Time for
Reactive System Modeling

Dipl.-Inf. Ass. iur. Insa Marie-Ann Fuhrmann

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2017

Kiel Computer Science Series (KCSS) 2018/2 dated 2018-12-14

URN:NBN urn:nbn:de:gbv:8:1-zs-00000346-a5
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via `https://www.informatik.uni-kiel.de/kcss`

Published by the Department of Computer Science, Kiel University

Real Time and Embedded Systems Group

Please cite as:

▷ Insa Marie-Ann Fuhrmann. *Time for Reactive System Modeling* Number 2018/2 in Kiel Computer Science Series. Department of Computer Science, 2018. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Fuhrmann18,
  author    = {Insa Marie-Ann Fuhrmann},
  title     = {Time for Reactive System Modeling},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2018},
  number    = {2018/2},
  doi       = {10.21941/kcss/2018/2},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

© 2018 by Insa Marie-Ann Fuhrmann

Herstellung:
BoD - Books on Demand, Norderstedt, Germany

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Reinhard von Hanxleden
Institut für Informatik
Christian-Albrechts-Universität zu Kiel
2. Gutachter: Prof. Dr. Michael Mendler
Fakultät für Wirtschaftsinformatik und
Angewandte Informatik
Otto-Friedrich-Universität Bamberg

Datum der mündlichen Prüfung: 19. Februar 2018

Zusammenfassung

Reaktive Systeme interagieren mit ihrer Umgebung, indem sie in Zyklen, sogenannten *Ticks*, Eingaben einlesen und Ausgaben errechnen und zurückgeben. Meistens sind reaktive Systeme sicherheitskritisch. Sie werden typischerweise in spezialisierten Modellierungstools entworfen und die dazugehörigen Modellierungssprachen sind auf die formale Analyse der Konstruktivität optimiert, die eindeutige Ausgaben garantiert. Dies wird unter anderem durch eine bewusste Abstraktion von Laufzeitaspekten erreicht. Dennoch ist die maximale Laufzeit eines Ticks ein bedeutender Aspekt, denn sie bestimmt die mögliche Interaktionsfrequenz zwischen Umwelt und System und damit die Rechtzeitigkeit von Reaktionen. Dementsprechend muss bei der Systementwicklung darauf geachtet werden, dass die Spezifikationen bezüglich der maximalen Ausführungszeit eines Ticks gewahrt bleiben.

In dieser Arbeit wird ein genereller Ansatz der interaktiven Ausführungszeitanalyse vorgeschlagen, der diese Aufgabe durch die Anzeige aktueller und detaillierter Zeitwerte in der Modelldarstellung erleichtert. Das Konzept hierzu beruht auf einer übergreifenden Schnittstelle, über die wechselnde Modellierungs- und Analysewerkzeuge flexibel miteinander verbunden werden können, welches Vergleiche und die Wiederverwendung von Testmodellen erleichtert. Der vorgestellte Ansatz ist sowohl für datenflussbasierte als auch für zustandsbasierte Systeme geeignet. Dies wird durch das Konzept ermöglicht, dass anstelle von Funktionen beliebige Codeabschnitte unter Analyse gestellt werden können. Der vorgeschlagene Ansatz umfasst zusätzlich die optische Hervorhebung von zeitkritischen Modellbereichen und den Einsatz von Konzepten aus der Modellierungspragmatik.

Das Konzept wurde praktisch mit einer Eclipse-basierten Implementierung in Open-Source-Software ausgewertet; dies schließt die Anwendung in einer Benutzerstudie ein.

Abstract

Reactive systems interact with their environment by reading inputs and computing and feeding back outputs in reactive cycles that are also called *ticks*. Often they are safety critical systems and are increasingly modeled with highlevel modeling tools. The concepts of the corresponding modeling languages are typically aimed to facilitate formal reasoning about program constructiveness to guarantee deterministic output and are explicitly abstracted from execution time aspects. Nevertheless, the worst-case execution time of a tick can be a crucial value, as it decides the frequency in which the system can interact with its surroundings. An excessive tick Worst Case Execution Time (WCET) can lead to lost inputs or tardy reaction to critical events. Thus, the modeler has to make sure that the timing behaviour of the system under development meets the specifications.

This thesis proposes a general approach to interactive timing analysis, which enables the feedback of detailed timing values directly in the model representation to support timing aware modeling. The concept is based on a generic timing interface that enables the exchangeability of the modeling as well as the timing analysis tool for the flexible implementation of varying tool chains. This aims at enhancing the comparability of tools and facilitates the sharing of benchmark model suites. The introduced approach is applicable not only to dataflow-based systems, but also to state-based systems. The latter is enabled by a concept for communicating analysis requests and responses for arbitrary code parts instead of a restriction to function granularity. The proposed timing analysis approach includes visual highlighting and modeling pragmatics features to guide the user to WCET hotspots for timing related model revisions.

The approach is practically evaluated with an open-source Eclipse-based example implementation for the modeling language SCCharts, which includes a user study.

Acknowledgements

Foremost I thank my advisor Reinhard von Hanxleden who enabled me to take part in the faszinating DFG projects PRETSY and PRETSY 2[1], which thankfully also supported this thesis. Reinhard von Hanxleden initiated the topic of this thesis and contributed lots of time, ideas and advice. He made it possible for me to register to several workshops and conferences and take part in international meetings. This helped me to form scientific connections I treasure very much. I am grateful in equal parts for the advice and the leeway needed for a grown ex-lawyer to travel the realms of doctoral studies in computer science safely. Education in the RTSYS group in Kiel also prepared me well for working in industry. And finally, it included a lot of valuable and fun experiences on and beyond the campus.

Also I wish to thank the Co-Leader of the PRETSY projects, Michael Mendler very much for warmly welcoming me as a part of the team, for his encouragement and countless hours of inspiring discussions and valuable feedback. I remember our workshops in Bamberg, Kiel and Paris fondly. I broadened my knowledge in informatics theory under his guidance. As a side effect I learned to know and value the Franconian Switzerland and its gentle people.

I am grateful to have worked with Joaquín Aguado in the project, whose teaching talent and patient explanations were invaluable in getting started on formal discussions about constructivity and language semantics.

To all three I owe many thanks for including me in the research and authors team for the fundamental publications on Sequential Constructiveness.

Thanks to all members and supporters of the PRETSY team, which includes also Partha Roop, Marc Pouzet, Gerard Berry, Edward Lee, and the members of their groups, for the exceptional cooperation and also for superb human companionship. I had a most instructive and inspiring time.

Special thanks also go to David Broman who was the main partner on the interactive timing analysis project and devoted a huge amount of time to discussions, virtual and personal meetings, and the preparation of publications. He was the co-father of the timing analysis interface and

[1]DFG HA 4407/6-1 and 6-2

complemented my modeling tool perspective with his precious knowledge of the timing analysis tool side. He was also the one who wrote the first formalization of the timing analysis interface. I am also grateful for his helpful feedback and advice and his ability to give impulse and encouragement exactly when it was needed and to make me feel comfortable during my first steps in the international science community.

Furthermore, I am very grateful to Alexander Schulz-Rosengarten, who helped me very much in leveraging his general tracing approach for KIEL Integrated Environment for Layout Eclipse RichClient (KIELER) Sequentially Constructive Charts (SCCharts) for the interactive timing analysis. Also, thanks to Steven Smyth and Christian Motika who have supported me with their comprehensive knowledge about the compilation and code generation processes of the KIELER SCCharts tool.

I thank also the participants of the RePP workshop 2014, the Synchron workshop 2015, and the RTNS conference 2016 for their constructive feedback.

I have been blessed to be part of a working group of exceptionally skillful, nice and humorous colleagues. Everybody was ready to offer a hand when needed, often no matter his own personal workload. I have learned a lot from each of you and also had a great time, for the RTSYS people do not only know when to work overtime, but also when to have a beer together. Thank you all so very much. Special thanks go to Miro Spönemann. He was my mentor for my Diploma thesis and some of his advice is still effective for me today.

Finally I would like to thank from my heart my family and friends, especially my mother, always the championesse for education and a magician in supporting without smothering, my father for his encouragement and backup, my brother, who infected me with the computer science bug in the first place, and two great women who are always sunshine on a rainy day: my friends Tina and Bärbel.

Contents

1 Introduction **1**
- 1.1 Modeling of Reactive Systems 1
  - 1.1.1 Reactive Systems 1
  - 1.1.2 High-level Modeling Languages 2
  - 1.1.3 Graphical Modeling of Reactive Systems 5
- 1.2 Timing Analysis and Synchrony Hypothesis 8
- 1.3 When Modeling Calls for Timing 12
  - 1.3.1 Use Cases from the User Perspective 12
  - 1.3.2 Applications from the Compiler Perspective 14
- 1.4 Contributions 14
- 1.5 Publications 15
- 1.6 Outline 21

2 Motivation **23**

I Fundamentals 31

3 Related Work **33**
- 3.1 Timing Analysis 33
- 3.2 Timing Analysis for Synchronous Languages 35
- 3.3 Feedback of Timing Values and Interactive Analysis 38
- 3.4 PRET Architectures 43
- 3.5 Pragmatics of Graphical Modeling 45

4 Sequential Constructiveness **47**
- 4.1 Basic Concepts of Multiple Variable Access in the SC MoC . . . 47
- 4.2 SCL, pSCL, SCG, SCCharts 49
- 4.3 Operational Semantics 50

4.4 Structural And Priority-Based Analysis 53
4.5 Signal Emulation and Signal Schizophrenia 54
4.6 Relationships of Constructiveness Classes 55
4.7 Important Terms related to Sequential Constructiveness (SC) . 55

5 Definitions **61**

II Interactive Timing Analysis 69

6 Concepts **71**
6.1 Interactive Design Flow 72
6.2 Timing Program Points and Tracing 74
6.2.1 The general concept of TPP 75
6.2.2 Tracing and Backmapping 78
6.3 Categorization of Timing Values 80
6.4 Special Aspects of TPP Placement and its Semantics 85
6.4.1 Timing Program Point (TPP) Placement and Parallel Structures 86
6.4.2 TPP Placement and Loops 89
6.5 Aggregation of Timing Values 97
6.5.1 Aggregation of Fractional Time Values 98
6.5.2 Aggregation of Local Time Values 99
6.5.3 Investigated Subtraction Approach 101
6.5.4 Concept Expansion for Local Value Aggregation . . . 102
6.6 Scale of TPP Combinations 103
6.7 Function Analysis 106
6.8 Value Assumptions 107
6.9 State Representation 108
6.10 Interactive Timing Interface 111
6.10.1 Basic Interface Formalization 111
6.10.2 Interface Expansion for General ITAGs 115
6.10.3 Interface Expansion for Local Value Aggregation . . . 117

7 Time for SCCharts **119**
7.1 At home in KIELER—Technical Background 120
7.1.1 Aim and Scope of the KIELER project 120
7.1.2 SCCharts in KIELER 122
7.1.3 Used Technologies 125
7.2 User Interface 127
7.3 TPP Placement with Tracing 131
7.3.1 Employing the KIELER SCCharts Tracing Framework . . 132
7.3.2 Specific Tracing Aspects 134
7.4 Interactive Timing Analysis and
Code Generation 136
7.5 TPP and Compiler Optimization 138
7.6 Identifying State 139
7.7 Timing Request File Generation 141
7.8 Modeling Pragmatics for Interactive Timing Analysis 142
7.8.1 Focus and Context for Large Models 143
7.8.2 Function WCET on Mouse Hover 143

8 Evaluation **147**
8.1 Test Cases 147
8.1.1 Specifically Designed Test Cases 147
8.1.2 Test Models from the KIELER project 157
8.1.3 Generated Test Models 157
8.2 Artifact submission 160
8.3 Validation of Extended SCChart Features and Timing 160
8.3.1 Weak and Strong Abort 161
8.3.2 Deep and Shallow History 163
8.3.3 Complex Final States 165
8.3.4 CountDelay 168
8.3.5 Signals 168
8.3.6 Suspend 170
8.3.7 Deferred 171
8.4 User Study 171

III Outlook and Conclusion 177

9 Future Work 179
9.1 Modeling Tool Side 179
9.2 The Analysis Tool Side 180
9.3 Additional Applications from the Compiler Perspective . . . 182

10 Conclusion 185

A Overview Test Models 189

B Example Timing Focus 193

Bibliography 197

List of Figures

1.1 Schematic view of a tick in a reactive system. 2
1.2 Overview of the SCCharts syntax. 9
1.3 Clocked Systems. 11
1.4 Timing hotspot. 13

2.1 Motivational example. 24
  a Simple robot model. 24
  b SCCharts model of the robot. 24
2.2 Timing related revision. 26
  a Robot model in SCChart with hotspot highlighting. . . . 26
  b The improved robot model. 26

3.1 Esterel Toolchain by Ju et al. [JKA+08]. 40

4.1 SCL and SCG 50

5.1 Interactive Timing Analysis Graph (ITAG). 65

6.1 Design flow of interactive timing analysis. 73
6.2 Concept of tracing for interactive timing analysis. 81
6.3 Example ITAG with conditional and loop. 82
6.4 Time value categorization example. 84
  a Critical path. 84
  b Fractional time value. 84
  c Local time value. 84
6.5 Infeasible paths and local time value. 85
6.6 ITAG with nested concurrency. 86
6.7 TPP matching in loops. 90
6.8 Simple loop and TPP. 92
6.9 Conditional branch within loop. 94

6.10 Alternative branch in loop. 94
6.11 Path request response. 96
a Critical path with loop. 96
b TPPs on the overall critical path. 96
c TPP list. 96
6.12 Schematic example aggregation. 98
6.13 Local time value aggregation. 100
6.14 Worst case TPPs combination. 104
6.15 Optional combination with a classical analysis tool. 107
6.16 Mutually exclusive inner region execution. 109

7.1 KIELER SCChart modeling perspective. 123
7.2 Preference page. 128
7.3 User interface. 130
7.4 Tracing in KIELER. 132
7.5 Tracing visualization. 134
7.6 Entry action attribution. 135
7.7 Timing effect of unoptimized code generation. 137
7.8 Model with thread communication. 139
7.9 Focus and context view. 144
7.10 Screenshot: Function WCET on mouse hover. 145

