the previous case.

**Case 3:** $\varphi = \varphi_{1} \lor \varphi_{2}$.

Then $(\tau, t) \models \varphi_{1}$ or $(\tau, t) \models \varphi_{2}$. By induction we have $(\tau', G(t)) \models R_{a}^{u - x}(\varphi_{1})$ or $(\tau', G(t)) \models R_{a}^{u - x}(\varphi_{2})$. Therefore $(\tau', G(t)) \models R_{a}^{u - x}(\varphi_{1}) \lor R_{a}^{u - x}(\varphi_{2}) = R_{a}^{u - x}(\varphi)$.

**Case 4:** $\varphi = \varphi_{1} \land \varphi_{2}$.

The proof is similar to the previous case.

**Case 5:** $\varphi = \varphi_{1} \cup \varphi_{2}$.

Then, there is some $t_{2} \in I$, such that $(\tau', t + t_{2}) \models \varphi_{2}$ and for all $0 \leq t_{1} < t_{2}$, $(\tau', t + t_{1}) \models \varphi_{1}$. By induction we have $(\tau', G(t + t_{2})) \models R_{a}^{u - x}(\varphi_{2})$. For any $0 \leq t'_{1} < G(t + t_{2}) - G(t)$, by the bijection and monotone increasing properties of $G$, there is a $0 \leq t'_{1} < t_{2}$, such that $G(t + t'_{1}) = G(t) + t'_{1}$. By induction we have $(\tau', G(t + t'_{1})) \models R_{a}^{u - x}(\varphi_{1})$. Hence $(\tau', G(t) + t'_{1}) \models R_{a}^{u - x}(\varphi_{1})$. By the $[x, y]$-boundedness property of $G$ and Lemma 7.3, it is easy to see that $G(t + t_{2}) - G(t) \in I \oplus [x - y, y - x] \cap [0, \infty)$. Hence, $(\tau', G(t)) \models R_{a}^{u - x}(\varphi_{1}) \cup I \oplus [x - y, y - x] \cap [0, \infty)R_{a}^{u - x}(\varphi_{2}) = R_{a}^{u - x}(\varphi)$.

**Case 6:** $\varphi = \varphi_{1} \cap \varphi_{2}$.

The proof is similar to the previous case, which is briefly given as follows.

By Lemma 7.3, if $G(t + t_{2}) - G(t) \in I \oplus [x - y, y - x]$, then $t_{2} \in I$. There are two possibilities:

- \( (\tau', t + t_{2}) \models \varphi_{2} \). By induction we have $(\tau', G(t + t_{2})) \models R_{a}^{u - x}(\varphi_{2})$, that is $(\tau', G(t) + t'_{2}) \models R_{a}^{u - x}(\varphi_{2})$.

- There is some $0 \leq t_{1} < t_{2}$, such that $(\tau', t + t_{1}) \models \varphi_{1}$. Let $t'_{1} = G(t + t_{1}) - G(t)$. Then, it is not hard to prove that $0 \leq t'_{1} < t'_{2}$. By induction we have that $(\tau', G(t) + t'_{1}) \models R_{a}^{u - x}(\varphi_{1})$.

Hence, $(\tau, G(t)) \models R_{a}^{u - x}(\varphi_{1}) \cup I \oplus [x - y, y - x] \cap [0, \infty)R_{a}^{u - x}(\varphi_{2}) = R_{a}^{u - x}(\varphi)$.

This completes our inductive proof of Lemma 7.4.

Now we can present the major result of this section in Theorem 7.5.
Theorem 7.5 Let $\epsilon \in \mathbb{R}^{>0}$, $t \in \mathbb{R}^{>0}$ and $\varphi \in \text{MTL}$. Further let $\tau$ and $\tau'$ be two absolute $\epsilon$-spanning sequential timed state sequences, then $\tau \models \varphi$ implies $\tau' \models R^\epsilon_a(\varphi)$.

Proof By Lemma 7.1, there exists $[x, y] \subseteq R$ with $y - x \leq \epsilon$ such that $\tau'$ is absolute $[x, y]$-close to $\tau$. By Lemma 6.2 and Lemma 7.4, $\tau' \models R^\epsilon_a(\varphi)$. Since $y - x \leq \epsilon$, by Theorem 5.13 we have $\tau' \models R^\epsilon_a(\varphi)$.

Example 7.1 Consider two timed state sequences $\tau$ and $\tau'$ as shown in Figure 7.2, where $p$, $q$, $r$ and $s$ are atomic propositions. $\tau$ and $\tau'$ are formalized as follows.

$\tau$: $\{(p, [0, 1])\}, \{(p,q, r, [1, 2])\}, \{(r, [2, 3])\}, \{(s, [3, 4])\}, \{\emptyset, [4, \infty)\}$

$\tau'$: $\{(p, [0, 0.9])\}, \{(p,q, r, [1, 2])\}, \{(r, [2, 3, 3])\}, \{(s, [3, 4])\}, \{\emptyset, [4, \infty)\}$

It is easy to calculate that $D^a_b(\tau, \tau') = [-0.1, 0.2]$ and that $S^a_n(\tau, \tau') = 0.3$. Based on the $S^a_n$ distance between $\tau$ and $\tau'$, we can predict real-time properties of $(\tau', G(t))$ from those of $(\tau, t)$. To illustrate this, we present the following examples.

Consider MTL formulas:

1. $\varphi_1 = p \land q$
2. $\varphi_2 = q U_{[0, 1]} r$
3. $\varphi_3 = \square_{[1, 2]} r$

By the interpretation of MTL formulas, we know that all of these formulas are satisfied by $(\tau, 1)$. By Lemma 6.2 and Lemma 7.4, $(\tau', 0.9)$ satisfies the following formulas:

1. $R^0.3_a(\varphi_1) = p \land q$
2. $R^0.3_a(\varphi_2) = q U_{[0, 1.3]} r$
3. $R^0.3_a(\varphi_3) = \square_{[1, 3.1, 7]} r$

For any $\mu < 0.3$, we can see that $R^\mu_a(\varphi_2)$ is not satisfied by $(\tau', 0.9)$. Therefore, $y - x$ is the minimal value that can be guaranteed by Lemma 7.4 for real-time property-preservation on the basis of absolute $\epsilon$-weakened formulas.

Recall Definition 4.7 and Definition 5.3. For any timed state sequence $\tau \in S^a_{prop}$, $\tau^{[x, y]}_{[x, y]}$ is a set containing all timed state sequences $\tau'$, whose absolute $\epsilon$-spanning distance from $(\tau, t)$ is less than or equal to $y - x$. We can easily extend the real-time preservation between two timed state sequences to sets of approximate timed state sequences. This is given in Corollary 7.6.
Corollary 7.6 For any non-empty interval \([x, y]\) ⊆ \(\mathbb{R}^\geq 0\), \(\varphi \in \text{MTL}\) and \(\tau \in S^\text{Sp}_\text{prop}\), if \(\tau \models \varphi\) then \(\tau^{[x,y]}_a \models R^{y-x}_a(\varphi)\).

7.1.3 Real-time property-preservation between timed systems

Until now, we have concentrated on real-time property-preservation between timed state sequences. However, in practice, we often need to analyze real-time properties of a timed system instead of those of a single timed state sequence. Since a timed system consists of a set of timed state sequences, the problem of examining the satisfaction of an MTL formula \(\varphi\) by a timed system \(S\) is equivalent to that of examining its satisfaction by all timed state sequences in \(S\). In this way, real-time property-preservation between timed state sequences can be extended to the analysis of real-time property-preservation between timed systems.

Theorem 7.7 Let \(S\) be a sequential timed system and let \(\varphi\) be an MTL formula. For any \(\epsilon \in \mathbb{R}^\geq 0\), if \(S \models \varphi\), then \(S \models R^\epsilon_a(\varphi)\).

Now we define a proximity measure for timed systems based on absolute timing differences.

Definition 7.2 Let \(S_1\) and \(S_2\) be two sequential timed systems. \(S_2\) is called absolute \(\epsilon\)-close to \(S_1\) iff for any timed state sequence \(\tau\) in \(S_2\), there exists a sequence \(\tau'\) in \(S_1\), such that \(\tau\) and \(\tau'\) are absolute \(\epsilon\)-spanning.

Theorem 7.8 Let \(S_1, S_2\) be two sequential timed systems such that \(S_2\) is absolute \(\epsilon\)-close to \(S_1\). Let \(\varphi\) be an MTL formula. If \(S_1 \models \varphi\), then \(S_2 \models R^\epsilon_a(\varphi)\).

Proof The theorem follows from Definition 5.3 and Theorem 7.5.

Example 7.2 Assume timed system \(S_2\) is absolute 0.3-close to \(S_1\). If \(S_1\) satisfies property \(\Box (\text{req} \rightarrow \Diamond_{[3,4]} \text{ack})\), then we can predict that \(S_2\) satisfies property \(\Box_{[0,3,\infty]} (\text{req} \rightarrow \Diamond_{[2,7,4,3]} \text{ack})\).

We proved that the absolute \(\epsilon\)-weakened formula of \(\varphi\) can be preserved from a timed system to its absolute \(\epsilon\)-close timed systems. However, for certain formulas, better preservation results may be obtained. In the following subsection, we show that for any formula \(\varphi\), its absolute \(\Box\)-weakened formula can be preserved between two absolute \(\epsilon\)-close timed systems. Function \(R^\Box_a\) weakens formula \(\varphi\) to a less extent than function \(R^\epsilon_a\) does, in case that the time-bounds of the unless or always operators start from 0. For example, \(R^\Box_a(\Box p) = \Box p\), while \(R^\epsilon_a(\Box p) = \Box_{[\epsilon,\infty]} p\).

7.1.4 Property-preservation with unless formulas

Here we show that for certain special cases, a better real-time property preservation result can be achieved between timed systems. In Section 5.3.3, we proved that formula \(R^\Box_a(\varphi)\) is stronger than \(R^\epsilon_a(\varphi)\). Now, we prove that \(R^\Box_a(\varphi)\) can be preserved between two absolute \(\epsilon\)-spanning timed state sequences.
First, Lemma 7.9 refines the result proven in Lemma 7.3 for a special case.

Lemma 7.9 Let \( T \) and \( T' \) be two time interval sequences, and let \( G \) be an absolute \([x, y]\)-tube function from \( T \) to \( T' \). Let \( I \) be an interval of non-negative reals and \( l(I) = 0 \). For any \( t_1, t_2 \in R^{\geq(l(I))} \) with \( t_2 \geq t_1 \), if \( G(t_2) - G(t_1) \in I \oplus [0, y - x] \), then \( t_2 - t_1 \in I \).

Proof We know that
\[
t_2 - t_1 = G(t_2) - G(t_1) - (G(t_2) - t_2) + (G(t_1) - t_1) \leq G(t_2) - G(t_1) - x + y.
\]
By Lemma 5.6, we know that \( I \oplus [0, y - x] \oplus [0, y - x] = I \). Since \( G(t_2) - G(t_1) \in I \oplus [0, y - x] \), by the definition of the \( \oplus \) operator, \( G(t_2) - G(t_1) + y - x \in I \).

On the other hand, \( l(I) = 0 \), we have two cases:

- \( 0 \in I \): since \( t_2 \geq t_1 \), \( t_2 - t_1 \geq 0 \), it is easy to see that \( t_2 - t_1 \in I \).
- \( 0 \notin I \): By the definition of the \( \oplus \) operator, we know that \( 0 \notin I \oplus [0, y - x] \) too. Therefore \( G(t_2) \neq G(t_1) \). By the bijection property of \( G \), we know that \( t_2 \neq t_1 \). That is, \( t_2 - t_1 > 0 \). Then, it is easy to see that \( t_2 - t_1 \in I \).

This completes the proof of Lemma 7.9.

Now we show that \( R_{a}^{(y-x) \Box} (\varphi) \) can be preserved between timed state sequences when they are absolute \([x, y]\)-close.

Lemma 7.10 Let \( \tau \) and \( \tau' \) be two sequential timed state sequences. Further let \( t \in R^{\geq(l(I))} \), \( \varphi \in MTL \), and let \( G \) be an absolute \([x, y]\)-tube function from the time interval sequence of \( \tau \) to that of \( \tau' \). Then \( (\tau, t) \models \varphi \) implies \( (\tau', G(t)) \models R_{a}^{(y-x) \Box} (\varphi) \).

Proof This lemma can be proven by induction on the structure of MTL formulas. The proof is very similar to Lemma 7.4, except for the proof of \( \text{unless} \) formulas, the time-bound interval of which starts from 0.

Let \( \varphi = \varphi_1 \lor \varphi_2 \) such that \( l(I) = 0 \). By Lemma 7.9, if \( G(t + t_2) - G(t) \in I \oplus [0, y - x] \), then \( t_2 \in I \). The reminder of the proof is similar to Case 6 of Lemma 7.4.

Theorem 7.11 Let \( \epsilon \in R^{\geq 0} \), \( t \in R^{\geq 0} \) and \( \varphi \in MTL \). Further let \( \tau \) and \( \tau' \) be two absolute \( \epsilon \)-spanning sequential timed state sequences. Then \( \tau \models \varphi \) implies \( \tau' \models R_{a}^{\Box} (\varphi) \).

Proof The proof is similar to that for Theorem 7.5.

Again, we extend real-time property-preservation between timed state sequences to timed systems, which is presented in the following theorem.

Theorem 7.12 Let \( S_1, S_2 \) be two sequential timed systems such that \( S_2 \) is absolute \( \epsilon \)-close to \( S_1 \). Let \( \varphi \) be an MTL formula. If \( S_1 \models \varphi \), then \( S_2 \models R_{a}^{\Box} (\varphi) \).
Notice that formula $R_n^a(\varphi)$ is stronger than formula $R_n(\varphi)$. The real-time property-preservation result proven in this subsection is therefore stronger than that in the previous section.

**Example 7.3** Reconsider Example 7.2. By Theorem 7.12, we can predict that $S_2$ satisfies property $\Box(req \to \Diamond[2,7,4,3]ack)$, which is stronger than property $\Box[0.3,\infty](req \to \Diamond[2,7,4,3]ack)$.

### 7.2 Property-preservation based on absolute timing differences: concurrent case

In this section, we extend the property-preservation result for sequential timed systems to concurrent timed systems. Due to the interleaving semantics incorporated into the behavior formalization and the semantics of the logic, the proofs for the concurrent case cannot be directly derived from those for the sequential case.

To simplify our proof, we first present a corollary, which facilitates the proof for the preservation of \textit{until} formulas. In Section 5.1.2 (the semantics of the \textit{until} operator), we have seen that the validation of an \textit{until} formula on a timed state sequence depends on whether the second time interval is left-closed (see Figure 5.1). To prove the preservation of \textit{until} properties between two timed state sequences, the absolute $[x,y]$-tube mapping between time points in corresponding time interval sequences should ensure that their image sets are always left-closed (that is, $\min\{F(t,i)\}$ exists.). Corollary 7.13 states that there exists an absolute $[x,y]$-tube function such that the image set of any source point has a minimum (note that the absolute $[x,y]$-tube function in the concurrent case is a multi-valued function).