8.1 Test models for state based analysis. 149
a Test model CircleWithCalls. 149
b Test model CircleWithCalls2. 149
8.2 Test model Controller. 150
8.3 Test model FunParc2. 152
8.4 Test model MedicalAid. 153
8.5 The local time value of region Basic in MedicalAid 154
8.6 The test model Feeder 155
8.7 The test model MultiWait 156
8.8 Test model SCU_Monitor. 158
8.9 Generated test model model003. 159
8.10 RTNS artifact approval seal. 160
8.11 Test models for strong and weak abort. 162

a Strong abort. 162
b Weak abort. 162
8.12 Test models for deep and shallow history. 164
a Deep history. 164
b Shallow history. 164
8.13 View with processor cycles. 165
8.14 Test model with complex final state. 166
8.15 Complex final state with concurrency. 167
8.16 The test model CountDelay 168
8.17 Test models for deep and shallow history. 169
a Test model for signal pre value. 169
b Test model for signal reinitialization. 169
8.18 The test model Suspend 170
8.19 The test model Deferred 171
8.20 The experiment model PlantController 172
8.21 Experiment working time graph. 175

9.1 Time triggered scheduling. 183

B.1 Focus and context example expanded. 194
B.2 Focus and context example collapsed. 195

List of Tables

4.1 The iur relations for concurrent variable access. 49
4.2 SC related terms. 59

6.1 Timing values for the example in Figure 6.8 93

A.1 Overview table for test models. 192

List of Acronyms

ASC acyclic Sequentially Constructive

ARM Advanced Reduced Instruction Set Computer (RISC) Machine

ASCET Advanced Simulation and Control Engineering Tool

BCET Best Case Execution Time

CFG Control Flow Graph

CPS Cyber-physical System

DAG Directed Acyclic Graph

DFG Deutsche Forschungsgemeinschaft

DMA Direct Memory Access

DRAM Direct Random Access Memory

DSL Domain Specific Language

ELK Eclipse Layout Kernel

GUI Graphical User Interface

IASC *iur*-acyclic Sequentially Constructive

ILP Integer Linear Programming

IPET Implicit Path Enumeration Technique

ISA Instruction Set Architecture

ITAG Interactive Timing Analysis Graph

iur initialize-update-read

JOP Java Optimized Processor

KEP Kiel Esterel Processor

KIELER KIEL Integrated Environment for Layout Eclipse RichClient

KIEM KIELER Execution Manager

KIML KIELER Infrastructure for Meta Layout

KiTT KIELER Transformation Tracing

KLay KIELER Layout Algorithms

KLighD KIELER Lightweight Diagrams

KTA KTH's Timing Analyzer

KTM KIELER transformation mapping

LCA Least Common Ancestor

LWCET Local Worst Case Execution Time

MoC Model of Computation

MDE Model Driven Engineering

M2M Model to Model

OTAWA Open Tool for Adaptive WCET Analysis

PRET Precision Timed

PRETSY Precision-Timed Synchronous Reactive Processing project

PRETSY 2 Precision-Timed Synchronous Reactive Processing project 2

pSCL pure Sequentially Constructive Language

PTARM Precision-Timed Advanced RISC Machine (ARM)

RISC Reduced Instruction Set Computer

RTNS International Conference on Real-Time Networks and Systems

SASC structurally acyclic sequentially constructive

SC Sequentially Constructive

SCADE Safety Critical Application Development Environment

SCChart Sequentially Constructive Chart

SCG Sequentially Constructive Graph

SCL Sequentially Constructive Language

SC MoC Sequentially Constructive Model of Computation

SC Sequential Constructiveness

sct textual SCCharts language

SIASC structurally initialize-update-read (iur)-acyclic sequentially constructive

SLIC Single-Pass Language-Driven Incremental Compilation

SMM Software Managed Multicore

SPM Scratchpad Memory

SSA Single Static Assignment

SYNCHRON International Open Workshop on Synchronous Programming

TLB Translation Lookaside Buffer

TPP Timing Program Point

TPPG Timing Program Point Graph

UML Unified Modeling Language

WCET Worst Case Execution Time

WCRT Worst Case Reaction Time

Chapter 1

Introduction

1.1 Modeling of Reactive Systems

1.1.1 Reactive Systems

Reactive systems are computational systems that interact with their environment. Typically, they are embedded components of entities that are intended to regulate, control, move in, or cooperate with their surroundings. Mostly, these entities are Cyber-physical Systems (CPSs) [Lee08]. For example, a reactive system can be part of an air-conditioning system, an elevator, a car, a train, an air craft, the control system of a nuclear plant, or a robot. Reactive systems have an interface to their environment that allows them to receive and send information. It is in the essence of reactive systems that the information sent is a computed answer to the information received—a reaction.

We say that reactive systems receive *inputs* and compute and feed back *outputs*. If the environment is physical, reading inputs usually consists of sampling sensor values, which is followed by the calculation of appropriate outputs that are sent to related actuators. For example, an air conditioning system might sample a value given by a temperature sensor for a room, compare it with a desired value and compute the necessary regulations. Finally, it might for example send out a message to activate the compressor of the air conditioner to cool the room. Typically, the interaction with the environment is repeated in input-computation-output cycles, which in the following are called *ticks*. One tick is visualized schematically in Figure 1.3. This image shows the reactive system in its environment. The two sides of the communication interface allow for the reading of inputs on one

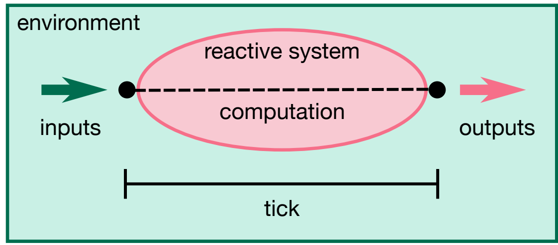


Figure 1.1. A schematic view of a reactive system with its environment, visualizing one tick.

hand and for the feed back of outputs on the other. Together with the computation, these read and write actions form one tick cycle.

The main contribution of this comprises a timing analysis feedback method tailored for reactive system modeling that reflects the characteristics and drawbacks of the modeling languages commonly used in this domain, their implementation of the tick concept, their timing-related abstraction and the resulting specific execution time analysis questions. Thus, in the following I introduce high-level modeling languages for reactive system design, the graphical modeling paradigm and their relation to timing analysis.

1.1.2 High-level Modeling Languages

A large group of CPSs that comprise reactive systems are safety critical and their failure could threaten the integrity of the health and life of humans or at least lead to severe economic loss. Functional correctness, which includes the determinism of the reactions to any specific sequence of inputs, is therefore crucial and its provability is typically required, often by law. A great part of the challenge of this task is due to the requirement that a modeling language for reactive systems has to be able to express functional concurrency, as the environment is often complex and characterized by a number of parallel aspects and potential events to respond to. As there are usually interdependencies, the modeling language must present a solution for the communication between program parts handling different aspects of functionality.

Hence, reactive systems are increasingly designed in high level modeling tools with modeling languages that are abstracted from real-time in order to facilitate formal proofs on reactivity and determinism, like for example the synchronous languages [BCE+03]. A common characteristic of synchronous languages is their discretised view of time, concentrating on the sequence of ticks as discrete *instances* instead of viewing time as a continuum. In synchronous languages, real time can even be viewed as an input event, for example real seconds might be represented by a boolean input *event* `sec` that becomes true when a second has passed. The abstraction is carried even further by the assumption that the computations of the reactive systems are atomic and are completed in zero time. This pair of assumptions is known as the *Synchrony Hypothesis*. With this hypothesis, the programmer can argue about causality without having to consider timing-related aspects. The synchronous languages are devided roughly in two groups, the *data-flow based* and the *control-flow based languages*, both of which will be introduced with examples in the following.

Data-flow languages typically operate on a notion of sequences of values, in which values are related to sequential instances. Programs are mostly expressed as equations or relational systems on those sequences. Two prominent synchronous languages focused on data flow are *Lustre* [CPH+87; HCR+91] and *Signal* [BLJ91; GGB+91]. Both languages provide (possibly infinite) value sequences. In Lustre, they represent variables and are called *flows*, in Signal they are called *signals*. In both languages, common operators like addition and multiplication can be applied pointwise to the sequences. Also, both languages offer the possiblity to express a concept of succession with operators that can define a sequence x to be a delayed version of another sequence y, in Lustre expressed as $y = PRE(x)$, in Signal represented by a delay function $y := X\$1$. Lustre programs can be structured by *nodes* that for a number of input flows define a number of output flows. In Signal, signals can be *composed*, written $S_1|S_2$ for two signals S_1 and S_2. The outcome is called a *process*. Two processes can be composed as well, also denoted with the pipe symbol as $P_1|P_2$ for two processes P_1 and P_2. The effect of composition is that signals with common names will be interpreted as common signals for communication.

Data-flow languages are often *polychronous*, also called *multiclocked* languages, which means that the programs can have a number of clocks. Signal is a characteristic multiclocked language. The concept of clocks in Signal is expressed with the help of an additional value $\perp$, which is used to denote that a signal is *absent* in an instant. An instant is also called *reaction*. Whenever a signal has a value $\neq \perp$, then it is *present*. Thus, each signal has its own clock. It is represented by all instances in which the signal is present. As long as the observer looks at one signal alone, its sequence could be written without the use of any $\perp$ symbols as only the instances matter in which the signal is present [BL90]; the clock of a signal is a local notion in Signal. But as soon as more than one signal is regarded, it is possible that there are instances in which one signal S_1 is present, but another signal S_2 is not. To express this instant, we have to write the value $\perp$ for S_2. However, two signals may have the same clock, also called being *single-clocked*. This is the case, if they are present and absent in the same reactions. Basic operations like plus or multiplication are only defined on signals that are single-clocked. In consequence of the multiclocked concept, Signal provides a filter function, downsampling a signal with the help of a boolean filter signal. It is also possible to create a new signal by interleaving two existent signals with preferation of one of them for instances in which both are present.

Lustre also offers the possibility to define multiple clocks, but in Lustre, the clocks all have to be synchronized to a main clock, whose frequency is the supremum of the other clock frequencies.

In contrast to the typically declarative data-flow oriented languages, there are imperative synchronous languages suited to describe the control-flow of a computation. A widely known control-flow based synchronous language is *Esterel* [BC84; BS91; Ber00; Ber02]. Though a multiclocked version has been introduced [BS01], the original version of Esterel is synchronized to a single clock. Esterel is an imperative language with a set of basic instructions called *kernel statements*. Additionally there are high-level language constructs that can be defined in terms of kernel language constructs, but help to express complex program structures in a shorter notation. A basic concept of Esterel are *signals*, which are considered to be present in a tick if and only if they are emitted in this tick. Consequently,

an *emit* statement is part of the kernel statements. The language allows to use a test for the presence of a signal as conditional to branch control flow. Also there is a loop structure, sequentiality of statements expressed by a semicolon, and a parallel construct to define concurrent threads. It is possible to declare local signals, to preempt execution, and to implement exeption handling. The notion of ticks is represented by *pause* statements. When the control flow of a thread reaches a pause statement, it stops for the rest of this tick and resumes in the next tick. Finally, the execution is stopped forever when the control flow reaches a *halt* statement.

Synchronous languages are typically compiled to a host code representation in a more low-level programming language like for example C. This process is also called *host code generation*.

1.1.3 Graphical Modeling of Reactive Systems

Tools for the design of reactive systems are often based on graphical modeling languages or graphical representations of originally textual modeling languages. Graphical notations can help to directly express characteristics of a model, as they open up the second dimension for the model representation, for example to express containment relationships with the help of spatial inclusion [Fuh11]. A graphical representation can also be employed with the goal to help the user to understand the abstract structures of synchronous languages more easily, especially the relational and equation based systems of data-flow oriented systems. Therefore, graphical modeling languages have been developed as well as modeling languages that have a graphical as well as a textual representation. Typically, the graphical representation of a data-flow based language consists of actor-oriented block diagrams, while control-flow based languages are usually described with state automata.

High-level modeling tools for the design of reactive sytems usually leverage graphical representations. The *Safety Critical Application Development Environment (SCADE)*[1] is based on the language Lustre and is provided by Ansys Esterel Technologies. Models are represented graphically with a focus on data-flow block diagrams. However, there is also a section of

[1]http://www.esterel-technologies.com/products/scade-suite/

the SCADE suite dedicated to modelling with state automata. SCADE is a complete IDE with a strong focus on verification. *Simulink*[2] is also a commercial modeling tool that visualizes models in block diagrams, but it is based on the textual Matlab language, which is not a synchronous language. Matlab is specialized on mathematical computations with matrices and arrays, which in its nature is closer to the data-flow oriented than the control-flow based paradigm. Simulink is specialized on the support of model simulation. Simulink and Matlab are trademarks of the developer MathWorks. The ETAS GmbH has developed the *Advanced Simulation and Control Engineering Tool (ASCET)*[3], which is designed for developing software for the automobile industry and is strongly focused on the generation of C code. ASCET supports modelling in a combination of different diagram paradigms, block diagrams as well as state automata, textual languages and equations. Object-based programming is integrated as well as the language C for the use of library functions and arithmetic services.

Well known languages for the control-flow based graphical programming of reactive systems are for example *Statecharts* [Har87] and the related *SynchCharts* [And96b; And96a; And03], which are more specifically inteded for the design of reactive system modeling and also have similarities to the synchronous language Argos [Mar91; MR01]. All three languages are automata-based with a graphical syntax. They provide elements to express hierarchy and concurrency. A difference between the three languages lies in their position to hierarchy-crossing transitions, where source and target state of a transition are not on the same hierarchy level or not contained in the same superstate. These transitions are allowed by Statecharts, but neither in in SynchCharts [And04], nor in Argos, with the explanation that they are comparable with gotos and make it impossible to reason about subsystems [Mar91]. SynchCharts can be directly translated to Esterel. Thus, they benefit from its formal semantics and additionally are an explicit candidate for multiformalism development, using a textual and a graphical representation together [And96b]. A tool that implemented such a multiformalism approach and supported design, verification and generation of hardware and software was *Esterel Studio* [Ber07], but the tool

[2] www.mathworks.com/products/simulink/

[3] http://www.etas.com/en/products/ascet_software_products.php

is currently not available in the original autonomous form. An example for a statecharts based developing tool for reactive systems is the open source toolkit *YAKINDU*[4], developed by the itemis AG. For the Argos language, a dedicated programming environment called *Argonaute* was introduced by Maraninchi, the developer of Argos [Mar89].

Finally, there are open source research tools concerned with graphical modeling. One of them is *Ptolemy II*[5], a successor of the *Ptolemy Classic*[6] environment. Ptolemy II offers a framework for modelling in an *actor oriented* style. This means that Ptolemy diagrams are typically block diagrams with software components that have input and output *ports* to communicate with the environment and one another in concurrent computation. Ptolemy is not restricted to a specified base language and its semantics, but provides the concept of *director* components, which implement the model of computation for the designed system. A director can be defined for each hierarchy level of the model, so that heterogeneous modelling can be investigated. This approach is widened by the possibility to combine actor oriented modelling with finite state machines in the so called *modal models*.