**Corollary 7.13** Let $I$ and $I'$ be two labelled time interval sequences. If $D_n^a(I,I') \subseteq [x,y]$, then 1) there is an absolute $[x,y]$-tube function $F$ from $I$ to $I'$, such that for any $\langle t, i \rangle \in T^I$, $\min\{F(t,i)\}$ exists, and 2) for any $\langle t', i' \rangle \in F^{-1}(t,i)$, $\min\{F^{-1}(t',i')\}$ exists.

**Proof** By the definition of the $D_n^a$ displacement, it is easy to prove that there exists an absolute $[x,y]$-tube function from $I$ to $I'$. By constructing a function $F$ as in the proof of Theorem 6.8, it is easy to see that for any $\langle t, i \rangle \in I$, $\min\{F(t,i)\}$ exists, and for any $\langle t', i' \rangle \in T^I$, $\min\{F^{-1}(t',i')\}$ exists.

Now, we prove property-preservation between concurrent timed systems. Again, real-time property-preservation between timed state sequences is first investigated, which is then extended to timed systems.

**Lemma 7.14** Let $\sigma$ and $\sigma'$ be two concurrent timed state sequences. Further let $\langle t, i \rangle \in T^\sigma$, $\varphi \in \MTL$, and let $F$ be an absolute $[x,y]$-tube function from the labelled time interval sequence of $\sigma$ to that of $\sigma'$. Then $(\sigma, \langle t, i \rangle) \models \varphi$ implies that for any $\langle t', i' \rangle \in F(t,i)$, $(\sigma', \langle t', i' \rangle) \models R_n^{(y-x)}(\varphi)$. 
Property-preservation based on absolute timing differences: concurrent case

Case 6: \( \varphi \). The proofs for the first four cases (1) \( \varphi = p; \) (2) \( \varphi = \neg p; \) (3) \( \varphi = \varphi_1 \lor \varphi_2 \) and (4) \( \varphi = \varphi_1 \land \varphi_2 \) are similar to the corresponding cases in the proof of Lemma 7.4, and are therefore omitted.

Case 5: \( \varphi = \varphi_1 \lor \varphi_2 \). Then there are \( t_2 \in I, j \) \((1 \leq j \leq m_{(t_2, \tau)}^\varphi)\) and \( \langle t, i \rangle \leq \langle t + t_2, j \rangle \), such that \( (\tau, (t_2, \tau)) \models \varphi_2 \), and for all \( t_1 \) and \( k \) that satisfy \( \langle t, i \rangle \leq \langle t_1, k \rangle < \langle t + t_2, j \rangle \), \( (\tau, (t_1, k)) \models \varphi_1 \). By induction we have that \( (\tau', (t_2, j')) \models R_{(x-y)}^y \beta(\varphi_2) \) for any \( \langle t_2, j' \rangle \in F(t + t_2, j, \tau) \).

In case that \( \langle t, i \rangle = \langle t + t_2, j \rangle \), we immediately have that \( (\tau', (t', i')) \models R_{(x-y)}^y \beta(\varphi) \).

In case that \( \langle t, i \rangle < \langle t + t_2, j \rangle \), two possible relations exist between \( F(t + t_2, j, \tau) \) and \( F(t, i) \) according to the increasing mapping property of \( F \).

- \( F(t + t_2, j) = F(t, i) \) and \( F(t, i) \) is singular. Let \( F(t, i) = \{ (t', i') \} \). By induction we have that \( (\tau', (t', i')) \models R_{(x-y)}^y \beta(\varphi_2) \). Now we show that \( 0 \in I \oplus [x - y, y - x] \). By the \([x, y]_a\)-boundedness property of \( F \), \( x < \langle t', i' \rangle < \langle t, i \rangle < y \) and \( x < \langle t', i' \rangle - \langle t + t_2, j \rangle < y \). At the same time, we have \( 0 = t' - t' \geq t_2 - (t' - (t + t_2)) + (t' - t) \). By induction we then have \( (\tau', (t_2, j')) \models R_{(x-y)}^y \beta(\varphi_2) \).

- \( F(t, i) < F(t + t_2, j) \). By Corollary 7.13 we know that \( \min \{ F(t + t_2, j) \} \) exists. Let \( (t_2, j^*) = \min \{ F(t + t_2, j) \} \). It is easy to see that \( \langle t', i' \rangle < \langle t_2, j^* \rangle \). By induction we have that \( (\tau', (t_2, j^*)) \models R_{(x-y)}^y \beta(\varphi_2) \). Since \( F \) is surjective and weakly monotonic, for any \( t_1 \) and \( k' \), \( 1 \leq k' \leq m_{(t_1, \tau)}^\varphi \) that satisfy \( \langle t', i' \rangle < \langle t_1, k' \rangle < \langle t_2, j^* \rangle \), it is easy to prove that there is some \( (t_1', k') \in T_{(x-y)}^y \) such that \( (t_1', k') \in F(t, \tau) \) and \( (t, i) < (t_1', k') < (t + t_2, j) \). By induction we have that \( (\tau', (t_1', k')) \models R_{(x-y)}^y \beta(\varphi_1) \). Similar to the previous case, by the \([x, y]_a\)-boundedness property of \( F \), it is not hard to prove that \( t_2 - t' \in I \oplus [x - y, y - x] \cap [0, \infty) \). Hence, \( (\tau', (t', i')) \models R_{(x-y)}^y \beta(\varphi_1) \).

Case 6: \( \varphi = \varphi_1 \lor \varphi_2 \). The proof is similar to the previous case, which is briefly given as follows.

For all \( t_2 \in I \oplus [x - y, y - x] \), \( 1 \leq j' \leq m_{(t_2, \tau)}^\varphi \) and \( \langle t', i' \rangle < \langle t + t_2, j' \rangle \), we can prove that there exist \( t_2 \in I \) and \( 1 \leq j^* \leq m_{(t_2, \tau)}^\varphi \), such that \( \langle t + t_2, j^* \rangle = \max \{ \min \{ F(t + t_2, j) \} \} \). There are two possibilities:

- \( (\tau, (t + t_2, j^*)) \models \varphi_2 \). By induction we then have \( (\tau', (t + t_2, j')) \models R_{(x-y)}^y \beta(\varphi_2) \).

- There are some \( t_1 \) and \( k \), \( 1 \leq k \leq m_{(t_1, \tau)}^\varphi \), such that \( \langle t, i \rangle \leq \langle t_1, k \rangle < \langle t + t_2, j^* \rangle \), and \( (\tau, (t + t_1, k)) \models \varphi_1 \). Furthermore, we can prove that there exists a \( (t_1', k') \) such that \( \langle t', i' \rangle \leq (t_1', k') < \langle t + t_2, j' \rangle \), where \( 1 \leq k' \leq m_{(t_1', \tau)}^\varphi \). By induction we have \( (\tau', (t_1', k')) \models R_{(x-y)}^y \beta(\varphi_1) \).
Hence, \((\tau, \langle t', i' \rangle)) \models R_a^{(y-x)}(\varphi_1) \lor \langle y, y-x \rangle R_a^{(y-x)}(\varphi_2) = R_a^{(y-x)}(\varphi).

This completes our inductive proof of Lemma 7.14.

We can also define the absolute \(\epsilon\)-spanning distance between concurrent timed state sequences (timed systems) in a similar manner to that in the sequential case. Now we obtain the main result of real-time property-preservation between concurrent timed state sequences in Theorem 7.15.

**Theorem 7.15** Let \(\epsilon \in \mathbb{R}^\geq 0\) and \(\varphi \in \text{MTL}\). Further let \(\tau\) and \(\tau'\) be two absolute \(\epsilon\)-spanning concurrent timed state sequences, then \(\tau \models \varphi\) implies \(\tau' \models R_\epsilon^\varphi(\varphi)\).

**Proof** The proof is similar to that for the corresponding theorem (Theorem 7.5) in the sequential case.

Similarly, real-time property-preservation between concurrent timed state sequences is readily extended to the analysis of real-time properties between concurrent timed systems.

**Theorem 7.16** Let \(S_1\) and \(S_2\) be two concurrent timed systems such that \(S_2\) is absolute \(\epsilon\)-close to \(S_1\). Let \(\varphi\) be an MTL formula. If \(S_1 \models \varphi\), then \(S_2 \models R_\epsilon^\varphi(\varphi)\).

In this section, a property-preservation result was proven for concurrent timed systems, which is the same as that for sequential timed systems. Following a similar line of reasoning, the preservation of until formulas in Section 7.1.4 can also be generalized to the concurrent case, which is shown by the following theorem.

**Theorem 7.17** Let \(S_1\) and \(S_2\) be two concurrent timed systems such that \(S_2\) is absolute \(\epsilon\)-close to \(S_1\). Let \(\varphi\) be an MTL formula. If \(S_1 \models \varphi\), then \(S_2 \models R_\epsilon^\Box(\varphi)\).

### 7.3 Property-preservation based on relative timing differences: sequential case

Till now, we have investigated real-time property-preservation between timed systems based on their absolute timing differences. However, in some cases, absolute time differences cannot effectively measure the nearness between two timed state sequences (timed systems). For example, given two timed state sequences \(\tau\) and \(\tau'\) as follows:

\[
\tau : (\delta_0, [0, 1)) (\delta_1, [1, 2)) \ldots (\delta_i, [i, i+1)) \ldots
\]

\[
\tau' : (\delta_0, [0, 2)) (\delta_1, [2, 4)) \ldots (\delta_i, [2i, 2i+2)) \ldots
\]

The absolute time difference between \(\tau\) and \(\tau'\) is unbounded from the above. Therefore, real-time properties of sequence \(\tau'\) cannot be effectively estimated from those of \(\tau\). To be able to predict real-time properties between these sequences, we need to investigate real-time property-preservation based on relative timing differences.
More specifically, we prove in this section a property-preservation relation between sequential timed state sequences based on the following two functions: the relative \([x, y]\)-tube function between timed state sequences, and the relative \([x, y]\)-stretching function between MTL formulas. The proven result is illustrated in Figure 7.3. If \(\bar{\tau}_2\) is relative \([x, y]\)-close to \(\bar{\tau}_1\) and \(\bar{\tau}_1\) satisfies a formula (property) \(P\), then \(\bar{\tau}_2\) satisfies the relative \([x, y]\)-stretched formula of \(P\). This idea is proven by the following lemma and theorem.

Given two time points \(t_1\) and \(t_2\) in a time interval sequence \(\bar{I}\), the following lemma reveals the relation between the distance \(t_2 - t_1\) and the distance \(G(t_2) - G(t_1)\), where \(G\) is a relative \([x, y]\)-tube function.

**Lemma 7.18** Let \(\bar{\tau}\) and \(\bar{\tau}'\) be two time interval sequences, and let \(G\) be a relative \([x, y]\)-tube \((\[x, y]\) \subset R^{\geq 0})\) function from \(\bar{\tau}\) to \(\bar{\tau}'\). Let \(I\) be an interval of non-negative reals. For any \(t_1, t_2 \in R^{\geq l(I_0)}\) with \(t_2 \geq t_1\),

(a) if \((t_2 - t_1) \in I\) then \((G(t_2) - G(t_1)) \in I \otimes [x, y]\);  
(b) if \((G(t_2) - G(t_1)) \in I \otimes [\frac{1}{y}, \frac{1}{x}]\) then \((t_2 - t_1) \in I\).

**Proof** (a) If \((t_2 - t_1) \in I\), by the definition of operator \(\otimes\), we know that \(\{(t_2 - t_1) \cdot x, (t_2 - t_1) \cdot y\} \subseteq I \otimes [x, y]\). By the \([x, y]\)-boundedness of \(G\), we know that \((t_2 - t_1) \cdot x \leq G(t_2) - G(t_1) \leq (t_2 - t_1) \cdot y\). Therefore, \(G(t_2) - G(t_1) \in I \otimes [x, y]\).

(b) By Lemma 5.10, we know that for any non-empty set \(I \otimes [\frac{1}{y}, \frac{1}{x}]\), \(I \otimes [\frac{1}{y}, \frac{1}{x}] \otimes [\frac{1}{y}, \frac{1}{x}] = I\). Notice that \(G^{-1}\) is a relative \([\frac{1}{y}, \frac{1}{x}]\)-tube function from \(\bar{\tau}'\) to \(\bar{\tau}\). The remainder of the proof is similar to the previous case.

**Lemma 7.19** Let \(\tau\) and \(\tau'\) be two sequential timed state sequences. Further let \(t \in R^{\geq I(N_0)}\), \(\varphi \in MTL\) and let \(G\) be a relative \([x, y]\)-tube function from the time interval sequence of \(\tau\) to that of \(\tau'\). Then \((\tau, t) \models \varphi\) implies \((\tau', G(t)) \models R^{[x, y]}_{t}(\varphi)\).

**Proof** By Lemma 7.18, the proof is similar to that of Lemma 7.4, and is therefore omitted here.

Note that given a relative \([x, y]\)-tube function \(G\) from time interval sequence \(\bar{\tau}\) to \(\bar{\tau}'\), \(G(l(I_0)) = l(I'_0)\). Theorem 7.20 gives the major property-preservation result between timed state sequences based on relative timing differences.

**Theorem 7.20** Let \(\tau\) and \(\tau'\) be two sequential timed state sequences such that \(\tau'\) is relative \([x, y]\)-close to \(\tau\). Further let \(\varphi \in MTL\). Then \(\tau \models \varphi\) implies \(\tau' \models R^{[x, y]}(\varphi)\).
The extension of the preservation result from timed state sequences to timed systems is similar to the absolute case. We first define a proximity measure for timed systems based on relative timing difference.

**Definition 7.3** Let $S_1$ and $S_2$ be two sequential timed systems. $S_2$ is called relative $[x, y]$-close to $S_1$ iff for any timed state sequence $\tau$ in $S_2$, there exists a sequence $\tau'$ in $S_1$, such that $\tau$ is relative $[x, y]$-close to $\tau'$.

**Theorem 7.21** Let $S_1$ and $S_2$ be two sequential timed systems such that $S_2$ is relative $[x, y]$-close to $S_1$. Let $\varphi$ be an MTL formula. If $S_1 \models \varphi$, then $S_2 \models R_{[x,y]}(\varphi)$.

**Example 7.4** Assume that timed system $S_2$ is relative $[0.5, 1.5]$-close to $S_1$, and assume that $S_1$ satisfies a time-bounded response property $\varphi$: $\Box(req \rightarrow \Diamond_{[1.5, 2]}(ack))$. Then we can predict that $S_2$ satisfies the relative $[0.5, 1.5]$-stretched property of $\varphi$, which is $\Box(req \rightarrow \Diamond_{[3, 4]}(ack))$, by Theorem 7.21.

### 7.4 Property-preservation based on relative timing differences: concurrent case

In this section, we extend the property-preservation results in Section 7.3 to concurrent timed systems. Since the proofs for the concurrent case are quite similar to those for the sequential case, here we only list the major results in the following lemma and theorems.