The *KIELER*[7] has also been established as an open source research project. The project comprises an editor for *SCCharts* [HDM+14]. SCCharts target the modeling of safety critical systems and are a dialect of Statecharts. SCCharts will be the language used for the example implementation of the concepts introduced in this thesis.

Similar to Esterel, the language is divided into a basic set of language constructs, called *Core SCCharts* and a derived set of *Extended Features*. An overview is given in Figure 1.2. The overview itself is depicted as an SCChart, the outer frame given by a *root state* called SCCharts_Overview. This state is a *superstate*, which means that it contains *regions*, which contain other states, thus expressing hierarchy. Regions are drawn in the image as white rectangles, for example the two regions labelled Core-SCCharts and Extended-SCCharts. Parallel regions in one state express functional concurrency. States that are not *superstates* are called *simple* states. For every region there is

[4] https://www.itemis.com/en/yakindu/statechart-tools/

[5] http://ptolemy.eecs.berkeley.edu/ptolemyII/

[6] http://ptolemy.eecs.berkeley.edu/ptolemyclassic/

[7] http://rtsys.informatik.uni-kiel.de/kieler

at least one *initial* state, in which the execution starts. Their counterparts are *final* states that represent termination of a region. Control shifts from one state to another with *transitions*, which can be *normal*, drawn as a solid arrow, or *immediate*, represented by an arrow with a dashed line, depending on whether they consume a tick or not, respectively. A state can only be left in the same tick in which it is entered, if it possesses an immediate outgoing transition. Thus, normal transitions represent tick borders in the SCChart. Conditional logic is implemented by transition *triggers*, which are boolean expressions that have to hold if the transition is to be taken. If more than one transition is triggered at a time, they have to be ordered with the help of numeric *priorities*, of which the smallest is considered the most important. Transitions can also have effects, which are statements to be executed in the case a transition is taken. Superstates can be entered by each kind of transition, but in a Core SCChart, they can only be left by normal terminations, which are automatically taken, if each region of a superstate has reached a final state in its execution. Normal terminations are drawn with a green triangle as a start marker and act as immediate transitions.

Several additional features are provided in the Extended SCCharts language part. Examples are shown in the lower region of the picture labeled Extended-SCCharts. For example these features include ways to preemt the execution of a superstate and to declare actions to be executed on entering a state, in each tick during its execution or on leaving the state, called *entry*, *during* or *exit* actions respectively. The Extended Features will be introduced in the rest of this thesis as they become relevant.

1.2 Timing Analysis and Synchrony Hypothesis

The Synchrony Hypothesis is an idealized view to support formal reasoning about causality and keep the programmer's view on concurrency issues simple. Of course, in the final concrete implementation, computations for one of the discrete synchronous steps take time. That this execution time is bounded is a prerequisite for the acceptability of the Synchrony Hypothesis [GGB+91].

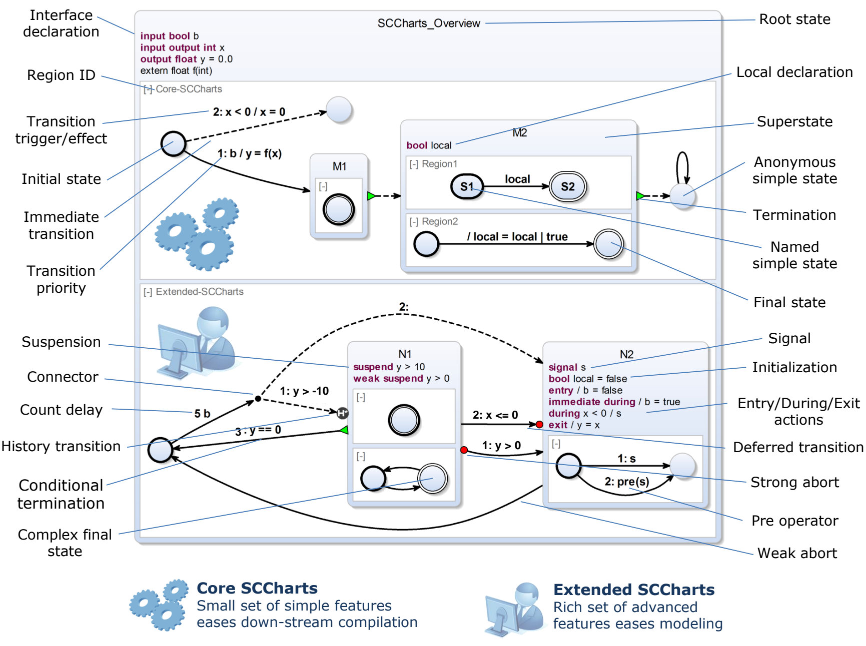


Figure 1.2. An overview of the SCCharts syntax with Core SCCharts in the upper and Extended SCCharts in the lower region. Source: [Mot17]

Also, the computations for one discrete step have to be finished, before the next set of events can be processed. The notion of preemption in synchronous languages differs from that in other programming paradigms. Synchronous languages follow a concept of integrating preemption primitives as first citizens in the language concept, instead of making use of preemption primitives of operating systems, whose loose semantics often make it hard to reason about correctness [Ber93]. This means that environmental inputs that are to trigger any preemptive or exeptional behaviour are read like any other input at the beginning of an instant.

Additionally, if we work in a system with clock ticks in regular intervals, the next sampling of inputs cannot simply start when one computation has

finished. Instead, the implementation must make sure that the next clock does not arrive before the computations of the previous tick are done. This means that the frequency of ticks must be determined by the longest time the computation for any input set in any state of the system may take, the WCET of the execution of one input-computation-output cycle. Note that if not otherwise stated, the term WCET will be used in this sense, i.e. related to one tick, in the remainder of this work.

This concept is visualized schematically in Figure 1.3. The illustration shows the behaviour of an unclocked system in comparison with a regularily clocked system. In both cases the computations for the second tick take maximal possible time, thus they are WCET executions. While in the unclocked system the ticks before and after, which consume less execution time, can trigger the next input-computation-output cycle on finishing, in the clocked system, execution will only be resumed with the next, regular clock tick. For the first and third tick in the image, this will result in idle time.

Though this is a disadvantage of the clocked scheme, it is not only helpful to support high-level programming with the synchrony hypothesis, it brings on predictability, which for safety critical systems is highly important. If a system samples input values and produces outputs at non-predefined intervals, it cannot be foreseen, whether an environment event will actually be perceived and processed in a specific tick or not, while clocks can be defined in relation of the frequency of external events or in relation of the necessities of reaction speed.

Polychronous languages with local clock notions and possibly irregular clocks for signals, like Signal, are bound to according restrictions as well, though they have a more event-based appearance. The calculations for one reaction have to be finished before the next reaction can be calculated. This gets expecially clear when we are programming a register in Signal, whose value is used to calculate an output signal, like Le Guernic et al. [GGB+91] present with their implementation of a mouse double-click. In that example, the authors store the information that a first mouse click event has happened in a register, represented by the derived boolean process `cell`, which outputs the last received value of a signal. The register is reset with a `relax` signal in intervals to limit the time span for a double click. When a click arrives,

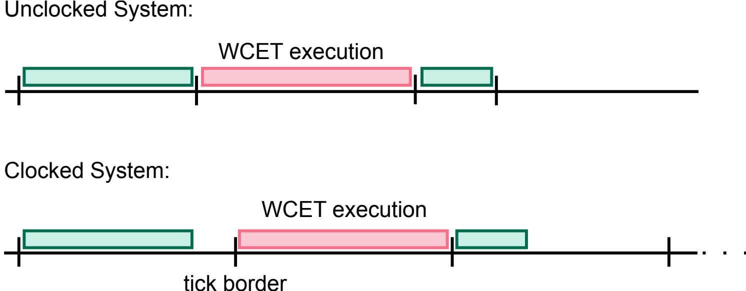


Figure 1.3. In contrast to unclocked systems, for a clocked system the WCET of the computation for one tick determines an interaction frequency.

the register decides whether it is a first or a second click and sets the signal representing a double-click accordingly. So, the calculation of the value register must be finished at every instant, before the next input is processed.

Thus, for reactive systems the WCET of a tick execution is crucial to determine the possible frequency of the interaction. The execution time of a tick may be measured, but as it is hard and often impossible to test every possible constellation, measurements typically do not guarantee that the highest measurement value retrieved is actually the WCET. As implementations of safety critical systems have to be predictable, typically provably safe WCET values retrieved by a timing analysis tool are to be preferred. However, the analysis of the timing behaviour of a program on a processor that is designed for excellent *average case* performance, thus comprising the use of caches and pipeline optimizations, is extremely complex and can only be achieved by introducing safe overapproximations, an overview over this topic is given by Wilhelm et al. [WEE+08]. For a set of processors especially designed to offer timing predictability, the Precision Timed (PRET) architectures, the WCET estimations can be significantly tighter [EL07].

1.3 When Modeling Calls for Timing

The abstraction from real-time described for the high-level modeling languages for reactive systems hides the timing characteristics of the system under development from the modeler and deprives the designer of the possibility to control the timing behaviour of the model. However, timing information is not only hidden on the user level, it is also not available in the compilation process as additional information to base automatical decisions on. In the following, I describe use cases from the modeling process as well as tasks in the code generation, in which interactive timing information feedback can be an asset to the processes. First, I introduce usecases that are related to the modeling process in Section 1.3.1, then I switch the focus to the compilation perspective in Section 1.3.2.

1.3.1 Use Cases from the User Perspective

When modeling a reactive system, there are often restrictions on the value of the overall WCET for the model for a discreet tick. On the one hand, these restrictions can originate in a necessity to react within a specific time span to an environmental event. For example, the air bag of a car is useless, if it opens too late to prevent the driver or passenger to be injured, because the signal to the related actor has not been sent in time. On the other hand, it is typically necessary to guarantee a certain interaction frequency with the environment to avoid missing events. Consequently, common use cases from the user perspective are to:

1. need information on the current WCET of the model in relation to the envisaged target platform, and
2. need information on where to start revising the model, if its WCET currently does not meet the specification.

The requirement for revision guidance is aggravated by the observation that often timing hotspots relate to relatively small parts of the model, but account for large amounts of the overall execution time [BDM+07]. See for example the SCChart in Figure 1.4. The functionality of this model is not

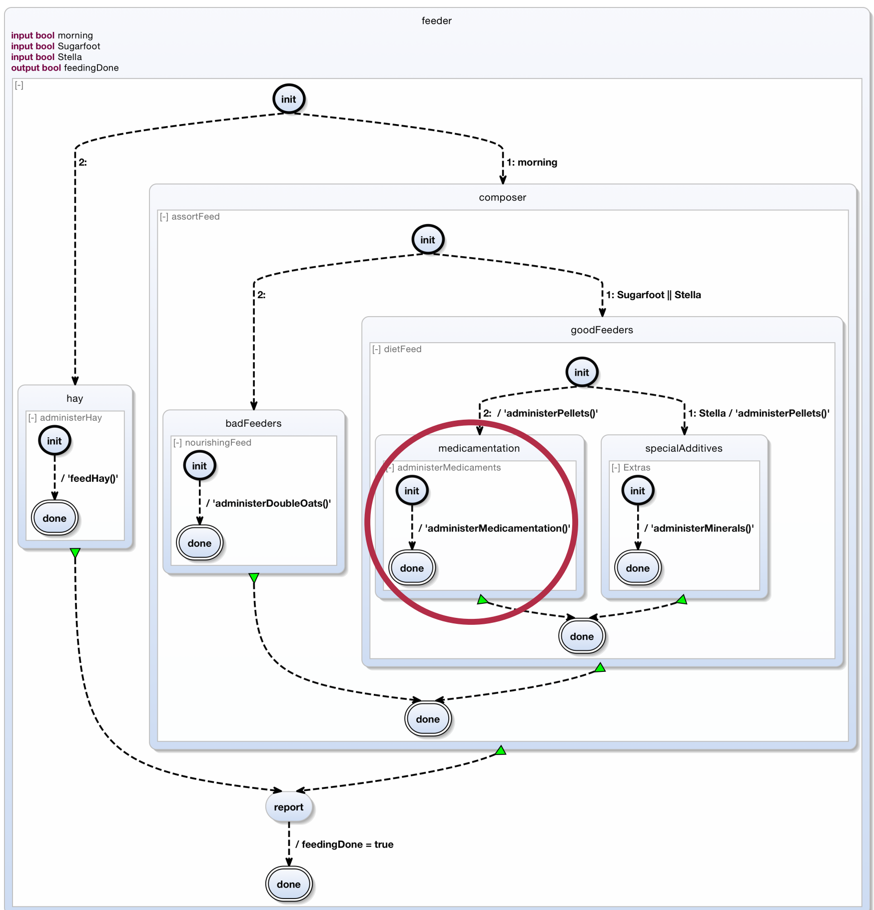


Figure 1.4. An SCCharts model with a concentrated timing hotspot. The hotspot is marked with a red circle in the picture.

important at this point. However, it is an example of a fairly small model that still might require quite some time to revise according to timing-related aspects. Most of the execution time for this model results from a *timing*

hotspot in a small region of the model, which is pointed out in the picture with a red circle. Without knowing where to find this timing hotspot, the modeler has to conduct a complete search with no guidance.

1.3.2 Applications from the Compiler Perspective

While solutions for the usecases from the user perspective are the main contribution of this thesis, there are also tasks related to the compilation and code generation process that can profit from the current feedback of timing values. Two of them are discussed in this thesis:

1. Allocation of code parts to hardware threads for actual parallel implementation with the help of execution time speedup calculation, and
2. scheduling of threads in parallel execution with deadline instructions in case of an implementation on PRET architectures.

If the processor in the final implementation offers the potential for parallel execution of code parts, the compilation needs a process to generate code for the different hardware threads. In this process, it needs to determine which code parts are to be distributed to which hardware thread. If a number of combinations is feasible according to other criteria, we can automatically use current timing analysis information to calculate possible execution time speedups and choose the optimum. Also, if the system is to be executed on a PRET machine, it is possible to schedule hardware threads with the help of deadline instructions by making the execution of a thread wait for needed communication information.

1.4 Contributions

The contributions of this thesis are:

- ▻ An *interactive timing analysis method* with timing *hotspot highlighting*.
- ▻ A categorization of *deep*, *flat*, *local* and *fractional* timing values and a discussion of their characteristics, their aggregation and the complexity of their interactive timing analysis.