**Lemma 7.22** Let $\tau$ and $\tau'$ be two concurrent timed state sequences. Further let $(t, i) \in T^{\tau}$, $\varphi \in \text{MTL}$, and let $\mathcal{F}$ be a relative $[x, y]$-tube function from the labelled time interval sequence of $\tau$ to that of $\tau'$. Then $(\tau, (t, i)) \models \varphi$ implies that $(\tau', \mathcal{F}(t, i)) \models R_{[x,y]}(\varphi)$.

**Theorem 7.23** Let $\tau$ and $\tau'$ be two concurrent timed state sequences, and let $\tau'$ be relative $[x, y]$-close to $\tau$. Further let $\varphi \in \text{MTL}$. Then $\tau \models \varphi$ implies $\tau' \models R_{[x,y]}(\varphi)$.

We can also define two concurrent timed systems to be relative $[x, y]$-close in the same manner as in the sequential case. Then we have the following preservation result.

**Theorem 7.24** Let $S_1$ and $S_2$ be two concurrent timed systems such that $S_2$ is relative $[x, y]$-close to $S_1$. Let $\varphi$ be an MTL formula. If $S_1 \models \varphi$, then $S_2 \models R_{[x,y]}(\varphi)$.

### 7.5 An example of real-time property preservation

Consider an intelligent light controller, which can adjust light intensity according to different input action sequences. Assume this controller is implemented by real-time software. The timing behavior of the software is depicted in Figure 7.4, where timing constraints are annotated on arcs. The proposition set has four atomic propositions, which have the following meaning:
An example of real-time property preservation

- **Active**: The controller detects the first click when the light is off.
- **Wait**: The light intensity is ready to be increased.
- **Normal**: The light intensity is normal.
- **High**: The light intensity is high.

If a click action is detected at the initial state ($\emptyset$), timer $x$ is activated (or reset) to measure the elapsed time since the first click, and the controller goes into state $\{\text{Active}\}$, after which the controller moves to state $\{\text{Wait}\}$ immediately. If a second click is detected within 2 seconds after the first click, the intensity of the light is set to high. On the other hand, if no click is detected within 2 seconds after the first click (i.e. the timer is out), the intensity of the light is set to normal. Under both observable states $\{\text{Normal}\}$ and $\{\text{High}\}$, if another click is detected, the light is turned off and the controller returns to the initial state.

A real-time property of the software can be described as follows:

$$
\varphi = \square(\text{Active} \rightarrow (\diamond [0,2) \text{High} \lor \diamond [2,\infty) \text{Normal})).
$$

Property $\varphi$ indicates that when a click is detected when the light is off, the light either goes into the **High** intensity within 2 seconds or goes into the **Normal** intensity after 2 seconds. It is not hard to see that this property holds in the model of Figure 7.4.

Assume that due to the non-zero execution time of software, the time-out messages may be received by the controller with a delay up to 0.01 seconds. Consequently, for any execution trace $\tau'$ in this realization, we can always find an execution trace $\tau$ in the model, whose $D_a$ displacement to $\tau'$ is a subset of $[0, 0.01]$. Therefore, according to Theorem 7.17, we can predict that the following real-time property $R_{0.01}^{0.01\diamond} (\varphi)$ is satisfied by the software realization:

$$
\square(\text{Active} \rightarrow (\diamond [0,2,01) \text{High} \lor \diamond [1.99,\infty) \text{Normal})).
$$

Notice that we can also reverse the above reasoning. If our objective is to check the satisfaction of the real-time property $\square(\text{Active} \rightarrow (\diamond [0,2) \text{High} \lor \diamond [2,\infty) \text{Normal}))$ in the realization, we can check the satisfaction of the following stronger property in the model,

$$
\square(\text{Active} \rightarrow (\diamond [0,2-\epsilon) \text{High} \lor \diamond [2+\epsilon,\infty) \text{Normal})),
$$

Figure 7.4: The timing behavior of the control software for intelligent lighting system
where the value of $\epsilon$ is determined by the absolute timing difference between the model and the realization.

Since no hardware clock is perfect, $R_{a, 0.01} \Box (\varphi)$ may not be satisfied by the realization, when it is interpreted in the true physical time domain. Assume that the hardware clock can be faster by up to 0.02% and slower by up to 0.01% w.r.t. the change rate of the true physical time. In this case, for any timed execution trace $\tau'$ in the realization interpreted in the hardware time domain, we can always find an execution trace $\tau$ in the true physical time domain, such that $D^s_{\tau}(\tau, \tau') \subseteq [1 - 0.01, 1 + 0.02]$, or $D^s_{\tau}(\tau', \tau) \subseteq [1 - 0.01, 1 + 0.02]$. Based on Theorem 7.24, we can predict that the following real-time property $R_{a, 0.01} \Box (\varphi)$ is satisfied by the realization when interpreted in the true physical time domain.

$$\Box (Active \rightarrow (\Diamond [0, 2.031) \text{High} \lor \Diamond (1.950, \infty) \text{Normal})).$$

Notice that $R_{a, 0.01} \Box (\varphi)$ is stronger than property

$$\Box (Active \rightarrow (\Diamond [0, 2.031) \text{High} \lor \Diamond (1.950, \infty) \text{Normal})).$$

Therefore, when a click is detected when the light is off, the realization ensures that the light goes into the High intensity state within 2.031 seconds or goes into the Normal intensity after 1.950 seconds in the true physical time domain.

### 7.6 Related work

The property satisfaction of a system can be addressed by the reachability analysis, which checks whether undesired (timed) states are reachable from initial states of the system [80]. With respect to real-time properties, it has been discussed in literature that an infinitesimal timing perturbation can result in property violations where undesired states become reachable. Most of these discussions are carried out within the framework of timed automata [5], where timing perturbations can be contributed by drifts in clocks or timing errors in guards.

In [36], robust timed automata are defined as a subclass of standard timed automata, in which a timed state sequence is always accepted together with all sequences in one of its small neighborhood. The reachability problem in robust timed automata is a subproblem of that in standard timed automata, but real-time properties of robust timed automata are insensitive to an infinitesimal timing error or clock drift. In [80], the authors proposed an algorithm to calculate a set of states $R^*_{\tau}(T, Z_0)$ for a standard timed automata $T$. It contains all states that are reachable from initial states in $Z_0$ on the condition that clocks of $T$ have infinitesimal amounts of drift. By modifying the standard search algorithm of region graphs, the reachability of $R^*_{\tau}(T, Z_0)$ can be decided. As a consequence, $R^*_{\tau}(T, Z_0)$ is robust against infinitesimal drifts in the clocks of $T$. In [27], similar techniques are proposed to address the influence of infinitesimal timing errors in guards of $T$.

The above mentioned research improves the robustness of the standard timed automata w.r.t. infinitesimal timing errors and clock drifts. However, in practice, especially when synthesizing real-time software from a formal model, the timing errors between a model and a realization can not be considered to be infinitesimal.
Therefore, the above mentioned results are not applicable in this situation. The work presented in this chapter addresses this problem by considering the model and its realization as two different timed systems with a certain (non-infinitesimal) absolute or relative timing differences. By investigating real-time property relations between two timed systems, we can predict real-time properties of the realization based on those of the model. Therefore, by using standard verification techniques to verify properties of the model, we can obtain properties of realization based on the property-preservation relation between timed systems.

7.7 Summary

In this chapter, we discussed property-preservation between timed systems. More specifically, two property-preservation results between timed systems are proven based on two different proximity measures (absolute and relative timing differences). Furthermore, the property-preservation results proven in this chapter hold for both sequential and concurrent timed systems. In the next part of this thesis, we apply these property-preservation results to real-time software synthesis.
Real-time property-preservation between timed systems
Part III

Predictable real-time software design

We have demonstrated that platform-independent design approaches have better predictability support for real-time software modelling than platform-dependent design approaches do in Part I of this thesis. However, due to the existence of timing differences between different time domains (i.e. virtual time represented by a system variable, digital hardware time represented by a digital hardware clock and reference time represented by a reference clock), a model and its corresponding realization may exhibit different timing behaviors and satisfy different real-time properties. More specifically, the following timing differences are prominent during real-time software synthesis, and are ineliminable in practice.

- **Absolute timing differences (denoted as timing deviations) between the virtual time domain and the hardware time domain**: A platform-independent model is executed in the virtual time domain, where actions are instantaneous. On the other hand, its realization is deployed on a target hardware/software platform and performed in the digital (or analog) hardware time domain, where the execution time of each action is nonzero and is influenced by optimization techniques, such as caching and pipelining. As a consequence, the timing behavior of the realization differs from the timing behavior of the model in the hardware time domain.

- **Relative timing differences (denoted as timing drifts) between the hardware time domain and the reference time domain**: The drift of a hardware clock can be affected
by a number of physical factors, such as temperature, supply voltage, shock and vibration. As a result, the timing behavior of a system interpreted in the hardware time domain may not be the same as that interpreted in the reference time domain.

In Part II of this thesis, we considered models and their realizations as timed systems. Further, we proved real-time property-preservation between timed systems based on their absolute timing differences and relative timing differences in a formal framework. These property-preservation results, in turn, can serve as a theoretical basis for automatic and correctness-preserving software synthesis in the context of platform-independent design approaches.

In this part of the thesis, we apply the property-preservation results obtained in Part II to real-time software synthesis. In Chapter 8, we apply the property-preservation results to real-time software synthesis and propose a correctness-preserving synthesis approach. In Chapter 9, we design a railroad crossing system using the proposed approach. In Chapter 10, we summarize this thesis and discuss possible directions for future work.
Chapter 8

Towards predictable real-time software synthesis

As indicated in Part I of this thesis, platform-independent design approaches generally provide better predictability support for real-time software modelling than platform-dependent design approaches do. However, they lack sufficient support for a correctness-preserving transformation from a model to its realization during system synthesis. In this chapter of the thesis, we present a predictable platform-independent design approach for real-time software. In contrast to existing platform-independent design approaches, correctness-preserving software synthesis is supported in this approach, which guarantees that the model and its realization satisfy the same qualitative and quantitative real-time properties (up to a small deviation).

In Section 8.1, we reveal the observation time differences of an action between the virtual time domain, the digital hardware time domain, the analog hardware time domain and the reference time domain. These differences can result in inconsistent functionalities or unbounded timing inconsistencies between a model and its realization. To eliminate functional inconsistencies and to bound timing inconsistencies caused by these timing differences, we propose in Section 8.2 two parameterized hypotheses, the absolute $[x, y]$-hypothesis and the relative $[x, y]$-hypothesis based on the absolute timing differences and the relative timing differences between timing behaviors respectively. The satisfaction of these hypotheses can ensure a correctness-preserving transformation from the model to its realization based on the property-preservation results proven in Part II of this thesis. In Section 8.3, we demonstrate that both hypotheses can serve as a foundation for real-time software synthesis. We explain that the absolute $[x, y]$-hypothesis can be incorporated into a synthesis tool Rotalumis, which can automatically transform a POOSL model into an executable realization. The satisfaction of the absolute $[x, y]$-hypothesis can ensure that the correctness of the system is preserved from the virtual time domain to the analog hardware time domain. On the other hand, we illustrate that the relative $[x, y]$-hypothesis is related to the drift of the hardware clock w.r.t. the reference clock. The satisfaction of the relative $[x, y]$-hypothesis can ensure that the correctness of the system is preserved from the analog hardware time domain to the reference time domain.
Section 8.4 discusses some related work, and Section 8.5 summarizes this chapter.

8.1 Bridging different time domains

In the semantics of platform-independent modelling languages, time is represented by a system variable (denoted as the virtual time), and actions are assumed to be instantaneous in this time domain. As a result, the timing aspect of a system can be treated orthogonally to its functionality, which improves the design predictability during system modelling (see Section 2.4.2). However, when models are transformed into their corresponding realizations, the time in the realizations is usually represented by physical time and the assumption of actions to be instantaneous is not valid anymore in this physical time domain. Any action in the realization does require a certain amount of physical time to execute. Due to limited controllability of the activation and execution times of each action, a timing difference exists between the observation times of an action in the virtual time domain and in the physical time domain. As a consequence, the real-time properties satisfied by the model may not hold in the realization.

In the previous chapter, we showed that real-time properties can be predicted between two real-time systems based on their (absolute or relative) timing differences. To establish the timing difference relations between a model and its realization, we first establish a sequence of (absolute or relative) timing differences between the observation times of corresponding actions in the model and in the realization. This sequence of timing differences can facilitate the establishment of the real-time property relations between the model in the virtual time domain and the realization in the physical time domain. Now, we clarify several concepts related to physical time before continuing the discussion on the timing differences between the virtual time domain and the physical time domain.

8.1.1 Several related concepts

In physical reality, time is often counted by clocks. Assume there exists a true physical time, "which flows equably without relation to anything external" [70]. Each clock $C$ is used to measure the progress of the true physical time, and can be considered to be a function which maps each time point $t$ in the true physical time domain $T_t$ to another time domain $T_c$ counted by $C$. In the following, we define several concepts related to clocks.

Definition 8.1 Analog clock:
An analog clock $C_a$ is a function defined as:

$$C_a(t) = \int_0^t f(t') dt',$$

where $f(t')$ is the change rate of $C_a$ at true physical time $t'$ ($t' \in R^>0$).

\footnote{The observation time of an action refers to the moment that the action is completed.}
Bridging different time domains

\[ C_d(t) = i \cdot \Delta, \quad \text{if} \quad t \in \left[ \sum_{j=0}^{i} p(j), \sum_{j=0}^{i+1} p(j) \right). \]

Figure 8.1: Several clock examples.

**Definition 8.2 Digital clock:**

A Digital clock \( C_d \) is a function defined as:

\[ C_d(t) = i \cdot \Delta, \quad \text{if} \quad t \in \left[ \sum_{j=0}^{i} p(j), \sum_{j=0}^{i+1} p(j) \right). \]

Here \( \Delta \) is the step-width of the clock, which refers to the time elapsed between successive ticks in the digital time domain, and \( p(i) \) represents the true physical time progress during the \( i \)-th step of the clock.

For brevity, we use \( s(i) \) to abbreviate \( \sum_{j=0}^{i} p(j) \) in the sequel.

In Figure 8.1, we give several clock examples.

- \( C_p(t) = t \). \( C_p \) is a perfect physical clock, and its change rate is \( f(t) = 1 \).
- \( C_a(t) = \frac{2}{3}t^{\frac{3}{2}} \). \( C_a \) is an analog clock, and its change rate is \( f(t) = t^{\frac{3}{2}} \).
- \( C_d(t) = i \) when \( t \in [1.5i, 1.5(i + 1)] \). \( C_d \) is a digital clock. In this example, \( \Delta = 1 \) and \( p(i) = 1.5 \) for all \( i \), indicating that the true physical time advances 1.5 seconds when clock \( C_d \) advances 1 second.