- ▻ The augmentation of the Eclipse-based open source modeling tool KIELER with *interactive timing analysis feedback* that leverages model element tracing in the Single-Pass Language-Driven Incremental Compilation (SLIC) [MSH14] approach.
- ▻ A formally defined *generic timing analysis interface* between modeling and timing analysis tools with seperated concerns regarding timing analysis for external function calls and for the tick function.
- ▻ Interface extensions for special problems of timing value aggregation and cyclic programs.
- ▻ The concept of *TPPs* and the discussion of their placement and semantics.
- ▻ The *adaptation of modeling pragmatics techniques* for interactive timing analysis.
- ▻ The introduction of an informal validation technique for the compilation of extended SCCharts features with the help of specified expected timing behaviour of test models.

1.5 Publications

In this section, I shortly introduce my publications and explain how they relate to this thesis. I present the main publications on contributions of this thesis first, then introduce the publications related to semantical foundations and finally refer to advised student theses and an advised internship that resulted in a technical report publication.

The following group of publications concern the main contribution of this paper. In all cases, I was the main author respectively presenter. Coauthors are David Broman, who developed the experimental timing analysis tool and adjoined the related explanations. Also, David Broman was partner in the development of the generic timing analysis interface, taking the timing analysis tool perspective, and wrote the first formalization of the interface. Steven Smyth acted as initial coauthor for aspects related to code generation. Alexander Schulz-Rosengarten implemented the general

model-element-tracing framework in KIELER and assisted me in using it for the interactive timing analysis. Reinhard von Hanxleden initiated the whole interactive timing analysis project, assisted the discussion of related work in these publications and constantly offered advice and reviewing.

[FBS+14a] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden, *Towards Interactive Timing Analysis for Designing Reactive Systems*, Reconciling Performance and Predictability (RePP'14), satellite event of ETAPS'14, April 2014.

In this peer reviewed workshop without formal proceedings, the concepts of the interactive timing analysis approach introduced in this thesis were presented for the first time.

[FBS+14b] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden, *Towards Interactive Timing Analysis for Designing Reactive Systems*, Technical Report, EECS Department, University of California, Berkeley[8], April 2014.

This is a technical report publication of the paper presented at the RePP'14 workshop.

[FBH15] Insa Fuhrmann, David Broman, and Reinhard von Hanxleden, *Interactive Timing Analysis for Designing Reactive Systems*. This was a presentation at the 22nd International Open Workshop on Synchronous Programming (SYNCHRON), Kiel, Germany, December 2015. It was the first introduction including the complete toolchain.

[FBH+16] Insa Fuhrmann, David Broman, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden, *Time for Reactive System Modeling: Interactive Timing Analysis with Hotspot Highlighting*.

This is the conference publication of the interactive timing analysis with the complete example toolchain at the 24th International Conference on Real-Time Networks and Systems (RTNS), Brest, 2016. We also submitted an artifact for this paper, including a distribution of the KIELER and KTH's Timing Analyzer (KTA) tools with the example implementation of

[8] `http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-26.html`

the interactive timing analysis (detailed in Chapter 7), a tutorial handout, a suite of example models used in the paper, and a training video. The artifact passed the revision and thus was rewarded the artifact evaluation seal, which reads: "RTNS Artifact evaluated. Consistent, complete, well documented, easy to reuse".

The following publications are about the semantical foundation of the Sequentially Constructive Model of Computation (SC MoC), which is fundamental to the semantic related discussions in this work. Their content will be introduced in detail in Chapter 4. As they are base publications of the Precision-Timed Synchronous Reactive Processing project (PRETSY) project, I was involved in the meetings, in which the fundamental ideas were developed. I am coauthor but not the main author of the resulting papers. I contributed small text parts, images as well as general comments and suggestions in internal reviews.

[HMA+13a] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien, *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*, Design, Automation and Test in Europe Conference (DATE'13), March 2013, Grenoble, France.

This is the base publication on Sequential Constructiveness.

[HMA+13b] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop, *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1308, August 2013.

This technical report is an extended version of the aforementioned paper, offering more detailed views on constructiveness analysis, the relationship of variables and signals and on SCCharts.

[AMH+14a] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann, *Grounding Synchronous Deterministic Concurrency in*

Sequential Programming, Proceedings of the 23rd European Symposium on Programming (ESOP'14), April 2014, Grenoble, France.

This paper is concerned with proving that Berry's constructive semantics is a conservative approximation of the SC MoC.

[HMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop, *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*, ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design, July 2014.

This is an extended journal paper presenting the evolvement of the DATE'13 paper foundations.

[AMH+14b] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann, *Grounding Synchronous Deterministic Concurrency in Sequential Programming*, University of Bamberg, Faculty of Information Systems and Applied Computer Sciences, Technical Report number 94, August 2014.

This technical report is an extended version of the aforementioned ESOP'14 paper, presenting more technical background.

[AMH+15c] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann, *Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency*, Acta Informatica, Special Issue on Combining Compositionality and Concurrency, 52(4): 393-442, 2015.

This journal paper introduces an extended abstract domain for denotational fixed point semantics. It is more precise than the one used in the paper presetend at ESOP 2014 [AMH+14a] and corrects a mistake made in ESOP 2014 publication, which concerned the initialization of variables. The revised semantics is also used to define a new constructiveness class that helps to link synchronous scheduling with imparative programming, ensuring determinism for sequential and concurrent programs. A long version of this paper is published as technical report:

[AMH+15a] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann, *Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency*, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1504, May, 2015. This techical report has also additionally been published from the University of Bamberg [AMH+15b].

I advised the following theses. This work does not built strongly on their results. Nevertheless, I introduce the interrelations shortly.

[Dud12] Björn Duderstadt. *A Statechart Dialect with Sequential Constructiveness*, December 2012, Diploma thesis, fellow advisors: Christian Motika, Reinhard von Hanxleden. This thesis was part of the development process of the SCCharts language which will be the language used for the example implementation of the interactive timing analysis.

[Joh13] Gunnar Johannsen. *Hardwaresynthese aus SCCharts*, October 2013, Master thesis, fellow advisor: Christian Motika. As this thesis is concerned with hardware generation in the SCCharts compilation process, it is related to the same netlist based compilation and code generation approach that is used for the example implementation in this thesis.

[Uml15] Axel Umland. *Konzept zur Erweiterung von SCCharts um Datenfluss*, March 2015, Diploma thesis, fellow advisor: Steven Smyth. This work is part of the development process of data-flow modeling in KIELER, which is not part of the contributions of this thesis.

[Wec15] Nis Wechselberg. *Model Railway 4.0*, March 2015, Master Thesis, fellow advisor: Christian Motika. The thesis by Nis Wechselberg is of a technical nature and mainly concerned with a new hardware structure for a model railway installation used for teaching, research and demonstrations.

[Wei15] Tibor Weiß. *Von Nebenläufigkeit zu Parallelität*, Oktober 2015, Bachelor thesis, fellow advisor: Steven Smyth. The thesis of Tibor Weiß is concerned with compilation for parallel execution of SCCharts. Therefore it is related to the topic of using timing information for the optimized

allocation of code to hardware threads. Also, there is a connection to the PRET deadline based scheduling.

[Som16] Dirk Sommerfeld. *Laufzeitmessung für SCCharts auf Lego Mindstorms*, April 2016, Bachelor thesis. Dirk Sommerfeld has implemented a solution to measure runtime of SCCharts on Lego Mindstorms. His results are not directly related to this thesis.

[Fli16] Niclas Flieger. *Comparison of Compilation Approaches in KIELER*, April 2016, Master thesis, fellow advisor: Steven Smyth. The master thesis of Niclas Flieger introduces a framework for comparing different compilers in KIELER and is not directly related to this work with the exception that it is shortly concerned with the effects of compilation on the runtime of generated code.

[Bus16] Jonas Busse. *SCCharts Modeling für Eingebettete Systeme mit limitierten Resourcen*, September 2016, Bachelor thesis, fellow advisor: Steven Smyth. The thesis of Jonas Busse introduces compiling optimizations for SCCharts. His work is also shortly concerned with speedups achieved by optimizations in the compilation process.

[Gri16] Lena Grimm. *SCCharts Debugging*, September 2016, Bachelor thesis. The thesis by Lena Grimm is about debugging possibilities for SCCharts and the implementation of a breakpoint mechanism for the SCCharts editor. The results of her work are not directly related to my thesis.

I also supervised the following internship project:

[Ban12] Subarno Banerjee. *Timing Analysis for the Precision Timed ARM Processor*, Kiel University, Department of Computer Science, Technical Report Nr. 1212, June 2012. This project was concerned with configuring the OTAWA [BCR+10] timing analysis tool for timing analysis for the PTARM processor of Berkeley [Liu12]. The technical report is shortly referred to in the discussion of additional future timing analysis implementations.

1.6 Outline

This introduction is followed by a motivation of the thesis topic with a lead example in Chapter 2. The remainder of the work is then devided in three parts, of which Part I is concerned with laying the groundwork with a discussion of the related work in Chapter 3, followed by an introduction into the SC MoC in Chapter 4. The SC MoC is the underlying Model of Computation (MoC) driving semantical discussions in relation to SCCharts. The first part ends with formal definitions of the basic terms in Chapter 5.

Part II is the main part of this thesis and is dedicated to the description and discussion of the proposed Interactive Timing Analysis method. The concepts are introduced in Chapter 6, while the concrete example implementation is described in Chapter 7, together with the implemented aspects of modeling pragmatics. The evaluation resulting from this implementation is reflected in Chapter 8.

In the last part of this thesis, Part III, I discuss future work and sum up the thesis results.

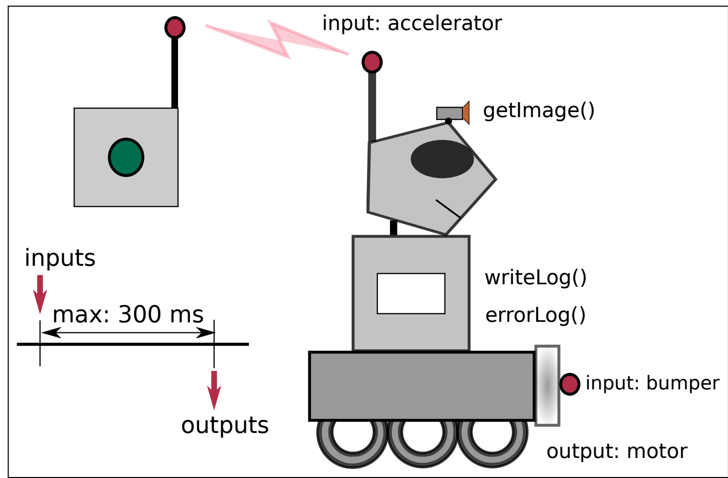
Chapter 2

Motivation

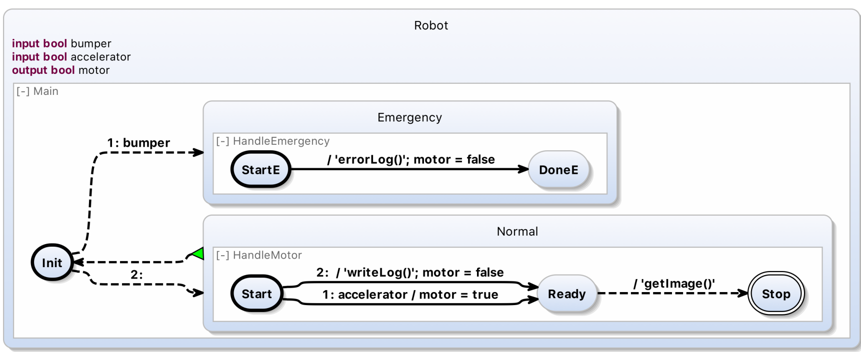
To motivate our interactive timing analysis approach, we turn to a concrete modeling situation. We assume that we want to design a small robot control model as it is sketched in Figure 2.1a. The robot has two boolean sensor inputs. The first belongs to a sensor on its bumper. It indicates a collision when its value is true. The second boolean input is related to the remote control, communicating whether the accelerator button is pressed or not. The system has a single boolean output activating the motor of the robot with a true value or making it stop with a false value. There are three functionalities of the system provided by a library of host code calls. The robot can take photographs with its camera, which is triggered by a call to the getImage() function. Also there are two different functions for logfile writes, writeLog() for the normal case and errorLog() in case of an emergency.

Assume that it is specified that the robot should take images, drive when the accelerator button is pressed, and stand still when it is released. Also the robot should react to a collision by stopping the motor. Also assume that the length of one tick may be no more than 300ms. Otherwise severe damage to the robot might occur. When the robot halts because the accelerator button is not pressed, it should write a normal logfile, in case it stops due to a collision, an error log file is requested.

Figure 2.1b shows an SCChart model of this simple reactive system called Robot with the inputs and output displayed in the left upper corner. The SCChart has three regions, depicted as white rectangels with the region names in the upper left corner. The outer region labeled Main contains two child superstates, Emergency and Normal, each containing exactly one region, labelled HandleEmergency and HandleMotor, respectively. Region Main handles the check on the input bumper, determining whether a collision



(a) Simple robot model.



(b) SCCharts model of the robot.

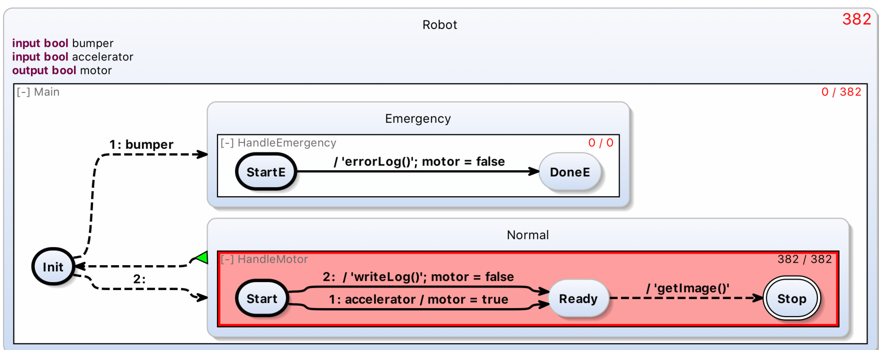
Figure 2.1. Motivational example: A small robot system with two inputs and one output, a maximal WCET specification of 300ms, and three host code functions.

has occurred or not. If bumper is true, control passes immediately to the Emergency state and its HandleEmergency region. This region is responsible for stopping the motor and writing the error log file. After that is accomplished, control rests in the DoneE state and the whole program finishes. If no collision has happened, the Main region imparts control to the Normal state. Its HandleMotor region makes a conditional decision based on the accelerator input. If that input is true, indicating that the accelerator button on the remote control is pressed, the output motor is set to true, causing the robot to drive. If the accelerator input is false, the alternative transition with priority 2 is taken, so that the motor output is set to false and a normal logfile is written using the provided host code call writeLog(). After that, in both cases, the getImage() function is called unconditionally. The HandleMotor region now terminates and control returns to the initial state of the whole program, Init in the region Main.

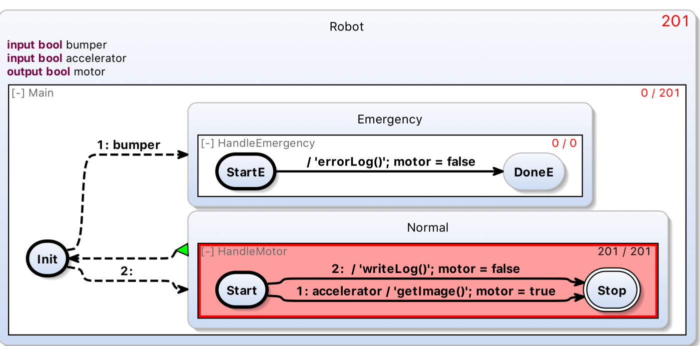
After modeling the robot's functionality, the next step is to determine, whether the timing specification is met. If the modeling tool has no integrated timing analysis, this would mean leaving the modeling process, triggering code generation and invoking a timing analysis tool on the generated code. Assume the modeling tool returns a WCET value for the model code that is above the specification threshold, say 382ms. Then the modeler has to return to the modeling process to revise the model. Furthermore, typically there is no information to guide the designer where to start the revision process. Whichever change the modeler decides to make, for a check whether this revision was successful, the whole loop has to be reiterated.

In this thesis, I propose to incorporate this loop of interaction with a timing analysis tool into the modeling tool. This is automated and complemented by a detailed timing analysis for single model elements as well as highlighting of timing hotspots. Figure 2.2a shows the SCChart model view with interactive timing analysis and hotspot highlighting generated by the augmented KIELER tool.

The overall WCET time value is displayed in the right upper corner of the SCChart root state called Robot, in this case it is a value of 382ms, which is not fast enough according to the specification. The timing constraint is exceeded by 132ms. However, this information alone does not help the modeler to identify the most costly parts of the model, the timing hotspots,



(a) The SCChart for the robot model with interactive timing analysis involving hotspot highlighting.



(b) The improved robot model.

Figure 2.2. The small robot model, in which the timing values change with the functional model revision. For each region, there are two time values, separated by a slash. The first timing value for the region does not include the time spent in child regions, while the value after the slash takes the child regions into account. The overall WCET value is shown in the right upper corner of the model state.

already shortly introduced in Section 1.3.1. Thus, with the overall timing value alone, there would be no guidance as to where the model revision should start. Hence, additionally the tool displays timing values for the regions of the model and also highlights the timing hotspots of the model. Both feedback methods are explained more closely in the following.

The detailed timing values for the regions are displayed in the right upper corners of the regions. There are two values for every region, separated by a slash. The first value denotes the WCET time value of this region without its child regions, while the second value also accounts for time spent in included subregions. For example the first value in region Main does not include time values for the regions HandleEmergency and HandleMotor, while the second incorporates the time values for these child regions. The time values in the illustration are given in milliseconds, which is one of three options in the KIELER tool, the two other variants being a display in processor cycles or in percent of the overall WCET value. The milliseconds are rounded to full milliseconds as typically the designer is not interested in smaller values for the revisions and the rounding enables quick readability. For the region Main alone, this leads to the display of the timing value 0ms. If the modeler switches to a display in processor cycles, it will be displayed that the execution of Main actually takes 10 processor cycles, a value that is irrelevant for the necessary model revision.

All time values given in this example denote the region's contribution to the overall WCET path. We call this the *fractional* time value of a region. Further details on different types of time values are given in Section 6.3. The time value for the HandleEmergency region is reported as zero, because it is not on the critical path for this model.

In addition to the detailed timing value feedback, the regions of the model are automatically highlighted with a background color shade corresponding to their relative timing relevance. Regions that contribute more than 50 percent to the overall WCET are also attributed with a more pronounced borderline in red.

With the help of the hotspot highlighting and the detailed time values, the modeler is quickly guided to the region HandleMotor to start the revision. The WCET of this region alone exceeds the timing specification.

The modeler has now the choice to make a change of one of the following types:

1. Functional changes within the boundaries of the specification,
2. Structural changes, or
3. Replacement of called library functions.

In this case for example, the modeler might find that the specification of the robot system allows to take pictures only in case the robot is actually moving. A resulting functional revision of the SCChart model is shown in Figure 2.2b. In this model version, the calls to writeLog() and getImage() will never be performed in the same tick now. The timing analysis feedback automatically updates the time values and confirms success, as the time value is 201ms now, well below the timing specification.

To work in this close interaction with the changes made by the modeler, the timing analysis has to be fast and detailed. On the other hand, for the usecase of guiding the modeler in timing related revisions, it might be feasible to use a more coarse grained heuristic for the interactive timing feedback and combine this with a classical WCET tool to be used at the end of the whole modeling process, to guarantee tightness and safety of the time values.

The concept for our interactive timing analysis approach is detailed in Chapter 6, an example implementation for the SCCharts editor of the KIELER project is introduced in Chapter 7. Finally, the concept is evaluated with the help of this implementation in Chapter 8.

Part I

Fundamentals

Chapter 3

Related Work

This thesis is related to publications in several areas concerned with timing analysis, tool chain integrations, execution platforms designed for timing predictability, and modeling pragmatics. This chapter contains an introduction to these related publications, starting with general research on timing analysis in Section 3.1. After that, specialized approaches to timing analysis for synchronous languages are discussed in Section 3.2. A positioning of this thesis in relation to other approaches for the integration of timing analysis in the modeling process can be found in Section 3.3. This is followed by a survey of work on PRET architectures, which offer special advantages regarding timing analysis in Section 3.4. Finally, related work on the pragmatics of graphical modeling is introduced in Section 3.5.

3.1 Timing Analysis

A large amount of research work has been publicated on the topic of WCET analysis of programs. A comprehensive overview is provided by Wilhelm et al. [WEE+08]. Several general techniques for WCET computation have been proposed. Early approaches [PK89; CP01] suggested a tree-based analysis method, where the syntax tree of the program is traversed recursively, computing the execution time for each node based on its type and corresponding rules. After the method had not been favored for some time because it did not allow to use flow facts directly to prune out infeasible paths, it has been revived by Harmon et al. [HSK+12]. This work was developed in the context of interactive timing analysis, and it leveraged one of the main advantages of the tree-based approach, namely that it performs fast and scales well, thus enabling fast roundtrip times.

Another method is the explicit path analysis, which performs computations to find the most costly program path directly on a control flow graph of the program. Kleinsorge et al. [KFM13] propose a comprehensive explicit path method that is not restricted to special kinds of Control Flow Graph (CFG) but works for general directed control flow graphs.

A method that also searches for the critical program path, but on an implicit level by solving an Integer Linear Programming (ILP), is the wide spread Implicit Path Enumeration Technique (IPET) approach introduced by Li and Malik [LM97]. For this approach, blocks and edges of a CFG are annotated with their execution time as well as with the number of their execution traversals. Then a system of constraints is built based on the structural constraints of the program and a function computing the program executing time is maximized.

Many challenges in the field of WCET analysis arise from the need to analyze processing features that have been introduced to improve the average performance of processor architectures. These features include cache hierarchies, branch prediction and complex pipelining devices with several strategies for improving pipeline hazards and optimizing pipeline throughput, all of which make timing analysis exceedingly complex and often lead to severe conservative overestimations [HLT+03]. Ferdinand et al. [FW99] present an approach for predicting the cache behaviour of programs with the abstract interpretation technique. This is a general static analysis method that investigates dynamic program properties based on the design of approximate abstract semantics, which was introduced by P. and R. Cousot [CC77; CC92]. Healy et al. [HAM+99] combine the analysis of cache and pipeline analysis. They also present one of the first approaches for a timing analysis user interface for C programming, also presented by Ko et al. [KHR+96] and more closely explained in Section 3.3.

Abstract interpretation as well as model checking have been critically discussed by Wilhelm [Wil04], who favors a combination of abstract interpretation and ILP over model checking on the grounds of inferior performance and scalability of the latter. Metta et al.[MBB+16] introduce a method to improve scalability of model checking for WCET analysis.

Another important topic of WCET analysis research is the detection of infeasible program paths, which means that only possible program flow is

respected for analysis, which typically results in tighter estimation values. For instance, Gustafsson et al. [GES+06] present an approach for pruning infeasible paths that is based on abstract execution. This means that the program is executed on abstract values, for example integer intervals that represent numeric values. These intervals can be narrowed and also decided during the abstract execution. This can not only decide about the feasibility of paths, but also on loop bounds, which is also a crucial aspect for WCET estimation and often solved only through programmer annotations and flow fact files provided by the designer.

3.2 Timing Analysis for Synchronous Languages

The analysis of the WCET of synchronous programs has a special focus on the WCET of a tick, often denoted as Worst Case Reaction Time (WCRT) [BTH08; RAV+09], because *reaction* is used as a term for a synchronous step [BCP+01]. In clocked systems, this value and the number of ticks, derived by a high-level analysis, decide the critical program execution time. The worst-case performance is an important factor, as synchronous languages target the design of safety-critical systems and therefore timing correctness is as important as functional correctness.

Li et al. [LLB+05] introduce the Kiel Esterel Processor (KEP) and a specific WCRT analysis method for this architecture. KEP is a reactive processor which handles instructions that are close to Esterel and which is not designed to optimize average runtime, so that no cache hierarchy, pipeline issues, branch predicitions or similar issues have to be considered in timing analysis. Its concept therefore simplifies WCET analysis in a fashion that is closely related to the PRET architectures introduced in Section 3.4. Another research work on WCRT estimation for the KEP has been presented by Boldt at al. [BTH08]. The authors introduce an algorithm based on a graph representation, the *Concurrent KEP Assembler Graph*. The method analyzes which threads terminate instantaneously and associate each statement with the number of instantaneously reachable instructions to find a maximum over all nodes. This thesis follows a general concept, as it aims for a general interface between modeling tools and timing analysis tools, which consequently targets

different processors. However, specialized processors like the KEP could also be involved, possibly with low effort due to the specialized instruction set level that simpilfies tracing and assignment of time values.

Mendler et al. [MHT09] propose a WCRT algebra as a groundwork for timing analysis targeting Esterel-style reactive processors. The targeted architectures offer hardware multithreading. The algebra is designed to express the imperative synchronous languages. In terms of the proposed algebra, this approach builds interface types for program fragments that create compositionality of WCRT analysis with the aim to enhance performance of the analysis computation for larger programs. In addition, the algebraic approach aims for flexibility as to the degree of exactness and the support for various timing abstractions.

A comprehensive work on algebraic formalization for a range of topics around discrete event systems and Petri Nets has been presented by Baccelli et al. [BCO+92]. This includes an elaboration on the Max-Plus Algebra, which can be employed for timing analysis of systems with parallel execution, as explained in Section 6.4.1.

Wang et al. [WRA13] introduce an ILP based approach called ILP_c for the timing analysis of synchronous languages. The approach reduces state space by ruling out state combinations that will never execute in the same tick, analyzing *tick alignment* [RAV+09]. An ILP model is refined from a version that does not respect tick alignment considerations by iteratively checking the actual alignment of ticks in the computed WCET path.

Raymond et al. [RMP+13] present an approach for the detecting of infeasible paths in synchronous programs to enhance timing analysis. The authors aim to leverage the semantic information that is known at the design level to improve the tightness of estimated WCET on binary code level. The considered language is Lustre [CPH+87], the used timing analysis tool is Open Tool for Adaptive WCET Analysis (OTAWA)[1]. Also the Lustre model-checker Lesar [Ray08] has been employed to detect infeasible paths in the model. The authors introduce a traceability analysis tool that links high-level expressions to binary conditional branches for the compilation process from Lustre to C code to binary code. With the help of high-level

[1] http://www.otawa.fr

information and the traceability result, the tool refines the ILP formulation of the WCET problem generated by the OTAWA tool to remove infeasible paths. This approach is related to the approach of pruning out infeasible paths with the help of state assumption information proposed by this thesis in Section 6.9, which also helps to leverage high-level information for low-level timing analysis.

Logothetis et al. [LSM03b] enhance the Quartz [Sch01a; Sch01b] language tool framework with low-level runtime analysis. They represent real-time systems as Kripke structures [LS01] and achieve to back-annotate timing values to the representation of all single transitions of the synchronous program. Their goal is timing-related system verification. The approach shows that detailed timing values are retrievable, but the granularity is fixed on transitions in the formal representation. The authors also present an approach for taking arithmetic instructions into account [LSM03a].

Mendler et al. [MRB16] introduce an approach where timing semantics is integrated into an algebraic semantics of synchronous programs. In this, they address the problem that timing correctness is essential for synchronous problems, but is not integrated into the formal semantic discussion of synchronous languages. The authors propose a holistic, compositional semantics that integrates Gödel-Dummet min-max-plus algebra [Dum59] and formalizes the tick alignment problem [RAV+09].

Kuo et al. [KSR11] present a WCRT analysis approach for synchronous programs that is based on a reachability analysis, which is less complex than model checking. It has been implemented for the PRET-C language, which is a synchronous language based on C [ARG10].

Also for PRET-C, Andalam et al. [ARG11] introduce a method for pruning infeasible paths that is based on model checking. This approach analyzes the abstracted state-space, which is related to the state assumption approach in the interactive timing analysis interface proposed in this thesis.

3.3 Feedback of Timing Values and Interactive Analysis

There are several tool chain integrations to link modeling or programming tools with timing analysis. Not all share the same motivation with the approach proposed in this thesis as it is described in Section 1.3 and Chapter 2 and moreover all concentrate on specific tool combinations, not on a general approach as suggested in this thesis. Nevertheless, most of the publications encounter at least one of the problems that had also to be regarded in context of the approach presented here.