**Clock deviation:** For two clocks \( C_1 \) and \( C_2 \), the deviation of \( C_2 \) w.r.t. \( C_1 \) at true physical time \( t \) is defined by the following function:

\[ E_{(C_1,C_2)}^a(t) = C_2(t) - C_1(t). \]

For example, in Figure 8.1, the clock deviation of \( C_a \) (and \( C_d \)) w.r.t. \( C_p \) is calculated by \( E_{(C_p,C_a)}^a(t) = C_a(t) - C_p(t) = \frac{2}{3}t^{\frac{3}{2}} - t \) and \( E_{(C_p,C_d)}^a(t) = i - t \), when \( t \in [1.5i, 1.5(i + 1)] \).

**Clock drift:** For two analog clocks \( C_1 \) and \( C_2 \) with change rates \( f_1(t) \) and \( f_2(t) \) respectively, the drift of \( C_2 \) w.r.t. \( C_1 \) at true physical time \( t \) is defined by the following function:

\[ E_{(C_1,C_2)}^r(t) = \frac{f_2(t)}{f_1(t)}. \]
For example, in Figure 8.1, \( E_{(C,p,G_a)}^r(t) = \frac{t^1}{t^1} = t^1 \). Here, we only formally define the drift between two analog clocks. This is due to the fact that the discretization of digital clocks often leads to unbounded drift between clocks. The change rate of a digital clock is either 0 within a step or \( \infty \) at discretization points. The ratio between the change rates of a digital clock and that of an analog clock can be infinitely large. For the same reason, we also leave the drift between two digital clocks undefined.

### 8.1.2 Timing differences between various time domains

Based on the clock concepts introduced in the previous section, we can address the difference between observation times of an action in the virtual time domain and in the physical time domain. As we have mentioned previously, the real-time properties of a system are often analyzed based on its model in the virtual time domain. In reality, different constituents of a real-time system may be timed by different clocks. For example, the real-time controller part is usually timed by a digital hardware clock, while the physical environment part is usually timed by an analog physical clock. In general, a specific reference clock is often chosen, such that real-time properties of different constituents can be checked in a uniform time domain of interest (denoted as the reference time domain).

Therefore, to reason about the real-time properties of a realization based on those of its model, we first establish a proper relation between the observation times of an action in the virtual time domain and in the reference time domain. In this section, we achieve this by building a sequence of (absolute and relative) timing differences between several time domains.

To simplify the following discussion, we assume that the realization of the controller is timed by a digital hardware clock and the reference time is analog. These assumptions hold for many real-time systems, where the computational device running the controller software is driven by a digital hardware clock and where the physical environment is timed by an analog physical clock. Furthermore, the result based on these assumptions can be easily adapted to other situations, e.g. when the hardware clock is analog or the reference clock is digital.

In the following, we explore the timing differences between the virtual time domain and the reference time domain by taking several steps (see Figure 8.2). More specifically, we first explore the absolute timing differences between the virtual time domain and the digital hardware time domain. Then, we address the timing differences between the digital hardware time domain and the analog reference time domain, which are further decomposed into the absolute timing differences between the digital hardware time domain and its (auxiliary) analog hardware time domain, and the relative timing differences between the analog hardware time domain and the analog reference time domain.
Figure 8.2: The timing relation between different time domains

Timing differences between the virtual time domain and the digital hardware time domain

A platform-independent model is executed in the virtual time domain, where actions are instantaneous. On the other hand, the realization of a system is deployed on a target execution platform. Time on this execution platform is usually counted by a digital hardware clock, and the execution of any action in this digital hardware time domain needs to take up a non-zero number of clock cycles. Due to uncontrollable physical time, the activation time of each action cannot always be infinitely accurate, and due to the techniques that are used to boost the average computing performance of the platform, such as caching, pipelining and memory management, the exact execution time of actions in this time domain can be also unpredictable.

For any action, an absolute timing difference (denoted as timing deviation) exists between its virtual time in the model and its digital hardware time in the realization. For example in Figure 8.2, if action α is observed at virtual time $t_2$, we can only guarantee that α is observed within some interval $[t_2 + d_1, t_2 + d_2]$ ($d_1 \leq d_2$) in the digital hardware time domain. The values of $d_1$ and $d_2$ can vary for different actions, different execution platforms and different real-time schedulers.
Towards predictable real-time software synthesis

Timing differences between the digital hardware time domain and the reference time domain

No digital hardware clock is perfectly accurate w.r.t. the analog reference clock. The accuracy of a digital hardware clock can be affected by discretization and a number of physical factors such as temperature, supply voltage, shock and vibration. As a result, the timing of actions in a realization based on a digital hardware clock is not identical to that based on an analog reference clock. Since the absolute timing differences between them can be unbounded during execution, it is not effective to measure these timing differences based on their absolute differences.

An intuitive solution to this problem is to use the ratio between the change rates of clocks to measure the observation time differences between the digital hardware time domain and the reference time domain. However, this solution is not directly applicable, since the ratio between the change rate of a digital clock and that of an analog clock is unbounded (see the clock drift discussion in Section 8.1.1). As a consequence, we cannot use the absolute timing difference or the relative timing difference alone to effectively establish the relation between the observation times of actions in the digital time domain and in the reference time domain. To solve this problem, we propose the following solution.

The basic idea of the solution is that we can construct an auxiliary analog hardware clock (analog hardware clock in short) $C_{ah}$, which has a bounded deviation w.r.t. digital hardware clock $C_{dh}$ and has a bounded drift w.r.t. analog reference clock $C_{ar}$. Then the observation time of an action in the digital time domain and that in the reference time domain can be bridged by this auxiliary clock. This idea is illustrated in more detail in the following.

Based on Definition 8.1, analog reference clock $C_{ar}$ is defined as

$$C_{ar}(t) = \int_{0}^{t} f(t')dt',$$

where $f(t')$ represents the clock change rate at the true physical time $t'$. Similarly,
based on Definition 8.2, digital hardware clock $C_{dh}$ can be represented as function

$$C_{dh}(t) = i \cdot \Delta, \text{ if } t \in [s(i), s(i + 1)),$$

where $s(i) (i \in N)$ represents the true physical time progress for the first $i$ steps of clock $C_{dh}$, and $\Delta$ is the step-width of clock $C_{dh}$.

During the true physical time period $[s(i), s(i + 1)]$, the progress of clock $C_{dh}$ is $\Delta$, and the progress of clock $C_{ar}$ is $\int_{s(i)}^{s(i+1)} f(t')dt'$. We denote the ratio between the progress of two clocks during the true physical time period $[s(i), s(i + 1)]$ as the drift factor $r_i$ of clock $C_{dh}$ w.r.t. clock $C_{ar}$, which can also be considered as the ratio between their average clock change rates during the same period. More specifically, $r_i$ is defined as

$$r_i = \frac{\Delta}{\int_{s(i)}^{s(i+1)} f(t')dt'}.$$

If $r_i > 1$, then the average change rate of the digital clock is faster than that of the analog reference clock during true physical time period $[s(i), s(i + 1)]$ (or during the $i$-th step of $C_{dh}$). Furthermore, $r_i$ can vary for different steps of $C_{dh}$. In practice, $r_i$ is usually bounded.

Now we can construct an (auxiliary) analog clock $C_{ah}$ for digital clock $C_{dh}$ w.r.t. analog clock $C_{ar}$, the change rate of which at the true physical time $t$ is defined by

$$f_{ah}(t) = r_i \cdot f(t), \text{ if } t \in [s(i), s(i + 1)),$$

where $r_i$ is the drift factor of $C_{dh}$ w.r.t. $C_{ar}$ during its $i$-th step, and $f(t)$ is the change rate of clock $C_{ar}$ at true physical time $t$. Consequently, when $t \in [s(i), s(i + 1))$, we have

$$C_{ah}(t) = \int_0^t f_{ah}(t')dt' = \int_0^{s(i)} f_{ah}(t')dt' + \int_{s(i)}^t f_{ah}(t')dt' = \sum_{j=i}^{i+1} \int_{s(j-1)}^{s(j)} f(t')dt' \cdot f(t)dt + r_i \cdot \int_{s(i)}^t f(t')dt' \cdot .$$

$$= \Delta \cdot i + r_i \cdot \int_{s(i)}^t f(t')dt'.$$

Figure 8.3 shows the relation between digital hardware clock $C_{dh}$, analog reference clock $C_{ar}$ and analog hardware clock $C_{ah}$. We can deduce that both the clock deviation of clock $C_{ah}$ w.r.t. clock $C_{dh}$ and the clock drift of clock $C_{ar}$ w.r.t. clock $C_{ah}$ are bounded. More specifically, for $t \in [s(i), s(i + 1))$, the clock deviation from $C_{ah}$ to $C_{dh}$ is:

$$E_{(C_{ah, C_{dh}})}^a(t) = \Delta \cdot i + r_i \cdot \int_{s(i)}^t f(t')dt' - \Delta \cdot i = r_i \cdot \int_{s(i)}^t f(t')dt'.$$

Since $0 \leq r_i \cdot \int_{s(i)}^t f(t')dt' \leq r_i \cdot \int_{s(i)}^{s(i+1)} f(t')dt' = \Delta$, then it is easy to see that $0 \leq E_{(C_{ah, C_{dh}})}^a \leq \Delta$. Similarly, for $t \in [s(i), s(i + 1))$, the clock drift from $C_{ar}$ to $C_{ah}$ is:

$$E_{(C_{ah, C_{ar}})}^r(t) = \frac{f(t)}{f_{ah}(t)} = \frac{1}{r_i},$$
where $\frac{1}{t_i}$ is the ratio between the time progress of clock $C_{ar}$ and the time progress of clock $C_{dh}$ during the $i$-th step of $C_{dh}$. In practice, $\frac{1}{t_i}$ is bounded for all $i \in N$.

Let us have a look at the observation times of action $\alpha$ (shown in Figure 8.2) in the analog hardware time domain and the analog reference time domain. We have mentioned that action $\alpha$ is observed in time interval $[t_2 + d_1, t_2 + d_2]$ in the digital hardware time domain. Since the timing deviation from clock $C_{ah}$ to clock $C_{dh}$ is always in interval $[0, \Delta]$, we can safely say that $\alpha$ is observed in time interval $[t_2 + d_1, t_2 + d_2 + \Delta]$ in the analog hardware time domain. Assume that $\frac{1}{t_i}$ lies in interval $[r_1, r_2]$ meaning that clock $C_{ar}$ is at most $r_2$ faster and at least $r_1$ faster than clock $C_{dh}$ during a step of $C_{dh}$. In this case, we can predict that the observation time of action $\alpha$ should be in time interval $[(t_2 + d_1) \cdot r_1, (t_2 + d_2 + \Delta) \cdot r_2]$ in the reference time domain.

**Example 8.1**, Consider a controller for a flash board showing 4 consecutive letters "IEEE" (similar to Example 2.4). One possible solution to the controller design is to use three parallel processes $I$, $E$ and $S$. $I$ emits letter $I$ every 0.3 seconds, $E$ emits letter $E$ every 0.1 seconds and $S$ issues four blank spaces every 0.3 seconds to erase the letters. The three processes start from 0.01, 0.02 and 0.25 seconds respectively.

The timing behavior of the controller model can be depicted in Figure 8.4, where $x$ is a logic clock used to express timing constraints on actions and $x := 0$ indicates the reset of $x$ to 0.

Suppose the third action of the controller (sending out the second letter 'E') in the digital hardware time domain is observed in time interval $[0.12, 0.12 + 0.001]$. Further, assume the speed of the digital hardware clock $C_{dh}$ to be 100MHz (the stepwidth of $C_{dh}$ is $10^{-8}$ seconds), and each step of clock $C_{dh}$ can drift up to 0.1% (faster or slower) w.r.t. the analog reference clock $C_{ar}$. In other words, during each step of clock $C_{dh}$, the average change rate of clock $C_{ar}$ is at most $\frac{1000}{999}$ and at least $\frac{1000}{1001}$ faster than that of clock $C_{dh}$. In the following, we estimate the observation time of the action in the reference time domain.

- Assume clock $C_{ah}$ is the auxiliary analog hardware clock of $C_{dh}$. For any $t$, we then have $0 \leq E_{(C_{ah}, C_{ah})}^{a}(t) < 10^{-8}$. Hence we can predict that the action is observed in time interval $[0.12, (0.12 + 0.001) + 10^{-8}]$ in the analog hardware domain, which is counted by clock $C_{ah}$.

- By the construction of clock $C_{ah}$, we know that $E_{(C_{ah}, C_{ar})}^{a}(t)$ equals the ratio between the average change rates of clock $C_{dh}$ and $C_{ar}$ during each step of $C_{dh}$. In this case, $\frac{1000}{999} \leq E_{(C_{ah}, C_{ar})}^{a}(t) \leq \frac{1000}{1001}$. Therefore, we can predict that the action is observed in time interval $[0.12 \times \frac{1000}{999}, (0.12 + 0.001 + 10^{-8}) \times \frac{1000}{999}]$ in the analog reference time domain.
8.2 Hypotheses for real-time software synthesis

In the previous section, we have established the relation between the observation times of an action in the virtual time domain and in the reference time domain. This is accomplished by building a sequence of absolute and relative timing differences between several time domains (see Figure 8.2). In this section, we further investigate the influence of these timing differences on the real-time property relations between the model in the virtual time domain and the realization in the reference time domain.

More specifically, two parameterized hypotheses are proposed to bound the (absolute and relative) timing differences between the model and the realization during system synthesis. We show that real-time properties of the realization can be predicted from those of the model based on the parameters of the hypotheses, if both hypotheses are satisfied during the synthesis. Furthermore, the correctness of the realization can be guaranteed on the basis of property prediction.

8.2.1 Absolute \([x, y]\)-hypothesis

We showed that an action, which is observed at time \(t\) in the virtual time domain, is observed within time interval \([t + t_1, t + t_2]\) in the digital hardware time domain and within time interval \([t + t_1, t + t_2 + \Delta]\) in the analog hardware time domain. Without careful treatment on these timing deviations of actions, the deviation between the observation times of an action in the virtual time domain and in the analog hardware time domain can accumulate without bounds during system execution (see Example 2.3). Even worse, these timing deviations may lead to a different execution order of actions between the realization and the model, which may result in faulty behaviors (see Example 2.4).

To avoid the above problem, we propose the *absolute \([x, y]\)-hypothesis* to bound the absolute timing difference between the model in the virtual time domain and the realization in the analog hardware time domain. By doing so, we can predict the real-time properties of the realization (in the analog hardware time domain) from those of the model. This is elaborated on in the following.

We can use a timed action sequence to represent an execution of a real-time system (either a model or a realization). Each timed action sequence can be considered as a sequence of timed actions \((\alpha_i, t_i)\), where \(t_i\) is the observation time of action \(\alpha_i\). A timed action sequence can alternatively be represented as a pair of sequences, an action sequence \((\alpha_0, \alpha_1, \alpha_2, ...)\) and a time sequence \((t_0, t_1, t_2, ...)\).

The absolute \([x, y]\)-hypothesis requires that for any timed action sequence \(\tau_{\alpha}\) of the realization in the analog hardware time domain, the following conditions are satisfied.
1. There exists a timed action sequence $\tau_\alpha$ of the model in the virtual time domain, such that $\tau_\alpha$ and $\tau_\alpha'$ share the same action sequence $\alpha$.

2. For each action in $\alpha$, the deviation between its observation times in $\tau_\alpha$ and in $\tau_\alpha'$ should be within interval $[x, y]$. Parameters $x$ and $y$ in the hypothesis represent a lower and an upper bound of absolute timing difference between two time domains respectively.

Now, we explain how the absolute $[x, y]$-hypothesis can ensure correctness preservation between the virtual time domain and the analog hardware time domain.

In Chapter 7, we introduced that real-time property relations between two timed state sequences can be established based on their absolute timing differences. Those relations reveal that real-time properties can be preserved (up to a small deviation $y - x$) between two absolute $[x, y]$-close timed state sequences. The absolute $[x, y]$-hypothesis assumes that timed action sequences in the analog hardware time domain are $[x, y]$-close to those in the virtual time domain based on their absolute timing differences. Therefore, to ensure that the absolute $[x, y]$-hypothesis can preserve the correctness between the timing behavior in the virtual time domain and that in the analog hardware time domain, we need to examine the property-preservation between timed action sequences. This is achieved by the following line of reasoning.

- **Convert timed action sequences to timed state sequences:**
  A timed execution trace of a system can be represented by either a timed state sequence or a timed action sequence. It is easy to encode one form of representation into the other. One possibility to encode a timed action sequence into a timed state sequence is by letting the observation of each action be an instantaneous state, and inserting a new duration state $\Phi$ between any two instantaneous states [71]. For example, suppose a timed action sequence $\tau_\alpha$ is

  $$(\alpha_1, t_1)(\alpha_2, t_2)(\alpha_3, t_3)\ldots(\alpha_i, t_i)\ldots,$$

  where $t_i$ represents the observation time of $\alpha_i$. Then the corresponding timed state sequence $\tau_\delta$ is

  $$(S_{\alpha_1}, [t_1, t_1])(\Phi, [t_1, t_2])(S_{\alpha_2}, [t_2, t_2])\ldots(S_{\alpha_i}, [t_i, t_i])(\Phi, [t_i, t_i+1])\ldots.$$

  Here $S_{\alpha_i}$ is a state containing the single atomic proposition “$\alpha_i$ is observed” and $\Phi$ is a state at which no atomic proposition is observed. Singular interval $[t_i, t_i]$ indicates that the duration of state $S_{\alpha_i}$ is instantaneous, and interval $[t_i, t_{i+1}]$ represents the duration of state $\Phi$ between $S_{\alpha_i}$ and $S_{\alpha_{i+1}}$.

- **Relate the proximity between timed action sequences and the proximity between their corresponding timed state sequences:**
  Let $\tau_\alpha$ and $\tau_\alpha'$ be two timed action sequences, which share the same action sequence $\alpha$. Assume $\tau_\delta$ and $\tau_\delta'$ to be their corresponding timed state sequences. Then $\tau_\delta$ and $\tau_\delta'$ share the same state sequence. Furthermore, for any action $\alpha_i$ in $\tau_\delta$, if the deviation from $t_i$ to $t_i'$ is in interval $[x, y]$, we can easily derive that $\tau_\delta'$ is absolute $[x, y]$-close to $\tau_\delta$ (see the definition of absolute $[x, y]$-displacement in Section 4.2.1). Consequently, by Definition 7.2, we know that if the absolute
Hypotheses for real-time software synthesis

If the \([x, y]\)-hypothesis is satisfied, then the realization represented by a set of timed state sequences in the analog time domain is absolute \((y - x)\)-close to its model in the virtual time domain.

- **Preserve real-time properties between timing behaviors:**

  By Theorem 7.17, if the realization is absolute \((y - x)\)-close to its model, and the model satisfies MTL property \(P\), then the realization satisfies property \(R_{\alpha}^{(y-x)\Box}(P)\). Property \(R_{\alpha}^{(y-x)\Box}(P)\) has the same form as \(P\), but its quantitative timing bounds have an absolute deviation of \(y - x\) from those of \(P\). Hence, real-time properties hold in the virtual time domain can be preserved in the analog hardware time domain by up to a deviation of \(y - x\), if the absolute \([x, y]\)-hypothesis is satisfied.

In case that the absolute \([x, y]\)-hypothesis is satisfied during software synthesis, we can predict properties of the realization in the analog hardware time domain based on the properties of the model in the virtual time domain. More specifically, the model and the realization satisfy exactly the same qualitative real-time properties, such as liveness and safety properties [28]. Furthermore, the model and the realization share the same quantitative real-time properties (up to a deviation of \(y - x\)), such as periodic and deadline properties. In Section 8.3.2, we show how the absolute \([x, y]\)-hypothesis can be incorporated into a synthesis tool.

**Example 8.2** Reconsider Example 8.1. It is easy to see that the letter(s) I, IE, IEE, IEEE and four blank spaces iteratively appear on the flash board in the model. Furthermore, we assume \(p\) and \(q\) to be two atomic propositions with the following meaning:

- **\(p\):** The ‘IEEE’ word appears.
- **\(q\):** The ‘IEEE’ word is erased.

It can be checked that quantitative real-time property \(\varphi_v\) holds in the model, which states that the ‘IEEE’ word appears at exactly 2.7 seconds after it is erased.

\[
\varphi_v = \Box(q \rightarrow \Diamond_{[2.7,2.7]}p).
\]

Suppose that the absolute \([x, y]\)-hypothesis is satisfied during the synthesis of the IEEE flash board controller software and \([x, y] = [0, 0.001]\). This indicates that the observations of actions in the hardware analog time domain are always later than their corresponding observations in the virtual time domain, but the delay never exceeds 0.001 seconds. In this case, we can ensure that the output letters on the flash board are always displayed in the correct order in the realization. The error observed in Example 2.4 will not occur in this case.

In Example 8.1, we showed that property \(\varphi_v\) is satisfied by the model in the virtual time domain. Now we can predict that the realization satisfies the following property \(\varphi_{ah}\) in the analog hardware time domain, which has a deviation of 0.001 from property \(\varphi_v\) (see Section 7.1.4).

\[
\varphi_{ah} = R_{\alpha}^{0.001\Box}(\varphi_v) = \Box(p \rightarrow \Diamond_{[2.699,2.701]}q).
\]

\(\varphi_{ah}\) states that the realization of the controller in the analog hardware time domain always displays the word ‘IEEE’ \(t\) seconds \((t \in [2.699, 2.701])\) after the word is erased.
8.2.2 Relative $[x, y]$-hypothesis

Since hardware clocks are never perfect in practice, a clock drift often exist between the (auxiliary) analog hardware clock and the reference clock. To predict the real-time properties of the realization in the reference time domain based on those in the analog hardware time domain, we propose the relative $[x, y]$-hypothesis, which requires that:

The ratio $R$ between the change rates of the (auxiliary) analog hardware clock $C_{ah}$ and the reference clock $C_{ar}$ is within interval $[x, y]$, where $x$ ($y$) is a lower bound (an upper bound) of $R$.

If the relative $[x, y]$-hypothesis is satisfied by the target platform, it is easy to see that the ratio between the change rates of reference clock $C_{ar}$ and analog hardware clock $C_{ah}$ should be within interval $[\frac{1}{y}, \frac{1}{x}]$.

The clock drift of $C_{ar}$ w.r.t. $C_{ah}$ can be also used to estimate the relative timing difference between timing behaviors interpreted in the reference time domain and those in the analog hardware time domain (see Example 4.5). In Section 7.4, we have shown real-time property-preservation results between timed systems (see Theorem 7.24). Therefore, we can predict real-time properties of the realization in the reference time domain from those in the analog hardware time domain.

Example 8.3 Reconsider the IEEE flash board controller in Example 8.2. Assume the change rate of the analog hardware clock can deviate from that of the reference clock by up to 0.5% (faster or slower)\(^2\). In this case, we can calculate that the ratio of the change rates between the reference clock and the hardware clock is within interval $[\frac{1}{1.005}, \frac{1}{0.995}]$. Consequently, we can predict that the realization of the IEEE flash board controller satisfies real-time property $\varphi_{re}$ in the reference time domain.

$$\varphi_{re} = R_r^{\frac{1}{1/1.005, 1/0.995}}(\varphi_{ah}) = \Box(q \rightarrow \Diamond[2.699/1.005, 2.701/0.995]p).$$

Since $[2.699/1.005, 2.701/0.995] \subset [2.685, 2.715]$, $\varphi_{re}$ is stronger than property $\Box(q \rightarrow \Diamond[2.685, 2.715]p)$. This indicates that the realization of the controller in the reference time domain always sends out the ‘IEEE’ word $t$ seconds ($t \in [2.685, 2.715]$) after the word is erased.

From the above example, we can see that the values of $x$ and $y$ in the absolute $[x, y]$-hypothesis and in the relative $[x, y]$-hypothesis have a direct impact on the preservation of quantitative real-time properties. This is illustrated by the following example, in which both hypotheses are involved.

Example 8.4 Assume model $M$ satisfies a deadline property $P_M$: $\Box(p \rightarrow \Diamond[2,3]q)$ indicating that stimulus $p$ is always followed by response $q$ within 2 to 3 seconds. Realization $R$ is synthesized from $M$ respecting both the absolute $[x_a, y_a]$-hypothesis and the relative $[x_r, y_r]$-hypothesis. Then we know that $R$ satisfies property:

\(^2\)In this case, the timing error can run up to around 7 mins per day.
The above example indicates that the quantitative difference between real-time properties of a model and its realization can be reduced by changing the values of either \( y_a - x_a, x_r \) or \( y_r \). The value of \( y_a - x_a \) is affected by the model itself, the scheduling algorithm and the computational capability of the target platform (see Section 8.3.2), while the values of \( x_r \) and \( y_r \) can be estimated by the drift factors between the digital hardware clock w.r.t. the reference clock.

The above reasoning can also be applied in a reverse way. For example, the drift of the analog hardware clock w.r.t. the reference clock is independent from the design process and can be pre-measured. Assume \( x_r \) and \( y_r \) are 0.99 and 1.02 respectively, and we require that property \( P_R (\square(p \rightarrow \diamond{[2-(y_a-x_a),3+(y_a-x_a)]q})) \) should be satisfied by the realization. In this case, the model should satisfy property \( P_M = \square(p \rightarrow \diamond{[(2-y_a+x_a)y_r,(3+y_a-x_a)/x_r]q}) \). The smaller the value of \( y_a - x_a \) is, the weaker property \( P_M \) will be. By estimating the value of \( y_a - x_a \), designers can know beforehand property \( P_M \) that the model should satisfy, in order to correctly deploy the system on the target platform.

8.3 Correctness-preserving software synthesis

In the previous section, we revealed ineliminable timing differences between a model and its corresponding realization, and we proposed two hypotheses to bound timing inconsistencies and to eliminate functional inconsistencies between a model and its realization. In this section, we show that both hypotheses (but especially the absolute \([x,y]\)-hypothesis) can be supported by synthesis tools for real-time software synthesis.

In the following, we first give a brief overview of the platform-independent modelling language POOSL (Parallel Object-Oriented Specification Language). Our discussion about POOSL focuses on its execution framework, which generates action traces for POOSL models. Following that, we show how the absolute \([x,y]\)-hypothesis is supported by the synthesis tool Rotalumis, which can convert a POOSL model into its realization. Finally, a toy example is given to show the advantages of the proposed synthesis approach.

8.3.1 POOSL

In this subsection, we briefly review the POOSL language, which is used to model real-time software system in this thesis. A more detailed explanation of the syntax and semantics of POOSL can be found in [96, 33, 97]. Here we give an outline of its execution framework, which helps to understand the correctness-preserving transformation from a POOSL model to its realization (Section 8.3.2).

\[ P_R = R_\delta^{[1,1/y_r,1/x_r]}(R_\delta^{(y_a-x_a)} \square (P_M)) \]
\[ = R_\delta^{[1,1/y_r,1/x_r]}(\square(p \rightarrow \diamond{[2-(y_a-x_a),3+(y_a-x_a)]q})) \]
\[ = \square(p \rightarrow \diamond{[2-(y_a-x_a)/y_r,(3+y_a-x_a)/x_r]q}) \]
Towards predictable real-time software synthesis

The POOSL language integrates a process part based on a timed and probabilistic extension of CCS and a data part based on the concepts of traditional object-oriented languages [100]. A POOSL model consists of a set of parallel processes, which perform their activities asynchronously and communicate with each other synchronously by message passing. The POOSL language is supported by the SHE-Sim tool, which has been successfully applied to model and analyze many industrial systems such as an internet router [93], a network processor [92], a microchip manufacturing device [50] and a multimedia application [98].

Based on a timed CCS-like semantics, any POOSL model can be represented by a timed labelled transition system. Each trace in the timed transition system represents a possible execution of the POOSL model. In [96, 33], the authors proposed an execution framework for POOSL models. In this framework, each process in the model is represented by a process execution tree (PET), and the evolution of the model is achieved by choosing and executing available actions from these PETs.

For example, Figure 8.5(a) shows the POOSL model of the two synchronized processes $P$ and $Q$ in Example 2.2. The PET of each process is given in Figure 8.5(b). Each leaf of a PET is a statement (such as sending message $a!\text{syn\_sig}$) or a (recursively defined) process method (such as process $\text{init()}$). During the evolution of the system, each PET provides its candidate actions (such as $a!\text{syn\_sig}$ in process $P$ and $a?\text{syn\_sig}$ in process $Q$ in Figure 8.5(b)) to the PET scheduler and dynamically modifies its tree according to the choice made by the PET scheduler. For example, after performing the synchronization (sending message $a!\text{syn\_sig}$ and receiving message $a?\text{syn\_sig}$), each PET modifies its tree from Figure 8.5(b) to Figure 8.5(c). The correctness of this
execution framework w.r.t. the semantics of the POOSL language is formally proven in [33].

8.3.2 Rotalumis

Software synthesis tool Rotalumis first takes as input a POOSL model built during system modelling and automatically generates the C byte code for the target platform. Then Rotalumis executes the byte code using a build-in interpreter. In this subsection, we give a line of reasoning to demonstrate the compliance of the Rotalumis tool with the absolute $[x, y]$-hypothesis. Specifically, the following techniques are adopted in Rotalumis.

1. **Process execution tree:**
   The POOSL language provides ample facilities to describe system characteristics such as parallelism, nondeterministic choice and communication that are not directly supported by implementation languages, such as C++ and Java. In order to provide a correct and smooth mapping from a POOSL model to a C++ realization, PETs are used to bridge the gap between the semantics of the two languages. The data part of a POOSL model is transformed into C byte codes interpreted by a byte code interpreter. The process part of a POOSL model is interpreted as a C++ tree structure, which has the same behavior as the PET implemented in SHESim. As a result, the generated realization exhibits exactly the same behavior as that in the model, if interpreted in the virtual time domain.