As mentioned above in Section 3.1, the technical challenge that back annotation of timing values demands a fast analysis performance has been addressed by Harmon et al. [HSK+12]. They present a toolchain for the analysis of Java sources that allows the back annotation of functions and program statements. The authors concentrate on a tree-based analysis technique, which they favor over alternative techniques because of its superior performance. However, the approach can lead to considerably pessimistic estimations, especially on benchmarks with nested conditionals and loops. Nevertheless, the ability of the approach to attribute timing values to statements in the Java code could make it a candidate for connection to the interactive timing analysis interface, as it facilitates the derivation of detailed timing values. Such a connection would enable a comparison with other timing analysis tools on practical benchmarks to investigate the claim of the authors that typical programs for real-time systems do not challenge the weaknesses of their approach. We did not choose this approach for the example implementation, as it does not support state-based analysis, which we intended to be part of the evaluation.

A similar approach, also using a control flow tree based quick analysis method for fast interactive timing analysis of Java code, has been proposed by Meng et al. [MSQ17] with the aim to warn the programmer in case a timeout risk occurs during the programming process. In our approach, this feature is replaced by the direct timing value feedback in the model view which keeps the modeler informed about the timing properties of the model.

A low-level toolchain integration for the analysis of C code was proposed by Ko et al. [KHR+96]. The main goal of the authors is to enable the user to annotate the C code with timing constraints, which are checked by the timing analysis. C code and assembly are linked on basic block level in the tool so that the programmer can watch code parts and associated assembly code in two windows side by side. Code parts can be selected with the help of constraint specification and a dedicated constraints window, in which the constraints can be selected. Also code parts can be selected in a main window in the granualities of functions, loops, paths, subpaths, and ranges of machine instructions. Notably, the mapping between source code and assembly does not respect movement of assembly lines due to compiler optimization. The authors explicitly assign the responsibility to ensure that the selected source lines and assembly instructions under analysis match to the user. The approach proposed in this thesis concerns higher-level modeling languages and the specification of code parts for analysis in the generated host code happens automatically in relation to specific model element presentations. Interaction with the modeler on host code level is not envisaged, so the preservation of code part specification information during compilation and compiler optimization must be handeled automatically, as described in Section 7.5.

Persson and Hedin [PH99] introduce an approach for interactive execution time predictions for the use case of aquiring tight but safe upper bounds for task execution times for real-time scheduling. In spite of the different use case, the authors match the aim of the approach proposed in this thesis in that they want to allow the user to access feedback on execution time information throughout development. Their approach targets the language Java, however the prototype introduced in the publication is based on a simplified language version. The method is based on reference attributed grammars [Hed00], which are used to implement timing schemata as introduced for different statement types by Shaw [Sha89]. In a later publication, Persson and Hedin [PH00] suggest an interactive timing analysis approach for Java with the same usecase of real-time scheduling and user feedback on task execution times during the development process. The analysis is performed on byte code level and focal points are garbage collection and dynamic binding. The analysis is based on timing schemata

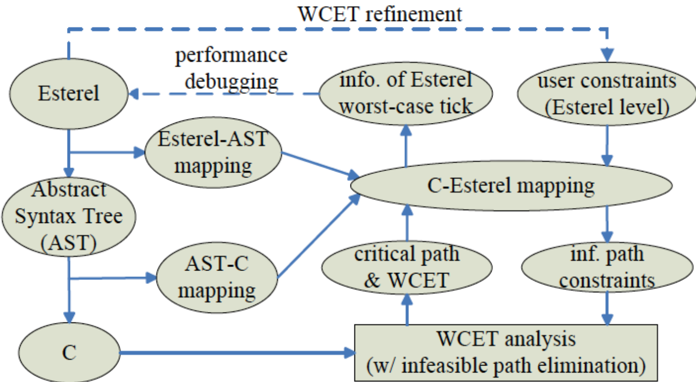


Figure 3.1. The timing analysis toolchain for Esterel by Ju et al. Source: [JKA+08]. The compilation to C is mapped and timing information on the critical tick is made available for performance debugging. High-level information is used in the timing analysis.

that are expressed in reference attributed grammars. Code parts that can be expressed in these terms can be surveyed for their timing behaviour.

The timing analysis approach for Esterel proposed by Ju et al. [JKA+08] is related to our approach in the respect that they introduce a mapping of the relation of their Esterel programs to the generated C code, as illustrated in Figure 3.1, which gives an overview over the framework proposed by the authors. Mapping implementations as these are a prerequisite for a modeling tool to implement the general interactive timing analysis interface proposed in this thesis. Furthermore, the authors simplify the detection of infeasible path patterns in the C code with the usage of syntax and semantics information of the original Esterel program. Partially this covers similar problems as those we solve with the more general approach to enable state assumption information to be passed from the modeling to the analysis tool. Finally, the publication includes back-annotation of timing information and critical path information to the original Esterel program, as we offer for general models. However, they do not differenciate specific timing informations for single model elements.

Another tool chain integration was introduced by Kirner et al. [KLF+02] for Matlab/Simulink models. Like the approach presented in this thesis, their integration is able to derive detailed timing values for model elements. However, their solution is tailored to the concrete Matlab/Simulink application where subsystems relate to task functions in the source code and the model is partitioned in blocks with functions for code generation, especially a function with the algorithm of the block. These block functions are provided with start and stop markers with identifiers that enable the back annotation of timing values. The approach therefore is not as general as the one proposed in this thesis, where model elements are not necessarily represented by parcelled blocks and functions. Even in the Matlab/Simulink context, the proposed approach can only roughly handle the fact that blocks sometimes get overlapped due to optimization features. In that case each block gets assigned the whole execution time of its code, which can lead to overestimation issues. The detailed back annotation of timing values is close to our approach, but does not highlight the timing hotspots of the model. Furthermore the concept does not include the possiblity to annotate arbitrary model elements, the feedback is fixed on blocks and tasks.

Ferdinand et al. present a proprietary approach, the integration of the timing analysis tool *aiT* into the *SCADE* tool. It is close to the approach proposed in this thesis as the authors also aim to feed back timing information to support the modeling process. The tool integration also offers detailed timing values. A main difference to the approach introduced in this thesis is that the toolchain presented by Ferdinand et al. concentrates on analysis for a specific environment and does not aim for generality. In particular, the analysis granularity is limited to function level. Thus, the approach is not suitable for the analysis with a focus on semantical model elements, whose code representation cannot be mapped to generated functions but consists of multiple, unconnected segments in the code. It is therefore for example not applicable for general state-based model representations and arbitrary model elements, though it may be feasible in a setting where state-based representation is superimposed on an internal dataflow-style description.

The concept of TPPs, which we use to enable timing analysis for arbitrary model elements by marking arbitrary, and possibly dispersed areas of related code, see Section 6.2, has not been used by the introduced related

approaches. Relatively close to this concept are the start and stop markers in the Matlab/Simulink approach [KLF+02] described above. These markers are not needed in the same way as our TPPs to delimit arbitrary code blocks, as the code in this approach is structured by tasks, blocks and code generation functions, thus defining specific granularity. The start and stop markers are in the function of identifiers for these structures to enable later back annotation. In contrast, the main functionality of the TPPs proposed in this thesis is to enable specific timing analysis requests for any code area that represents a model element. This concept is general, as it can be applied to models that translate to functions and blocks as well as for arbitrary representations of model elements, for example for state-based systems, especially systems that are compiled to a single tick function.

The low-level approach for C code analyses by Ko et al. [KHR+96] introduced above also is concerned with code part specification for analysis, but as the specification happens on C code level, the programmer chooses directly in the application with the help of direct selection or constraint annotations. The association of assembly happens with a mapping on basic block level. Automatical insertion of markers is not required.

The SAXO-RT compiler for Esterel introduced by Closse et al. [CPP+02] works with so called *control points*, these are not markers but small code sequences that can always be executed in the same order. They are used in code sequentialization and are related to the *basic blocks* that are used for code generation for SCCharts [HDM+14], the modeling language of our example implementation. They are only marginally related to the code blocks marked by TPPs in our example implementation in that they signify possible context switches. However, while these refer to the code generation in case of the *control points*, TPPs are strictly aimed at marking code segments for model element representation in the interactive timing analysis. Thus, arbitrary code parts might be marked, as long as they represent the model element of choice in the timing analysis. Thus, the context switches in this case are those between the representations of model elements.

3.4 PRET Architectures

In 2007, Edwards and Lee [EL07] made their "case for the Precision timed machine". They criticized that processors were optimized for fast average-case performance without regard to the predictability and repeatability of their timing behaviour. Performance speed enhancing facilities like complex cache hierarchies, and Translation Lookaside Buffers (TLBs), as well as deep, possibly superscalar, pipelines with branch prediction and dynamic dispatch mechanism make timing analysis exceedingly difficult and increase the need for very conservative estimations. Additionally, the timing properties of such a system are nearly impossible to predict in advance, neither analytically nor practically, as repeatability of timing behaviour is not guaranteed. Therefore, often analytical methods are replaced by testing. Edwards and Lee argued that this heavily encumbers the design of embedded systems, as the cost for developing systems with certifiable timing behaviour is enormous and improvements cannot be made easily for existing systems. Edwards and Lee provide initial approaches for the design of architectures that offer precise and predictable timing, without disregarding performance issues. They propose to make use of special timing instructions, as introduced by Ip and Edwards [IE06], and *scratch pad* memories [ABS02] instead of caches. These are fast memories, whose allocation is under software control, which makes their behaviour predictable. Also they refer to pipeline interleaving as a method to ensure precise timing for deep pipelines, as elaborated by Lee and Messerschmitt [LM87]. This pipeline interleaving is done by scheduling instructions from multiple hardware threads in an alternate fashion.

Lickly et al. [LLK+08] introduce a simulation-based PRET architecture prototype with fast on-chip scratchpad memories connected to a Direct Memory Access (DMA) controller. Threads are assigned exclusive memory access windows managed by a so called *memory wheel* scheme which ensures determinism and predictability of memory access. The six stage pipeline supports the interleaved execution of six hardware threads. Also notably from the programmer viewpoint, the architecture provides a *deadline* instruction to set up timers that enforce a lower bound on execution time behaviour of threads. These instructions set deadline registers that count to zero and block threads for the corresponding time. These instructions can be used for scheduling thread communication.

Schoeberl [Sch06; Sch08] introduced the Java Optimized Processor (JOP), which is intended as a Java processor for real-time applications that prioritizes low-level WCET analysis over average case performance. It has its own instruction set. It has a two-level on-chip stack cache architecture [Sch05], which is microcode-controlled for predictability, and an instruction cache that stores complete methods [Sch04]. The three-staged pipeline results in short branch delays so that the design includes no branch prediction that would make analysis difficult. Schoeberl et al. [SPL09] also introduce the implementation and usage of a deadline instruction for the JOP. A concrete timing analysis approach based on IPET is comprehensively described by Schoberl et al. [SPP+10], who additionally present an approach based on model checking, which cannot compete with the IPET approach in performance, but facilitates the integration of complex processor models.

Liu et al. [LRB+12] present the Precision-Timed ARM (PTARM), which is also in Detail described by Liu [Liu12]. This is a PRET microarchtecture that implements a subset of the ARMv4 Instruction Set Architecture (ISA). It has a thread-interleaved five-stage pipeline with in-order execution that fetches a different hardware thread each cycle in a round-robin fashion. Thus, pipeline hazards are avoided. However, from the viewpoint of a single thread, its execution is slowed down as it is only assigned pipeline slots in turn with other threads, so overall performance is only unaffected, if all four hardware threads are employed. On the memory side, scratchpad memories and a previously introduced PRET Direct Random Access Memory (DRAM) controller with repeatable access latencies [RLP+11] are integrated. Also the ISA is augmented with additional timing instructions, of which *get_time* obtains the current platform time, *delay_until* is parameterized with a timestamp that is checked against the platform time. When it is not reached, the program is stalled. This is a similar mechanism to the deadline instructions used for scheduling explained above. Additionally, the architecture supports the possiblity to not only guarantee a minimum execution time, but also to handle the case that a maximum execution time is exceeded. In this case, the *Exception_on_expire* triggeres exception handling. The set up exception timers can be deactivated with the *deactivate_exception* instruction.

A PRET architecture specialized on Mixed-Criticality systems called *FlexPRET* has been introduced by Zimmer et al. [ZBS+14]. FlexPRET distinguishes *hard real-time threads (HRTT)*, for which the verification of timing properties must be ensured, and *soft real-time threads (SRTT)*. Threads are scheduled in an interleaved fashion with a concept that guarantees hardware-based isolation to HRTT, which simplifies analysis, while assigning free cycles to SRTT, so that the processor is efficiently utilized. FlexPRET also has timing instructions similar to those of the PTARM.

Special approaches around WCET analysis of PRET architectures have been proposed. Kim et al. [KBC+14] have introduced two approaches for WCET-aware code management for Software Managed Multicore (SMM) architectures, in which each core directly accesses its attributed Scratchpad Memory (SPM) in predictable time, but only accesses the main memory explicitly by DMA instructions. This offers good predictability properties for the usage in interactive timing analysis for reactive systems, which can only be leveraged, if the employed code management technique is optimized for the average-case execution time. Seshia and Rakhlin [SR12] propose an approach for timing analysis that is based on game theorie, which is not only targeting PRET architectures, but takes the timing predictability of the platform into account.

Another interesting tool cooperation, which is related to WCET centered systems, is introduced by Falk and Lokuciejewski [FL10]. By a tight integration of compiler and timing analysis tool, they enable code generation and optimization techniques for WCET reduction.

3.5 Pragmatics of Graphical Modeling

The research field of *modeling pragmatics* is concerned with enhancing the productivity of graphical modeling by supporting the modeler with customized model views for different usecases [Fuh11]. Fuhrmann and von Hanxleden [FH10] introduce the concept of *meta layout* which enables the automatic generation of different model diagram views with the help of automatic layout. The *view management* is then responsible for choosing the right representation with for example the right abstraction level or the

right additional graphical effects to present to the modeler. This can involve giving the user the choice between different view customizations as investigated in detail by Fuhrmann [Fuh11]. The different views can incorporate for example methods to make it easier to work with large models. Such models can usually only be viewed in total, which typically makes the details unreadable, or with just a model part showing on the screen in a zoom mode. The latter makes the detailed information available, but the context of the displayed model region is lost from the view. This is adressed by the *focus-and-context* notion, which means that the part or parts of the model that are in the focus of interest for the specific usecase are shown in greater detail than the surrounding context. Köth and Minas [KM02] introduce a concept to do this by filtering unnecessary information from the context parts of the model, while keeping the details of the focus area. Musial and Jacobs [MJ03] hone this approach especially for Unified Modeling Language (UML) class diagrams. Alternative focus-and-context-views are *fisheye views*. They can be implemented as *optical* fisheye views, in which the focus is enlarged as with a fisheye lens as described by Leung and Apperley [LA94], which also includes distortion of the context areas. Or they represent *graphical* fisheye views, in which the context is drawn smaller than the focus, as presented by Sarkar and Brown [SB92]. Fuhrmann [Fuh11] introduces the notion of *Meta focus*, which means that the focus can be defined by a certain semantic element of the model, like a signal in synchronous languages. This would cause all model parts that semantically reference this element to be emphasized. Also Fuhrmann investigates the possibility to collapse composite states that are out of focus. This latter approach is adopted for the example implementation in this thesis to offer a focus-and-context view for the highlighting of timing-critical regions in a model, as detailed in Section 7.8.1. Common in most pragmatics related features is that their implementation requires the availability of automatic layout, which is given in our example implementation in the KIELER project. Also, their employment should be accompanied by considerations about the mental map of the modeler [MEL+95]. Abrupt layout adjustments that influence the positioning of model parts strongly can be bewildering for the model observer. Animation of the view adaptation can help to lessen this effect.