   Since the progress of the virtual time is monotonically increasing, which is consistent with the progress of the analog hardware time, the action order observed in the virtual time domain should be consistent with that in the analog hardware time domain. Therefore, the PET scheduler ensures that for any timed execution of the realization, a corresponding timed execution in the POOSL model can always be found, such that both executions exhibit the same observable action sequence. Therefore, the first requirement (see Section 8.2.1) of the absolute $[x, y]$-hypothesis can be guaranteed by the synthesis tool.

2. **Synchronization between the virtual time and the analog hardware time:**
   To minimize the timing inconsistencies between the realization interpreted in the analog hardware time domain and that interpreted in the virtual time domain, the PET scheduler tries to synchronize the virtual time and the digital hardware time within a finite deviation interval $[a, b]$ during execution. Since the timing deviation of the analog hardware clock w.r.t. the digital hardware clock is within interval $[0, \Delta]$, the execution of the realization in the analog hardware time domain is absolute $[a, b + \Delta]$-close to a trace of the model w.r.t. their timing deviation. Note that in practice $\Delta$ often takes very small values. For example, $\Delta$ is around $10^{-7}$ seconds, if the speed of the digital hardware clock is 10MHz. As a result, for most real-time systems, the timing deviations between the digital hardware time domain and its analog hardware time domain can be incorporated into the estimation of the timing deviation between the virtual time domain and the digital hardware time domain. We can see
Towards predictable real-time software synthesis

that the tool complies with the second requirement (see Section 8.2.1) of the absolute \([x, y]\)-hypothesis.

The timing deviations between the virtual time domain and the analog hardware time domain can result in real-time property inconsistencies between the two time domains. However, if the action orders are consistent in both time domains, we can at least ensure that qualitative real-time properties are preserved. Quantitative real-time properties, on the other hand, can be preserved up to a small deviation, which is related to the greatest lower bound \((d_{gl})\) and the least upper bound \((d_{lu})\) of timing deviations for all actions. In general, the smaller the value of \((d_{lu} - d_{gl})\) is, the “closer” are the quantitative real-time properties satisfied in the analog hardware time domain to those in the virtual time domain. We can minimize the “distance” between quantitative real-time properties by adopting a platform of better performance, by reducing the number of actions executed at a certain virtual time moment or by reducing the overhead caused by the scheduling.

8.3.3 A toy example

In Section 2.4.2, we showed that the TAU G2 tool can model a software clock precisely. That is, the time progress counted by the clock model precisely coincides with the virtual time progress. However, the time counted by the clock realization can deviate from the time counted by the hardware clock in an unbounded way. This is due to the fact that the TAU G2 tool lacks a mechanism to preserve the timing semantics during system synthesis. However small, the execution time of each action in the realization can be accumulated during the execution (see Figure 8.6\(^4\)), which results in unbounded timing deviations. Furthermore, due to the unpredictable execution times of actions, the execution times of actions in the realization of the clock cannot be easily compensated for by reducing the duration of the delay by a constant value.

Using the POOSL language, we can model the same software system precisely. In contrast to the TAU G2 tool, the time counted by the realization generated by Rotalumis does not deviate much from the hardware time. This is due to the fact that

\(^4\)In Figure 8.6, we assume that each delay is performed without any timing errors in the realization generated by TAU G2.
the execution of the realization follows the semantics of the model. Although each action still takes a certain amount of time in the realization, by synchronizing the virtual time and the digital hardware time, the execution times of these actions are not accumulated (see Figure 8.6), thereby avoiding a timing deviation that grows without bounds.

In Figure 8.7(a), we give a comparison of the timing deviations of two realizations automatically generated by TAU G2 and by Rotalumis respectively. Both realizations are executed on the same platform, which has a 150MHz CPU and a memory of 128MB, and runs the Windows 2000 operating system. Since the observation times of actions in both realizations are recorded after their actual observation, the actual timing deviations should be a little bit less than those shown in Figure 8.7. According to one long run of the realization generated by Rotalumis, we estimate that an upper bound of the timing deviation between corresponding actions in the model and in the realization is 0.01 seconds. Figure 8.7(a) shows that the timing deviation of the realization generated by Rotalumis does not accumulate during the execution. Notice that several peaks exist in Figure 8.7(b), which are caused by other system tasks running on the underlying operating system. Next to unbounded timing deviations, functional errors, such as those in the examples of Section 2.4.2, are also avoided in the realizations generated by Rotalumis.

### 8.4 Discussions

In the previous sections, we proposed two hypotheses and demonstrated their applicability to real-time system design. In this subsection, we give a brief survey of hypotheses used in real-time system design (especially in real-time system synthesis), and clarify their relations.

The perfect synchrony hypothesis lays the basis for the timing semantics of a family of synchronous modelling languages, such as Esterel, Lustre and Statecharts. These languages are widely used to design synchronous reactive systems. The hypothesis
Towards predictable real-time software synthesis

assumes that the underlying machine is infinitely fast, and hence that the system re-
actions are synchronized with the system inputs. The hypothesis is an important
ingredient to reconcile input-output determinism and concurrency for modelling
synchronous reactive systems [16]. The requirement made in the perfect synchrony
hypothesis is actually similar to that made in the timing semantics of platform-

independent modelling languages, where actions are instantaneous. Therefore, the
perfect synchrony hypothesis is perfectly “suitable” for system modelling.

Additional techniques are needed to guarantee that real-time properties of a syn-
chronous model (such as an Esterel model) can be preserved in the realization. For
example, an Esterel model can be transformed directly into a hardware realization,
where each module in the model is implemented by a digital circuit, and the concur-
rency and communication structure in the model remain unchanged in the hardware
realization. The perfect synchrony in an Esterel model is replaced by the digital cir-
cuit synchrony in its hardware realization, where the realization reacts to any input
within exactly one clock cycle, no matter how complex it is. Real-time property-

preservation from the model to its hardware realization relies on the fact that the
digital circuit synchrony does not depart from the perfect synchrony, which is en-
sured by substituting one cycle time reactions for zero-time reactions [15].

In [9], the authors use a timed automata with a real-time task extension to model
real-time systems. A transition of the timed automata can be associated with the
starting a real-time task, which has parameters such as a priority, a worst case exe-
cution time and a deadline. The schedulability of tasks can be checked in the model.
During system synthesis, the synchrony hypothesis is employed, which assumes that
the execution times of control activities are neglectable w.r.t. the execution times of
tasks. Based on this hypothesis, it is guaranteed that the schedulability of tasks in
the model is preserved into the realization. However, the preservation of timing be-
aviors of control (or scheduling) activities in the realization is not clearly addressed
in the paper.

In [44, 45], the authors proposed a platform-independent approach for real-time pro-
gramming. The approach has two distinguished phases: the programming phase
and the implementation generation phase. In the former phase, timing constraints
are annotated to the program and its components, which allows the verification of
the program to be carried out independently of any execution platform. In the lat-
ter phase, the schedulability of the program for the execution platform is analyzed
based on worse case execution times of program components. The correctness of the
program can be preserved into the implementation if a feasible scheduling scheme
can be constructed.

In [48], we have proposed an ε-hypothesis as a basis for property-preserving system
synthesis. The ε-hypothesis is based on the results proven in [49], which are the
predecessor of the absolute case in Part II of this thesis. Therefore, the ε-hypothesis
is closely related to the absolute [x, y]-hypothesis given in this chapter. The major
difference between the ε-hypothesis and the absolute [x, y]-hypothesis is that the for-
mer only uses one parameter to measure the (upper and lower) bounds of timing
deviations of actions. In other words, we consider the ε-hypothesis to be a special
case of the absolute [x, y]-hypothesis, where \( x = -\epsilon \) and \( y = \epsilon \).
8.5 Summary

To cope with the problem of inconsistent system synthesis in platform-independent design approaches, we first revealed the ineliminable timing differences between the virtual time domain and the reference time domain. After that, we built a sequence of (absolute and relative) timing differences to bridge the two time domains. Finally, we proposed two parameterized hypotheses: the absolute \([x, y]\)-hypothesis and the relative \([x, y]\)-hypothesis to bound the timing differences between the two time domains. The satisfaction of both hypotheses during software synthesis guarantees that real-time properties of a (platform independent) model can be preserved in its realization. The application of the absolute \([x, y]\)-hypothesis in real-time software synthesis is illustrated by the Rotalumis tool, which is capable of correctly and automatically transforming a POOSL model into its realization. The relative \([x, y]\)-hypothesis is used to estimate the influence of inaccurate hardware clocks on the quantitative real-time properties of a system. This is especially important for designing distributed real-time systems, where system components are distributed on different platforms with different hardware clocks. The real-time properties of the whole system are often reasoned about in a unified reference time domain, instead of in the individual hardware time domains of distributed components.

In the next chapter, a railroad crossing system is presented as a case study, which is designed by using the proposed synthesis approach in this chapter.
In the previous chapter, we proposed a predictable design approach for real-time software, which is characterized by correctness-preserving software synthesis. In this chapter, we apply the proposed approach to the design of a railroad crossing system. This chapter is organized as follows. Section 9.1 gives an introduction to the railroad crossing system. Section 9.2 specifies several desired properties of the system, which will be used in later design steps. Section 9.3 briefly demonstrates the modelling process of the railroad crossing system from a high-level model until a synthesizable model. Section 9.4 illustrates the transformation from a POOSL model into a realization while preserving the correctness of the system. Section 9.5 summarizes the chapter.

9.1 A railroad crossing system

In this section, we introduce the functionality of the crossing system and propose a parallel solution to the system design.

9.1.1 System description

We choose a railroad crossing system as an example, which is similar to the standard railroad crossing problem used to compare different formal frameworks for modelling real-time systems [38]. As shown in Figure 9.1, four stations are connected by two orthogonal tracks. Train a (b) runs back and forth between stations 1 and 2 (and stations 3 and 4). Four sensors (A, B, C and D) are installed at some distance to the crossing to detect the passing of the trains. In reality, such a system can be a basic element in a larger railway network. The correct operation of the railroad crossing system requires that:
In this chapter, we exemplify the design of the controller for the railroad crossing system, which controls the motion of both trains and the assignment of the crossing. In our case, the physical system is built by LEGO materials. Some relevant data in our example are given in Table 9.1.

### 9.1.2 A parallel solution

The railroad crossing system can be abstracted as shown in Figure 9.2, where a controller interacts with physical objects (such as the trains and the sensors) through a communication interface. Due to the concurrent nature of the physical system, we
adopt a parallel solution to model the system. Each physical object (train or crossing) is controlled by a separate and relatively simple process as shown in Figure 9.3. The behavior of the whole system can be studied by investigating the interactions between these processes.

To distinguish physical objects from their control objects, we adopt the player concept in [97]. We denote physical objects as actors, control objects as images and their combinations together with interface objects as players. For example, physical trains are called train actors, the control objects of the physical trains are called train images and their combinations with interfaces are called train players (as shown in Figure 9.3). Note that physical crossing objects have no active physical behaviors, therefore the behavior of the crossing player is identical to that of the crossing image. We do not distinguish them explicitly in the sequel and simply denote both the crossing image and the crossing player as the crossing when there is no ambiguity.

The railroad crossing system has two distinct layers: the control layer and the physical layer. The control layer consists of a crossing image and two train images, which performs abstract reasoning and generates action plans for physical objects. The
physical layer consists of two train actors with sensors, which observe the physical world and execute physical actions. In this example, we consider the control layer to be a mirrored image of the physical layer, because both layers refer to the same physical objects in the environment.

To ensure correct and efficient operations of the whole system, both the control layer and the physical layer have to be synchronized in time to maintain the correspondence between the two layers. However, due to the existence of delays during the synchronization, such as the sampling delay, the transmission delay and the actuation delay, the control layer and the physical layer may not always be consistent with each other. For example, as shown in Figure 9.4, both the train actor and the train image are initialized at the same position. Due to the delay caused by the sensor sampling and the signal transmission, the train image is unaware of the train actor passing the left sensor until after some delay. By that time, the train actor is already at a position somewhat to the right of the left sensor.

To compensate for the above inconsistencies between the control layer and the physical layer, and to prevent the physical train from stopping inside the crossing area, we define a virtual crossing area in the controller, which is denoted as the critical zone in Figure 9.1. The critical zone is larger than the actual crossing area in the physical world. When a train image approaches a border of the critical zone from outside of the zone, the train image has to request for permission to enter the crossing (image). Notice that the corresponding train actor may already pass the border of the critical zone and approach the border of the crossing area. If the request is denied by the crossing, the train image has to stop the train actor immediately and let it wait until the permission is granted. The determination of the optimal size of the critical zone will be discussed in the next section.

One important benefit offered by our design solution is its flexibility w.r.t. the system topology. When the topology of the physical system changes (e.g. when the system is extended with more crossings and tracks), the behavior of each crossing process remains unchanged, and each train image can adapt easily to the new system ac-
according to the number of crossings on its running track.

9.2 Requirement analysis

The requirements of the system have been briefly introduced in the previous section. Now, we are going to specify these requirements more clearly so that they can be easily traced during the design process. The following system properties need to be considered during the system design:

- **P₀**: The system should be deadlock free.
- **P₁**: The crossing should ensure that the permission to enter the crossing is assigned to at most one train image at a time.
- **P₂**: The train image should ensure that the train actor can enter the crossing area only after it obtains the permission from the crossing.
- **P₃**: The train image should not release the crossing until its corresponding train actor has left the physical crossing.
- **P₄**: The system should operate as efficiently as possible.

The satisfaction of property **P₀** is determined by interactions between parallel processes in the system, which can be verified at a high level of abstraction and preserved during the refinements.

Property **P₁**, **P₂** and **P₃** together guarantee that the train actors never collide. The satisfaction of property **P₁** is determined by the crossing process itself. But the satisfaction of property **P₂**, **P₃** and **P₄** is closely related to two parameters, which are illustrated below.

- The size of the critical zone: A critical zone that is either too large or too small will lead to problems. A critical zone that is too large can cause unnecessary waiting of the train actor. On the other hand, a critical zone that is too small may not be able to prevent the train actor from stopping inside the crossing. The optimal size of the critical zone can be estimated based on the speed of the train actor and the synchronization delays between the train image and the train actor.