Chapter 4

Sequential Constructiveness

Sequential Constructiveness is a model of computation which was introduced to reconcile the guaranteed deterministic concurrency of the synchronous MoC and the intuitive and widespread sequential programming paradigm. This chapter offers a synopsis of the publications that introduced SC and investigated the theoretical background of classification and semantics as well as approaches for the analysis of programs [HMA+13a; HMA+13b; HMA+14] Although the findings of these papers are not contributions of this theses, they form part of the foundations for the interactive timing analysis approach, especially with regard to the example implementation introduced in Chapter 7. The following Section 4.1 introduces the basic idea of SC and its approach to ensure determinism for multiple sequential and concurrent variable accesses. Section 4.2 then introduces language representations, which is followed by a description of the operational semantics in Section 4.3. Section 4.4 describes two practical approaches for SC-analysis. Section 4.7 summarizes important term definitions.

4.1 Basic Concepts of Multiple Variable Access in the SC MoC

A fundamental strength of the synchronous languages lies in their guaranteed deterministic handling of concurrent variable accesses. Characteristically, the synchronous MoC requires that a variable is associated with only one value during a tick. The SC MoC is founded on the observation that this stipulation is a sufficient, but not necessary condition for ensuring determinate concurrent behaviour. The SC MoC lifts this requirement by

introducing the usage of sequentiality information for the determination of final variable values for a tick. Variables are allowed to have multiple values in case the order of the corresponding variable accesses is made explicit by sequential statements in the source code. This covers all sequential code without concurrency, thus for example allowing statements like `if (!done)...; done = true`, which would be rejected under the synchronous MoC.

As to concurrent variable accesses that are not ordered with explicit sequentiality, the paper requires that a static compiler analysis must be able to determine a single final value for each variable. This implies that either the accesses can be ordered by a scheduling protocol or it can be determined statically that the order of their execution does not influence the final value.

Two variable assignments are called *confluent*, when their execution order is irrelevant in a given configuration. A class of confluent assignments can be established by design with the help of a combination function $f(x, y)$ on x, so that for all x and all y_1, y_2 the following holds: $f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$. An assignment $x = f(x, e)$, where expression e does not reference x, is then called a *relative* write or an *update* of type f and updates of the same type are confluent. All writes that are not relative are called *absolute* writes or initializations.

The SC MoC requests that all variable accesses can be ordered by the scheduler in the initialize-update-read (iur) execution schedule, in which absolute writes are executed first in the *initialize* phase, relative writes after that in the *update* phase and reading accesses to variables are scheduled after the writes. Accordingly, for two statically concurrent accesses $n_1 || n_2$ to a variable x, we distinguish the *iur* relations as listed in Table 4.1[1]. The listed relations symbolize the scheduling constraints imposed on absolute and relative writes as well as reads in their different combinations. The relation between two initializations or relative writes of different types generally establishes constraints in both directions and thus is called a *conflict*. An exception is given, if the two writes can be established to be confluent nevertheless, for example, because they assign identical values. This exception extends to the overall stipulation of the *iur*-regime: One

[1]Note that the terminology has evolved from [HMA+13a], where the relations ir, iu, ur were called wr, wi, ir with the summarizing notation wir.

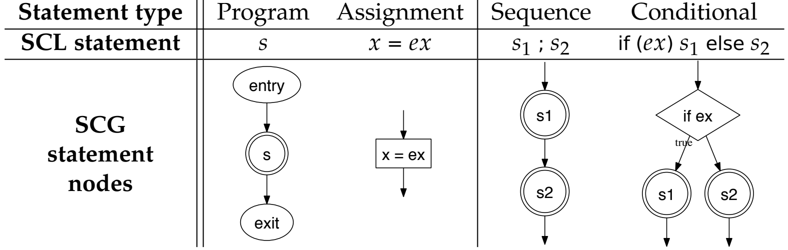
Table 4.1. iur relations concerning concurrent access to the same variable x.

Symbol	**Relation**
$n_1 \leftrightarrow_{ww} n_2$	n_1 and n_2 both initialize x or perform updates of different type, also called a *ww conflict*
$n_1 \rightarrow_{iu} n_2$	n_1 initializes, n_2 updates x
$n_1 \rightarrow_{ur} n_2$	n_1 updates x n_2 reads it
$n_1 \rightarrow_{ir} n_2$	n_1 initializes x, n_2 reads it

absolute write can be scheduled before a number of relative writes of the same type that have to be scheduled before all reads, unless variable accesses that are inserted out of this order can be determined to be confluent.

4.2 SCL, pSCL, SCG, SCCharts

For the illustration of the notions of scheduling, practical program analysis and formal semantics in the SC MoC, the Sequentially Constructive Language (SCL) language was introduced, which consists of assignments, a sequence operator, a conditional construct, a goto, the pause statement indicating tick borders, and the parallel construct to express concurrency. Further, a dedicated CFG-like program representation, the Sequentially Constructive Graph (SCG) is introduced, which includes a representative structure for each SCL statement type. All statements of SCL and their direct mappings to SCG structures are shown in Figure 4.1. The iur-relations can be denoted in the SCG in the form of edges of different types that illustrate the scheduling constraints induced by the concurrent variable accesses. These edges are called summarizingly *iur-edges*, while controlflow-edges are called *seq*-edges, which together with the *tick*-edges form the set of *flow* edges. As a graphical Sequentially Constructive (SC) language, SCCharts have been introduced [HMA+13b; HDM+13; HDM+14], which in their kernel language part directly correspond to representations in SCL and SCG. For an overview of the graphical language constructs see Figure 1.2 on page 9. For formal discussions on semantics, also the language pure Sequentially Constructive Language (pSCL) is defined, which is a minimalistic SC language for pure,



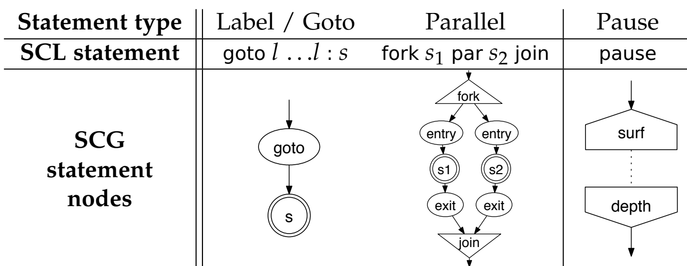


Figure 4.1. SCL statements and the corresponding SCG structures. SCG subgraphs are depicted by double circles, arrows denote sequential edges, and the dotted line signifies a tick edge. The image is adapted from [HMA+13a].

i.e. boolean signal variables with the following abstract algebraic syntax for program P: $\epsilon \mid \pi \mid$ ¡$s \mid$!$s \mid s\ ?\ P : P \mid P \| P \mid P\ ;\ P \mid rec\ p.P \mid p$. This denotes in order an empty statement, a pause statement, a reset, a set, a conditional, a parallel statement, a sequential statement, a label declaration and the goto for the label.

4.3 Operational Semantics

For the description of the operational semantics of SCL in the SC MoC, we differenciate between static *threads* given by the program structure, and the dynamic *thread instances* which relate to thread instantiations in a running program. A thread t has an *ancestor* set, which consists of t itself, its *parent* thread, which is the thread containing the fork that has forked t, denoted

as $p(t)$, and all nested parent threads in the thread hierarchy: $p(p...(p(t)...)$, also written as $p^*(t)$. A thread t_1 is *subordinate* to thread t_2, $t_1 \prec t_2$, if $t_1 \neq t_2$ and $t_1 \in p^*(t_2)$. Furthermore, a thread t_1 is concurrent to another thread t_2, written $t_1||t_2$, when there exists a Least Common Ancestor (LCA) fork. This means that there are threads t_1', t_2' with $t_1 \neq t_2$, directly forked by the same fork node that belong to the ancestor sets of t_1 and t_2 respectively.

We differentiate the *thread execution states* of *disabled*, for a thread that has not been forked, *enabled* for a forked thread, and the three substates for enabled threads: *active, pausing,* and *waiting*, where a pausing thread has reached the end of its tick execution and a waiting thread has forked a child thread and waits for the corresponding join. An enabled thread has a continuation, which consists of the current node instance with regard to SCG node instances and execution context information.

The operational semantics of SCL is based on these *continuations*.

Though generally extensible for further analyses, the additional context here is restricted to the execution status, which can be *active, waiting,* or *pausing*, corresponding to the thread states. In this semantics, the task of the scheduler is modeled by having to choose an active continuation from a finite set of continuations, named the *continuation pool*. If the scheduler is free to choose any active continuation without a protocol, we speak of *free scheduling*. Note that the continuation pool is finite, because the number of instanciated threads at any point of the program execution is finite, even for programs with unbounded instantaneous loops. The continuation pool has to satisfy some constraints that correspond to the characteristics of the continuations themselves. For example, the only continuations in the thread pool of status waiting can be those that are associated to join nodes.

Two elements are evolving in the course of the program simulation: the continuation pool C and the memory M, which attributes values to the variables. We call the pair (C, M) a *configuration*. Within a macrotick, the simulation happens in micro tick steps, in which the scheduler picks one active continuation. As parent threads wait for their child threads, the chosen continuation must belong to a thread that has no child thread continuation in the pool, which is called $\prec$-maximal.

After the continuation is chosen and executed, the set of continuations that is active next is determined and the memory is updated. If there is no

active continuation in the pool anymore, a tick border has been reached and a clock step is performed, where all pausing continuations switch from the surface node of the pause to the depth node of the pause. Note that in case of unbounded instantaneous loops, the continuation pool may never run out of active continuations, so that the macro tick never terminates. This is undesirable, but to rule this situation out is not a question of SC.

How the different kinds of chosen continuations, depending on their node type, influence the evolvement of the continuation pool and the memory can be defined in form of the function $nxt(c, \rho)$ for a continuation c and the memory ρ for the continuation pool and function $upd(c, \rho)$ for the memory update. For further information on these functions refer to [HMA+13b].

Relative to a valid configuration (C, M), two nodes n_1, n_2 in the corresponding SCG are *conflicting*, if there are two active continuations c_1, c_2 with node instances of n_1 and n_2 for which $c_1(c_2(C, M)) \neq c_2(c_1(C, M))$. Two nodes are called *confluent* in C, M, when there is no sequence of micro steps leading from (C, M) to another configuration (C', M'), such that n_1 and n_2 are conflicting in (C', M'). Two node instances are called *confluent*, when the corresponding nodes are confluent in the configuration, in which the first of the two instances is executed.

If two node instances are concurrent, but not confluent in a macro tick, a scheduler that heeds the iur-protocol has to choose its continuations so that the scheduling conditions signified by the iur-edges are met. If this is given for all node instances in a macrotick, it is called *SC-admissible*. A program execution that consists of a sequence of SC-admissible macro ticks is called an *SC-admissble run*. Concurrency of node instances means that they get active in the micro ticks of the same macro tick, belong to statically concurrent threads and their threads have been instanciated by the same LCA-fork instance.

A program is called *sequentially constructive*, when there exists an SC-admissible run for it and every existing SC-admissible run generates the same determinate sequence of macro tick outputs.

Note that SC does not require that the SCG is free of cycles of *seq*-edges, nor that its cycles must include a pause statement, as is prescribed for example in Esterel. To ensure that a program terminates and contains no unbounded instantaneous loop is an orthogonal question to SC.

4.4 Structural And Priority-Based Analysis

If a program has at least one schedule that does not contain a cycle, we call it acyclic Sequentially Constructive (ASC), if it has a schedule without a cycle with a iur edge, we call it *iur*-acyclic Sequentially Constructive (IASC). This includes the absence of a *ww*-conflict that cannot be resolved by confluence. Such a conflict also forms a direct iur-cycle because of the mutual scheduling constraints. Every IASC program is SC. We can perform an approximating SC analysis by analyzing the structure of the program given by its *seq*- and iur-edges. If the schedule given by the structural order is free of cycles with iur-edges, we call it structurally iur-acyclic sequentially constructive (SIASC). If it is cycle-free in general, we call it structurally acyclic sequentially constructive (SASC). The relationship between these classes is characterized by the following implications: SASC $\Rightarrow$ SIASC $\Rightarrow$ IASC $\Rightarrow$ SC and SASC $\Rightarrow$ ASC $\Rightarrow$ IASC.

A dataflow-based compilation approach for SCCharts as well as SCL that sequentializes the SCG with the help of the structural iur-edge information has been introduced and implemented in the context of the KIELER project [HDM+14; MSH14; SMH15; Smy13].

Another method for determining SC schedules is the use of *priorities*. Each node n representing a statement in the SCG gets assigned the maximal number of $\rightarrow_{iur}$ edges traversed by any path originating in n. The scheduler then considers all active nodes in active threads and gives control always to the one with the highest priority. This method can also be used to determine SC, because a program with finite priorities, which also implies that there are no $\leftrightarrow_{ww}$ edges, is IASC schedulable and thus SC.

4.5 Signal Emulation and Signal Schizophrenia

With the help of the iur scheduling regime, pure and valued signals, like those in Esterel for example, can be emulated with variables. For pure signals the emulation with a boolean variable s involves an additional thread that initializes the variable with an absolute write to false in each tick. All emits on the variable are performed as relative writes using the logical *or*: $s = s||true$. A true value stands for presence of the emulated signal, the false value for absence. For a valued signal, the boolean variable is augmented with an integer $scur$ to collect the value emissions. It is initialized with 0 and updated with relative writes with addition as the combination function. The pre operator, as it is given in Esterel for example, is implemented by storing the value of the variable in a fresh buffer variable that is copied to a pre-variable in the next tick. All necessary orderings are handled by the scheduling protocol.