- The releasing time of the crossing: Similarly, to ensure the maximal operation efficiency, the releasing time should not be too late or too early. Releasing the crossing too late might lead to the unnecessary waiting of the other train, while releasing the crossing too early may cause collisions. In principle, the control process of the train should release the crossing when the train just leaves the crossing border. The optimal releasing time of the crossing is estimated based on the length of the train actor, the position of the physical crossing border and the speed of the train actor.
Suppose the speed of train \( a \) (\( v_a \)) and its deceleration distance (\( D_a \)) are as specified in Table 9.1. An upper bound of the transmission delay (\( t_{trains} \)) between the controller and the sensors or the motors is 0.02 seconds and the sampling rate of the controller (\( t_{sampling} \)) is 0.02 seconds. Then we can calculate the greatest lower bound of the distance \( D_{min} \) between the border of the critical zone and the border of the crossing area using the following formula:

\[
D_{min} = v_a \cdot (2 \cdot t_{trains} + t_{sampling}) + D_a = 0.0732 m.
\]

Now property \( P_2, P_3 \) and \( P_4 \) can be further quantitatively specified by considering the efficiency of the system. We assume that the speed of each train actor is constant between two sensors, if no stop occurs. Therefore, when the train image receives the first sensor signal on each journey, it can estimate time \( t_2 \) at which its train actor arrives at the border of the critical zone. If its request to access the crossing is denied, the train image has to stop its train actor before \( t_2 \). On the other hand, the train image should not stop the train actor earlier than \( t_1 \) (\( t_1 \leq t_2 \))\(^1\). An earlier stop may decrease the efficiency of the system. Therefore, we require that the following real-time property should be satisfied by the system.

- \( P_5 \): On one journey, after the train image receives the first sensor signal, either it passes the crossing without stopping, or its request to access the crossing is refused and it has to send out a message to stop the train actor within time interval \([t_1, t_2]\).

In our example, \( t_1 \) and \( t_2 \) are 1.000 and 1.025 seconds respectively for train \( a \), and train \( a \) takes at most 1.7 seconds to pass the crossing from the position of the first sensor, if no stop occurs. More specifically, property \( P_5 \) for train \( a \) can be formalized using the following MTL formula.

- \( P_{5a} \).\( \Box(p_a \rightarrow ((\Diamond [1.000,1.025] r_a \land \Diamond [1.000,1.025] q_a) \lor \Box [0.1.7] \neg r_a)) \), where \( \Box \) and \( \Diamond \) denote “always” and “eventually” respectively and \( p_a, q_a \) and \( r_a \) represent the following atomic propositions.
  - \( p_a \): The train image of train \( a \) receives the first sensor signal on one journey.
  - \( q_a \): The train image of train \( a \) sends out a message to stop its train actor.
  - \( r_a \): The train image of train \( a \) receives a message that denies its request to enter the crossing.

The above formula states that after the train image receives the first sensor signal, either both \( r_a \) and \( q_a \) are observed within interval \([1.000,1.025]\) or the train actor passes the crossing without stopping.

We can see that the correctness of the controller depends on both the qualitative and quantitative real-time constraints. For example, to ensure qualitative correctness, we need to avoid granting access to the crossing to both train players at the same time.

---

\(^1\)Ideally, the system achieves maximum efficiency, when \( t_1 = t_2 \). However, this can hardly be realized in practice.
For quantitative correctness, we need to ensure that the train images perform control commands at the right time, such as the time to request the crossing, the time to stop train actors if permission is denied, and the time to release the crossing.

9.3 System modelling

As we have discussed in Section 2.3, system modelling usually involves a series of step-wise refinements. At each refinement step, the model is replaced by a more detailed one. The refinement of the model usually follows certain rules to ensure its correctness. A typical example is the refinement in transformational design approaches for parallel system modelling [54, 97], where a model is refined by a sequence of predefined rules. These rules ensure that both models are observational equivalent during the refinement, (i.e. both models exhibit the same behavior to external observers). This observational equivalence relation implies that certain properties can be preserved during the refinement. Another example is the rules proposed in [76], which ensure that certain real-time properties are preserved between discrete real-time models.

Since property-preservation between a set of refined models is not of our major concern, we leave out the investigation of details of the refinement rules for property preserving during model refinement. The major task in this section is to generate and present a detailed model for the subsequent system synthesis, where we put our focus on property-preservation from the model to its realization.

We use the top-down design paradigm to model the crossing system. It starts from building an initial model of the system, followed by successive refinements. An untimed refinement and a timed refinement are performed to obtain an adequate design model for the succeeding system synthesis. The consistency between different levels of models is ensured by observational equivalence.

9.3.1 Initial model

We first tackle the problem of obtaining a correct crossing assignment scheme. The related requirements on the crossing assignment state that the access to the same
crossing should be granted to only one train at a time \( (P_1) \) and that the system should be deadlock free \( (P_0) \). The state-action diagrams of the players involved \( (\text{Crossing, TrainA and TrainB}) \) are shown in Figure 9.5. Figure 9.5(a) gives the state-action diagram of a possible design of the crossing \( (\text{image}) \), where \( \text{tr}_a \text{?request} \) indicates that the crossing receives a \text{request} message from the \text{TrainA} player and \( \text{tr}_b \text{!granted} \) indicates that the crossing sends a \text{granted} message to \text{TrainB} player. The behavior of the train player \( (\text{including the train image, the train actor and the communication interface}) \) can be modelled at a high-level of abstraction. As shown in Figure 9.5(b), the behavior of the train player is abstracted into 4 states, where \( \text{cr!request} \) indicates that the train player sends a \text{request} message to the crossing.

In Figure 9.6, a tool environment for building the model is demonstrated. The main window \( (\text{SHESim System Level Editor}) \) in Figure 9.6 shows the working space to construct the model. Here the initial model consists of three processes, which are used to model the crossing \( (\text{Crossing}) \) and two train players \( (\text{TrainA and TrainB}) \). In this tool environment, the behavior of each player can be succinctly modelled in the POOSL language. For example, the left part of Figure 9.6 gives a piece of POOSL code, which models the behavior of train players. We can see that each state transition in Figure 9.5(b) can be directly mapped to a single POOSL statement. Furthermore, interactions between processes are accomplished through ports connected by static channels, such as ports \( \text{cr} \) and \( \text{tr}_a \) connected by channel \( A \) in Figure 9.6. The correctness of such a design solution can be verified by simulation of the initial model. The interaction diagram window at the right part of Figure 9.6 shows the interactions between the processes during simulation, which can assist designers to analyze possible design errors.

Furthermore, since the semantics of the modelling language \( \text{(POOSL)} \) is based on formal theory CCS [66] with a probabilistic real-time extension, it is also possible to verify properties of the model by using exhaustive verification techniques. We verified property \( P_0 \) and \( P_1 \) of the initial model in CWB-NC [25] and Spin [42].

\(^2\text{TrainA and TrainB exhibit the same untimed behavior.}\)
9.3.2 Untimed refinement

At the first stage of the model refinement, the behavior of the train players (TrainA and TrainB) is refined by using three parallel processes: TrainImage, TrainActor and Interface (as shown in Figure 9.7). The TrainImage models the behavior of the control process of the physical train. The TrainActor models the behavior of the physical train itself, and the Interface establishes the communication between the TrainImage and the TrainActor. The interactions between the three parallel processes are also specified in the model. The code shown in Figure 9.7 describes a part of the behavior of the TrainImage, starting from the moment when it receives a message from the first sensor until the moment when it releases the crossing. The code in bold-face refers to refined actions w.r.t. the abstract behavior of the train in the initial model. These actions are used to control the TrainActor through the Interface.

Figure 9.8 shows the state-transition diagram of each process in the train player. The behavior of the train image is given in Figure 9.8(a), where i!start indicates that the train image sends a start message to the interface. In Figure 9.8(b), i?pause indicates that a pause message is received from the train image and a!resume indicates that a resume message is sent to the train actor.

The refinement of the train player carried out in this step is performed in such a way that the observable communication behavior of the train players remains unchanged w.r.t. to the behavior in the initial model, which is in accordance with the notion of observation equivalence in CCS. By applying the reduction rules defined in standard CCS, we have formally proven that the behavior of our refined model in Figure 9.8 is indeed observationally equivalent to that of the model in Figure 9.5(b). Therefore, the refined train player (TrainActor || Interface || TrainImage) is a proper refinement of the train player in the initial model.

3Note that the timing information of the system is not yet considered in this model.
A case study – a railroad crossing system

(a) The state-action diagram of the train image

(b) The state-action diagram of the interface

(c) The state-action diagram of the train actor

Figure 9.8: The state diagrams of the train player
1: Check leftsensor(); // polling the sensor state till receiving a sensor signal
2: delay (T1); // the duration from the train passing the 1st sensor to reaching the border of the critical zone
3: cr!request;
4: sel
5: cr?granted;
6: i!continue;
7: delay (T2); // the duration from the train entering the critical zone to leaving the crossing without stopping
8: or
9: cr?denied;
10: i!pause;
11: cr?granted;
12: i!resume;
13: delay (T3); // the duration from the train restarting in the critical zone to leaving the crossing area
14: les;
15: cr!release;

Figure 9.9: The timed refinement of the crossing system

9.3.3 Timed refinement

To find a design solution satisfying the desired quantitative timing properties, we need first to incorporate the timing information into the untimed model. For example, the behavior description in Figure 9.7 can be extended with timing information as illustrated in Figure 9.9. T1, T2 and T3 are used to specify the timing relations between actions, which are estimated according to relevant information (such as the speed of the train and the size of the critical zone). In our case, T1 is 1.015 seconds and T2 is 0.7 seconds for the train image of train a.

It can be checked (see further) that the following property P_{5a}^v is satisfied by the model in the virtual time domain.

\[ P_{5a}^v : \Box(p_a \rightarrow (\Diamond[1.015,1.015]r_a \land \Diamond[1.015,1.015]q_a) \lor \Box[0,1.715] \neg r_a)) , \]

where p_a, q_a and r_a are defined as in Section 9.2. P_{5a}^v states that after the train image of train a receives the first sensor signal on one journey, either it receives a denied message from the crossing at 1.015 seconds and sends out a pause at the same time, or it moves on without receiving the denied message during time interval [0, 1.715].

Here we show that property P_{5a}^v is indeed satisfied by the model. Let us look at a piece of code for train images in Figure 9.9. After the train image of train a receives the first sensor signal on one journey, it waits for 1.015 (T1) seconds. Then at 1.015 seconds, the train image sends a request message to the crossing. Note that all interactions are instantaneous in the virtual time domain. The crossing replies to the train image without any time delay. In the case that the request to access the crossing is denied, the train image gets the denied message and sends out the pause message at exactly 1.015 seconds. In the case that the request to access to the crossing is granted, the train image just moves on for another 0.7 (T2) seconds to pass the crossing.
9.3.4 Synthesis model

In the previous steps of the design process, design decisions are made through successive refinements and a detailed design model can be obtained during this process. Now we prepare the final blueprint for system synthesis. In this step, the environmental part of the model (train actors) is removed, and the interface part is replaced by a single interface process (\texttt{LEGO\_Dacta\_Interface}), which acts as an intermediate between the controller and the physical world. The controller part remains unchanged.

The replacement of the interface processes by the \texttt{LEGO\_Dacta\_Interface} process (see Figure 9.10) does not influence the interactions between the controller part (train images) and the interface part. A POOSL data class \texttt{LEGO\_Dacta} plays the role to communicate with the physical world. The interactions between the interface processes and the train actors are replaced by the data methods of the \texttt{LEGO\_Dacta} class. For example, in Figure 9.10, the original interactions \texttt{a!pause} and \texttt{a?pause} between the interface process and the train actor of train \texttt{a} are replaced by a data method \texttt{dacta TurnOff(1)}, where \texttt{dacta} is an instance of the \texttt{LEGO\_Dacta} class and parameter “1” represents train \texttt{a}. The data methods in the \texttt{LEGO\_Dacta} class only define a high-level communication interface with the physical world in the synthesis model and have no direct communication capability with the physical world. During system synthesis, the \texttt{LEGO\_Dacta} class will be replaced by its counterpart in the C++ realization, which provides the actual physical communication with the physical world.

9.4 System synthesis

In this phase, the synthesis model devised during the design process is automatically transferred into a realization. We focus on the preservation of quantitative real-time properties between the model and the realization, which is exemplified by

\footnotesize

\textsuperscript{4}The duplicate port names in different interface processes have to be renamed in \texttt{LEGO\_Dacta\_Interface}. These syntactical replacements do not change the interactions between the interface processes and the image processes.
9.4.1 Implementation of communication interfaces

In Section 8.3.2, we have mentioned that Rotalumis can transform a POOSL model into its realization by adopting different strategies to map the process part and the data part of a POOSL model. For the data part, the standard POOSL data classes (such as Integer, String, Real, Object and their subclasses) can be directly transformed into interpretable C byte code. The data classes for the communication with the physical world are not provided by the standard Rotalumis tool. But the tool provides the mechanism to easily incorporate necessary communication classes into it. For the sake of completeness, here we give a brief illustration of the implementation of communication data classes in Rotalumis.

First, consider the infrastructure of the whole system. In our example, a separate physical Dacta Controller provides the low level device drivers to control the LEGO motors and sensors. The designed control processes interact with the physical trains and sensors through the Dacta Controller (as shown in Figure 9.11). The controller runs on a desktop computer and communicates with the Dacta Controller through a serial port.

Interface data class LEGO_Dacta defined in the synthesis model only provides a communication interface for physical communication. The actual implementation to communicate with the Dacta controller is incorporated into the Rotalumis tool. For example, the interface for stopping train \( a \) (dacta TurnOff(1)) can be implemented by a segment of C++ code as shown in Figure 9.12.
9.4.2 Parameter estimation in the absolute \([x, y]\)-hypothesis

In Section 8.3.2, we have demonstrated that the Rotalumis tool complies with the absolute \([x, y]\)-hypothesis, which guarantees the same qualitative and quantitative (up to a small deviation) real-time properties. Parameters \(x\) and \(y\) are lower and upper bounds of the timing deviation between the observation times of corresponding actions in the model and in the realization. In general, the larger the difference between \(y\) and \(x\) is, the larger the deviation is between the quantitative real-time properties satisfied by the model and the realization. In practice, we can obtain the values of \(x\) and \(y\) in various ways. Here we give two examples.

- During the execution of the realization, the scheduler can record the timing deviation of each action at run time. Then the values of \(x\) and \(y\) can be estimated according to the recorded timing deviations. Since this approach is based on simulation, it offers the benefit of easy applicability. Furthermore, no estimation is required of the execution times of actions and of the scheduling cost in this case. The major pitfall of this approach is that the analysis results based on simulation techniques may not be reliable, since each simulation run only exploits a part of a single trace.

- An alternative approach is to model the platform, the time duration of each action and the timing behavior of the scheduler. The values of \(x\) and \(y\) can be estimated in the integrated model of the platform and the system. The major challenge of this approach is to adequately estimate the execution time of each action and the timing cost of the scheduler. This estimation is especially difficult for complex platforms adopting techniques such as caching, pipelining and memory management, which make it difficult to predict execution times of actions and the scheduling cost. But this approach can be an effective solution for simple embedded platforms. This approach is elaborated upon in [31].

In the scheduler of Rotalumis, each action is invoked when the digital hardware time has exceeded its virtual time. In other words, the digital (or analog) hardware time of each action is no earlier than its virtual time. Therefore, we can safely estimate the lower bound \(x\) in the absolute \([x, y]\)-hypothesis to be 0. In our example, we estimated \(y\) using the first estimation approach, since it is difficult to model the

<table>
<thead>
<tr>
<th>Timing deviation ((10^{-5} \text{ seconds}))</th>
<th>1 (\sim) 5</th>
<th>5 (\sim) 10</th>
<th>10 (\sim) 20</th>
<th>20 (\sim) 30</th>
<th>30 (\sim) 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1,877,257</td>
<td>112,813</td>
<td>9,909</td>
<td>26</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 9.2: Statistics of the timing deviations
underlying platform \(^5\) adequately. We recorded around 2,000,000 timing deviations of actions between the virtual time and the digital hardware time. To reduce the timing overhead caused by recording activities, only the timing deviation of the last action observed at each virtual time moment is recorded (see \(d_f\) in Figure 9.13). As shown in Table 9.2, most timing deviations fall between \(1 \times 10^{-5}\) and \(5 \times 10^{-5}\) seconds, and only a few timing deviations reach up to \(5 \times 10^{-4}\) seconds \(^6\). These larger deviations are mainly caused by the background activities of the operating system. Since the observation time of an action in the analog hardware time is at most 1 stepwidth (\(\frac{1}{2} \times 10^{-6}\) seconds in this example) later than its observation time in the digital hardware time domain, we can estimate the value of \(y\) in this example to be 0.001 seconds.

Based on the estimated values of \(x\) and \(y\), we can predict the quantitative timing properties of the realization in the analog hardware time domain. For example, we already know that property \(P_{5a}^h = \Box (pa \rightarrow ((\Diamond [1.014,1.016]^r_a \land \Diamond [1.015,1.015]^q_a) \lor \Box [0,1.714]^\neg r_a))\) is satisfied by the model in the virtual time domain, so we can predict that property \(P_{5a}^h\) will hold in the realization interpreted in the analog hardware time domain.

\[
P_{5a}^h : R_{5a}^{\text{0.001}}(P_{5a}^h) = \Box (pa \rightarrow ((\Diamond [1.014,1.016]^r_a \land \Diamond [1.014,1.016]^q_a) \lor \Box [0,1.714]^\neg r_a)).
\]

### 9.4.3 Parameter estimation in the relative \([x, y]\)-hypothesis

The inaccuracy of the platform clock can also affect the quantitative real-time properties of a realization. This influence is addressed by the relative \([x, y]\)-hypothesis. Parameters \(x\) and \(y\) in the hypothesis are determined by the hardware clock of the platform, and are independent from any design solutions.

In our example, we choose the time of the environment as the reference time, which is counted by a perfect physical clock. If we assume that the average change rate of the hardware clock in the deployed platform can deviate from the perfect physical clock by at most 0.5% during each step of the hardware clock, then \(x\) and \(y\) in the hypothesis are 0.995 and 1.005 respectively. Now we can predict the real-time property \(P_{5a}^r\) satisfied by the realization in the reference time domain.

\[
P_{5a}^r : R_{5a}^{\text{0.005}}(P_{5a}^h) = \Box (pa \rightarrow ((\Diamond [1.014,1.016]^r_a \land \Diamond [1.014,1.016]^q_a) \lor \Box [0,1.714]^\neg r_a)),
\]

which is stronger than property \(P_{5a}^r : \Box (pa \rightarrow ((\Diamond [1.008,1.022]^r_a \land \Diamond [1.008,1.022]^q_a) \lor \Box [0,1.705]^\neg r_a))\) (see Section 5.2.2). Property \(P_{5a}^h\) ensures that after the image process (Train\textunderscore Image) of train \(a\) receives the first signal, either it receives a denied message and sends out a pause message within interval \([1.008,1.022]\), or it continues to move on during time interval \([1.008,1.022]\). \(P_{5a}^r\) is stronger than the required property \(P_{5a}^h\) in Section 9.2. Consequently, we can guarantee that train \(a\) never stops inside the crossing area, while still maintaining the operating efficiency of the system.

\(^5\)The underlying platform in our case is a PC with a PIII 700MHz processor, 128MB memory and the Windows 98 OS.

\(^6\)Note that the timing deviations are observed after the issuing of the actions. Hence, the actual timing deviations should be smaller than this value.
For some platforms, based on the estimated values of $x$ and $y$ for the absolute $[x, y]$ and relative $[x, y]$ hypotheses, it cannot be ensured that train $a$ always stops outside the crossing when a stop is necessary. In such cases, designers can refine their model to satisfy a stricter timing property, adopt a platform of better performance, or use more accurate hardware clocks.

9.5 Summary

In this chapter, we demonstrated the application of both the absolute $[x, y]$-hypothesis and the relative $[x, y]$-hypothesis in real-time software design. This is exemplified by designing a rail-road crossing system, the correctness of which relies on the quantitative real-time properties of the control processes. We illustrated a correctness-preserving software synthesis by preserving an important quantitative property from the model to the realization in the reference time domain. By estimating parameters in the absolute $[x, y]$-hypothesis and the relative $[x, y]$-hypothesis, we can predict the quantitative real-time properties of the realization based on those of the model. This approach can be applied also in a reverse way. For example, given a required property for the realization, by estimating the drift of the analog hardware clock w.r.t. the reference clock, one can determine the property that should be satisfied by the realization in the analog hardware time domain. Further by estimating the bounds of timing deviations caused by the execution of actions and the scheduler, one can derive the quantitative real-time properties that should be satisfied by the model in the virtual time domain in order to correctly deploy the system on the target platform.
Chapter 10

Conclusion

The continuously increasing complexity of real-time systems demands a multi-stage design process, where the design decisions are made through a series of design stages. During each design stage, design decisions are made based on a part of the system (either an abstraction or a component) only. Therefore, it is essential for a design approach to have sufficient support for predictability, which guarantees that earlier design decisions based on partial information of the system are still valid during later design stages, where more complete information is incorporated. In this thesis, we focused on the predictability issues during real-time software design. In this chapter, this work is summarized and suggestions for future research are presented.

10.1 Contributions

The contributions of this work are summarized in the following three areas.

The analysis of predictability support in real-time system design:

In Part I of this thesis, we first pointed out that in a multi-stage design process, predictability of a design approach is often exhibited as two closely related concepts: correctness preservation and compositionality. The former emphasizes the consistency of different abstractions of the system during system transformation, and the latter emphasizes the consistency of components during system integration. Furthermore, the combination of correctness preservation and compositionality can largely reduce design difficulty and improve design efficiency. We also introduced several basic requirements for the semantics of design languages (requirement, modelling and implementation languages) to obtain a predictable design.

Based on these requirements, we investigated the predictability support of existing real-time design approaches, which were classified into platform-dependent approaches and platform-independent approaches based on different timing concepts adopted. We identified problems causing unpredictability during real-time software
design for both approaches. In platform-dependent design approaches (adopting the hardware time concept), unpredictability is often observed during model analysis, model integration and model transformation. This is mainly caused by the competition for the shared physical time resource. On the other hand, platform-independent design approaches (adopting the virtual time concept) provide better support for predictability during model analysis, model integration and model transformation. However, due to the large gap between the timing semantics of the modelling and implementation languages, unpredictability is often observed during system synthesis in these approaches.

**Real-time property-preservation between timed systems:**

Due to the gap between the timing semantics of modelling languages and implementation languages, a platform-independent model can only be an approximation of its realization. Consequently, real-time properties verified in the model may not hold in the realization. We addressed this problem in a formal framework, where both the model and the realization are considered as timed systems.

First, we proposed two proximity measures between timed systems to specify their absolute timing difference and relative timing difference. We also showed that these proximity measures are essentially different according to the notion of equivalent metrics.

Second, we used MTL formulas to express real-time properties of a system. We proved that the weakening relation between MTL formulas can be derived from their sub-formulas and time-bounds. Several special weakening functions were proposed to establish real-time property relations between timed systems.

Third, we proposed tube functions to facilitate the proof of property-preservation between timed systems, which actually give another characterization of the proposed proximity measures.

Finally, we proved that real-time property can be preserved between approximate timed systems. The preservation results show that real-time properties of a timed system can be preserved into its approximate timed systems up to a small deviation. The extent of this deviation can also be predicted quantitatively.

**Correctness-preserving real-time software synthesis:**

In this part of the work, we proposed a correctness-preserving synthesis approach for real-time software, which is based on the real-time property-preservation results we have proven in the previous part.

First, we revealed the ineliminable timing differences between the model in the virtual domain and the realization in the reference time domain, and bridged the timing differences between the two time domains by a sequence of (absolute and relative) timing differences between several intermediate time domains.

Second, two parameterized hypotheses were proposed to bound the absolute timing differences and the relative timing differences between the various time domains.
The satisfaction of the two hypotheses ensures that qualitative real-time properties can be preserved between different time domains. Furthermore, the parameters of both hypotheses can be used to predict the preservation of quantitative real-time properties. We proposed a synthesis approach for real-time software, which shows that the two hypotheses can be compiled during the software synthesis from the model to the realization.

Finally, we used the proposed synthesis approach to design a railroad crossing system. The parameter estimation for both hypotheses was also illustrated.

10.2 Options for future research

We identify several interesting topics that are worth further investigation.

- In principle, the two-phase execution framework adopted by platform independent approaches provides better predictability support for real-time system modelling. However, most existing platform-independent approaches still lack a formal proof for real-time property-preservation between models at different abstraction levels. Systematic correctness-preserving real-time system modelling needs to be further investigated.

- In Chapter 8, we demonstrated that the absolute $[x, y]$-hypothesis can be incorporated into the Rotalumis synthesis tool. The parameters $x$ and $y$ capture the bounds of the deviation between the observation times of each action in the virtual time domain and in the analog hardware time domain. A smaller interval $[x, y]$ implies that the model and the realization have quantitative real-time properties that lie closer. The size of interval $[x, y]$ can be affected by the scheduling cost. How to optimize schedulers for specific applications to reduce the scheduling cost needs to be further investigated.

- The execution times of actions can also affect the size of interval $[x, y]$ in the absolute $[x, y]$-hypothesis. For computationally intensive applications, the execution of some actions may take a significant amount of hardware time, which leads to a large interval $[x, y]$.

We can relieve the above problem by introducing a proper abstraction mechanism, which hides computationally intensive actions from the reasoning of desired properties. For example, in process algebra (e.g. CCS and CSP), an abstraction mechanism is used to hide actions that are not observable or that are not interesting, and certain properties of the system can be verified based only on observable actions. In other words, the satisfaction of some real-time properties can be determined only by the sequence of observable actions and their observation times. Consequently, the preservation of these properties is determined only by bounds on the deviation between the observation times of observable actions in virtual time domain and in the analog hardware time domain. Many research topics are still open in this research area. Issues involve, for instance, how to minimize the timing deviation of observable actions between the virtual time domain and the analog time domain (which may be af-
fected by the execution times of unobservable actions) and how to incorporate a proper abstraction mechanism into a synthesis tool.

• In Chapter 8, we demonstrated that the Rotalumis tool can automatically generate a realization for a single-processor platform complying with the absolute \([x, y]\)-hypothesis. However, when a multi-processor architecture is used to deploy the realization, a central scheduler may not exist and the proposed techniques may be inefficient or unable to ensure the compliance of the absolute \([x, y]\)-hypothesis for distributed systems. It is a challenging research problem to automatically and correctly synthesize distributed systems.
Bibliography


[89] Telecommunication standardization sector of ITU. Recommendation z100; specification and description language (SDL), Nov 1999.


## Index

| A | abstraction | 17, 18 |
| A | arc | 43 |
| C | clock | 115 |
| C | ~ deviation | 115 |
| C | ~ drift | 115 |
| C | ~ step-width | 115 |
| C | composability | 23 |
| C | compositionality | 20 |
| C | correctness preservation | 14 |
| C | correctness-preserving | 23 |
| C | ~ abstraction | 23 |
| C | ~ refinement | 23 |
| D | displacement | 57 |
| D | $D_a^-$ | 61 |
| D | drift factor | 119 |
| F | function | 79 |
| F | $R_a ^<-$ | 78 |
| F | $R_a ^= -$ | 78 |
| F | absolute $\epsilon$-weakening $\sim$ | 79 |
| F | absolute $\epsilon\Box$-weakening $\sim$ | 79 |
| F | $R_t^{[x,y]}$ | 80 |
| F | relative $[x,y]$-stretching $\sim$ | 80 |
| F | tube $\sim$ | 83 |
| F | absolute $[x,y]$-$\sim$ | 84, 87 |
| F | relative $[x,y]$-$\sim$ | 90, 91 |
| G | graph | 43 |
| G | ~ annotation | 43 |
| G | ~ structure | 43 |
| H | hypothesis |  |
| I | interleaving semantics | 54 |
| L | language | 15 |
| L | design $\sim$ | 15 |
| L | implementation $\sim$ | 18 |
| L | modelling $\sim$ | 16 |
| L | requirement $\sim$ | 16 |
| M | model | 9 |
| M | ~ analysis | 9 |
| M | ~ integration | 9 |
| M | ~ transformation | 9 |
| M | MTL logic | 69 |
| N | node | 43 |
| N | normal form | 53, 55 |
| N | normalization | 53 |
| O | operator | 75 |
| O | $\ominus$ | 75 |
| O | $\ominus$ | 77 |
| O | $\otimes$ | 77 |
| P | piecewise | 8, 14 |
| P | platform-dependent approach | 24 |
platform-independent approach 29
predictability 1, 9, 13
proposition set 52

R
real-time property 43, 69
refinement 17, 19

S
absolute $\varepsilon$-spanning 96
state sequence 52
stepwise 8, 14
synthesis
  software ~ 10
  system ~ 9, 18
system modelling 16

temporal logic 47
  branching ~ 48
  linear ~ 47
time
  hardware ~
    digital ~ 111, 117
    physical ~ 4, 114
    true ~ 114
    virtual 9
    virtual ~ 30
time domain
  hardware ~
    digital ~ 117
    true physical ~ 114
    virtual ~ 30
time interval
  concurrent ~ 54
  sequential ~ 52
time interval sequence
  concurrent ~ 54
  sequential ~ 52
time-bound 69
timed state 83
timed state sequence
  concurrent ~ 54
  sequential ~ 52
timed system
  concurrent ~ 54
  sequential ~ 53
timeliness 4, 34
timing behavior 5, 51
timing deviation 111
timing difference
  absolute ~ 57
  relative ~ 61
timing drift 111
transformation
  design 18
  model 17
two-phase execution framework 29, 54
weakening relation 72
Jinfeng Huang was born on the 18th of January 1976 in Yangquan, Shanxi province, P.R.China.

He received his B.Sc. in computer science from China University of Mining and Technology in 1997, and his M.Sc. in computer science from Xi’an Jiaotong University in 2000 respectively.

In October 2000, he became a Ph.D student at (Eindhoven) Embedded System Institute (EESI), The Netherlands. From October 2000 to May 2002, he worked on formal methods for mobile network system design. From June 2002 to November 2004, he carried out his research on predictable real-time system synthesis, which led to this Ph.D. thesis.

Since April 2005, he became a postdoc researcher to continue his research on predictable system synthesis in Electronic System (ES) group at Electrical Engineering Department, Eindhoven University of Technology, The Netherlands.