This leads to an advantage of SC regarding the treatment of the schizophrenia problem of signals, which is concerned with signals that become absent and present in the same tick due to loop constellations. Earlier approaches under the sychronous MoC are either based on code transformation and have an exponential worst case code size increase or split loop bodys in their surface and depth parts with corresponding signal copies, which can result in quadratic code size increase [Ber00; SW01; TS04]. Leveraging the signal emulation with sequential variables, this problem can be handled in the SC MoC with growth of the code size linear to the number of signals and thus to the size of the program. This approach is based on a seperate surface and depth initialization of the emulation variable. While the former is placed after signal declaration in the surface part of the code, the latter is situated in a parallel loop that pauses first and starts in its depth part with a signal initialization. The SC MoC offers both the direct access to signal initialization and the iur-protocol that schedules these accesses in the right order.

4.6 Relationships of Constructiveness Classes

The relationships between SC and the program classes SASC, SIASC, IASC, and ASC are described in Section 4.4.

Furthermore, SC can be viewed in relation to alternative notions of constructiveness. For example, it overlaps with *P-constructiveness*, introduced by Pnueli and Shalev [PS91], which allows speculation on the absence of signals. This means that if a signal guards a conditional and it has not yet been emitted, it can be fixed as false to see whether a conflict occurs. This is the case if the signal gets emitted in the course of further execution. In this case the analysis backtracks to find a conflict free constellation. P-constructiveness and SC partly accept the same programs, but neither is a subclass of the other. *L-constructiveness*, as introduced by Boussinot [Bou98], permits speculation on absence as well as presence of a signal to find a *logically coherent* constellation, in which a signal has to be interpreted as absent, if no emission statement is executed. L-constructiveness does not permit unemits. L-constructiveness intersects with P-constructiveness as well as SC, but each class accepts programs the others do not accept. However, each of these constructiveness classes includes Berry's constructivity for Esterel. That prominent constructiveness notion introduced by Shiple et al. and Berry [SBT96; Ber02] poses the question whether the program can be translated into a delay independent boolean circuit. An equivalent analysis technique is the ternary *must/cannot*-analysis, which tries to determine signal values by non-speculative breadth-first absence and presence propagation, which either reaches a fixed point, in which for each variable the value has turned from undefined to either false or true, or deadlocks in wait for a stabilization. The latter leads to program rejection.

4.7 Important Terms related to SC

This section offers a tabluar overview of the important terms that characterize the publications and discussions on SC in the following Table 4.2. The overview aims for intuitive descriptions. For the more formal definitions, the references to the definitions and introductions in the papers are listed for each term.

Term	Description	Definition
SC MoC	Allows multiple values per tick and variable	[HMA+13a], Sec. I
SCL	Minimal SC language	[HMA+13a], Fig. 1
SCG	SC CFG variant, used as an internal program representation for analysis and compilation	[HMA+13a], Fig. 1
Fork	Start of parallel structure (one fork creates two threads)	[HMA+13a], Fig. 1
Thread	Static thread in the sense of a program structure	[HMA+13a], Sec. III B
Thread instance	Dynamic thread instance of program in execution	[HMA+13a], Sec. III B
Parent thread	Thread that contains the fork that created another thread (called child thread)	[HMA+13a], Sec. III B
Ancestor thread	Thread in a nested parent relation to another thread. A thread is regarded as part of its own ancestor set.	[HMA+13a], Sec. III B
LCA fork	Fork deepest in hierarchy that creates an ancestor to two threads	[HMA+13a], Def. 1
Concurrent threads	Concurrent threads share an LCA fork	[HMA+13a], Def. 1
Micro tick	Internal calculation step in a program	[HMA+13a], Sec. I, III A, Def. 2
Macro tick	Mapping R of length $len(R) \in \mathbb{N}$ of micro tick instances to nodes in an SCG	[HMA+13a], Def. 2
Run	Sequence of macro ticks	[HMA+13a], Def. 2

Concurrent node instances	Belonging to statically concurrent threads and the microticks of the same macro tick, instanciated by same LCA fork, clarified in the technical report: LCA fork instance	[HMA+13a], Def. 3, [HMA+13b], Def. 4
Confluent writes	Writes, whose execution order does not matter	[HMA+13a], Def. 4
Identical writes	Writes who set the same value	[HMA+13a], Def. 4
Effective writes	Writes that actually change the value of a variable	[HMA+13a], Def. 4
Combination function	Function $f(x,y)$ on x, so that for all x and all y_1, y_2 the following holds: $f(f(x,y_1),y_2) = f(f(x,y_2),y_1)$; aggregates variable values order-independently	[HMA+13a], Def. 5
Relative writes	Assignments aggregated with a combination function, considered confluent	[HMA+13a], Def. 6
Absolute writes	Writes that are not relative	[HMA+13a], Def. 6
Concurrent nodes	SCG nodes that may execute in concurrent node instances	[HMA+13a], Def. 8
$\prec$ (context: operational semantics)	SCL context: Subordination relation of threads, overloaded for their continuations as well. A thread t_1 is subordinate to thread t_2, if $t_1 \neq t_2$ and t_1 is an ancestor of t_2, also used to denote a preorder for processes, based on instanciation order. In pSCL context: Partial lexicographic order on thread identifiers, reflexive closure in any context written as $\preceq$	[HMA+13b], Definition 2, [AMH+14a], Sec. 2.3, [AMH+14b], Sec. 3 B.

Thread status	A thread is *disabled* before it is forked, then it is *enabled*. An enabled thread can be *active, waiting,* or *pausing*	[HMA+13b], Sec. 3.9
Continuation	A combination of a program (SCG) node instance and execution context information, especially the status: *active, waiting,* or *pausing*.	[HMA+13b], Sec. 4.1
Continuation pool	Finite set of continuations the scheduler is to pick an active execution from	[HMA+13b], Sec. 4.1
Configuration	A pair (C, ρ) of a continuation pool C and a memory ρ assigning values to variables	[HMA+13b], Sec. 4.2
free scheduling	The scheduler is allowed to pick from the pool of active continuations (later: processes) without a protocol	[HMA+13b], Sec. 5.2, [HMA+14], Sec. 3, [AMH+14b], Sec. III B
$nxt(c, \rho)$	Evolvement function for the continuation pool, after continuation c has been chosen and performed by the scheduler, later defined for processes	[HMA+13b], Sec. 4.2
$upd(c, \rho)$	Microstep memory update function for a chosen continuation c and the memory ρ, later defined for processes	[HMA+13b], Sec. 4.2, [AMH+14b], Def.1
iur scheduling	Initializations are scheduled before updates which are scheduled before reads: init;update;read	[HMA+14], Sec. 4.1

iur-edges	Edges representing the relations ww, iu, ir, ur (write confilict, init before update, init before read, update before read)	[HMA+14], Definition 2.1
SASC and SIASC	Program classes, defined by characteristics of the structurally derived schedule given by instantaneous sequential edges and *iur*-edges. Acyclic: SASC. No cycle with a *iur*-edge: SIASC	[HMA+14], Def. 5.4
SC-admissibility	macro tick (or run) meets all *iur* scheduling conditions	[HMA+13a], Def. 7, [HMA+14], Def. 4.7
Sequential Constructiveness	Program has at least one SC-admissible run and every such run generates the same trace of macro tick outputs	[HMA+14], Def. 7
Acyclic SC (ASC) schedulable	Program with a cycle-free schedule (term originally used in the sense of iur-edge-cycle free schedule)	[HMA+14], Definition 5.4
Priority	The priority of a program statement is the maximal number of $\rightarrow_{iur}$ edges traversed by any path originating in the node representing the statement in the SCG.	[HMA+13a], Def. 11

Table 4.2. Guide to important definitions related to Sequential Constructiveness. Where terms are defined identically in more than one of the publications, the first one is named. In case that the definition evolves, also publications that contain a later definition are named.

Chapter 5

Definitions

This chapter defines basic terms that will be used throughout the thesis. The definitions introduced here also form the basis for the interactive timing analysis interface in Section 6.10. This section includes some definitions adapted from the interface definitions in the publications for the RePP'14 workshop with the corresponding technical report and the RTNS'16 conference [FBS+14a; FBS+14b; FBH+16], which are introduced in Section 1.5. For the definition of terms for timing value categorization refer to Section 6.3.

Definition 5.1 (Graph)**.** A *graph* is a pair $G = (V, E)$, in which V is a set of *vertices* also called *nodes* and E is a set of *edges* $e \in \{\{u, v\} \mid u \in V, v \in V\}$.

Definition 5.2 (Directed Graph)**.** A *directed Graph* is a graph $G = (V, E)$ with $E \subseteq V \times V$, which means that the edges are ordered pairs, thus introducing the notion of direction. We also write $u \rightarrow_E v$ for $(u, v) \in E$. The edges of a directed graph are called *directed edges*.

Definition 5.3 (Edge Source, Edge Target, Outgoing Edge, Incoming Edge, Indegree, Outdegree, Sources, Sinks, Adjacency)**.** For a directed edge $e = (u, v)$ we call u the *source* and v the *target* of e and e is termed an *outgoing edge* of u and an *incoming edge* of v. The *indegree* of a node $v \in V$ is defined as $indeg(v) = |\{(u, v) \in E : u \in V\}|$. Correspondingly, we refer to $outdeg(v) = |\{(v, u) \in E : u \in V\}|$ as the *outdegree* of v. We call $v \in V$ a *source*, when $indeg(v) = 0$ or a *sink*, when $outdeg(v) = 0$. The source and target node of an edge are called *adjacent* to each other.

Definition 5.4 (Path, Directed Path, Subpath)**.** A *(directed) path* in a graph $G = (V, E)$ is a sequence of vertices $\langle v_1, \ldots, v_n \rangle$, $n > 1$, v_1 to $v_n \in V$, if $\{v_i, v_{i+1}\} \in E$ $((v_i, v_{i+1}) \in E)$ for all $i \in \{1, \ldots, n-1\}$. $n - 1$ is called the

length of the path. We write path short for directed path in the context of directed graphs. An arbitrary directed path between v_1 and v_n is abbreviated as $v_1 \rightarrow_E^* v_n$, when we want to refer to a specific path between the two nodes, the denoted path will be further qualified. We use $\bar{v}$ to refer to a path within G. We write $\texttt{length}(\bar{v})$ for the length of a path. For a path $\bar{v} = \langle v_1, ..., v_n \rangle$, a subpath is any contiguous subsequence of nodes $\bar{s} = \langle v_i, ..., v_j \rangle$ with $1 \leqslant i \leqslant j \leqslant n$.

Note that a path may contain the same node several times. Where it is necessary to refer to the instances, we enumerate the instances. For example, if v_m is a repeated vertex in a path $\bar{v}$, we identify the instances as $v_{m,i}$ with $1 \leqslant i \leqslant n_m$, where n_m is the number of instances of v_m in the path, in sequential order. See as an example for a path $\bar{v}$ with once repeated nodes v_m and v_o: $\bar{v} = \langle v_1, v_2, ...v_{m,1}, v_{o,1}, v_{m,2}, v_{o,2}, ..., v_n \rangle$.

Definition 5.5 (Reachability)**.** A vertex v is called *reachable* from another vertex u in a (directed) graph $G = (V, E)$, if there is a path $\bar{v} = u \rightarrow_E^* v$. We use the notation $reachable(u, v)$ for the boolean predicate that signifies whether v is reachable (true) from u or not (false).

Definition 5.6 (Directed Acyclic Graph (DAG))**.** A *DAG* is a directed graph in which no path $p = v_0, \ldots, v_n$ exists with $v_0 = v_n$.

Definition 5.7 (Weighted Graph)**.** A *weighted* graph $G_w = (V, E, w)$ is a graph with an edge weight function $w : E \rightarrow \mathbb{R}$.

With these basic graph related definitions, we can now define the ITAG. It is a CFG variant that is used in the definition of the interactive timing analysis interface as an abstract general representation of a function. We do not, however, demand the usage of an ITAG as an intermediate model representation in the compilation process of actual interface implementations.

Definition 5.8 (Interactive Timing Analysis Graph (ITAG))**.** An *ITAG* is a directed graph $G = (V, E)$ with $V = B \dot{\cup} P \dot{\cup} F \dot{\cup} J$, where B signifies *basic blocks* of code, P denotes markers called TPP, and F and J are *fork* and *join* nodes signifying concurrent program structures, detailed below. The ITAG has the following characteristics:

$$\forall v \in B : 1 \leqslant indeg(v) \wedge 1 \leqslant outdeg(v) \leqslant 2 \tag{5.1}$$

$$\begin{gathered}\forall v \in V : indeg(v) = 0 \vee outdeg(v) = 0 \implies v \in P \wedge \\ |v \in V : indeg(v) = 0| = 1 = |v \in V : outdeg(v) = 0|\end{gathered} \tag{5.2}$$

$$\forall v \in P : indeg(v) \leqslant 1 \wedge outdeg(v) \leqslant 1 \tag{5.3}$$

$$\forall v \in F : indeg(v) = 1 \wedge outdeg(v) = 2 \tag{5.4}$$

$$\forall v \in J : indeg(v) = 2 \wedge outdeg(v) = 1 \tag{5.5}$$

$$\begin{gathered}\forall x \in F : \exists!\, y \in J : \\ \forall e = (x, z) \in E : \\ \exists e = (w, y) \in E : \\ \exists \bar{v} = \langle x, z, ..., w, y \rangle : \\ |(v \in \bar{v} : v \in F)| = |(v \in \bar{v} : v \in J)|.\end{gathered} \tag{5.6}$$

The following explanations state the usage of the ITAG structure in the abstract representation of a function f:

The edges of the ITAG represent possible controlflow. Considering the vertices, we have the four different groups, B, P, F, and J, which represent different concepts in the structure of f. B signifies basic blocks of code which are code blocks that are executed sequentially from their beginning to their end. This means that they do not contain entry or exit points except at the beginning or at the end of the block. They are represented by nodes with one or more incoming edges and one or two outgoing edges, as stated by Equation 5.1. If there are two outgoing edges, this structure represents a conditional branch. In this case, the basic block contains a boolean condition, which we denote as $\texttt{cond}(b)$ for $b \in B$. One of the two outgoing edges represents the *true branch* of the conditional, denoting the control flow in the case the condition evaluates to true. The other edge represents the *false branch*, which signifies the control flow in the contrary case. We refer to the true branch as $\texttt{true}(b)$ and the false branch as $\texttt{false}(b)$ for $b \in B$.

P represents markers automatically placed in the function code, called Timing Program Point (TPP). These markers are explained in detail in Section 6.2. Equation 5.2 says that the ITAG has only one source and one sink, both being elements of P. This means that at the beginning and at the end of the function, there is a TPP marker. No $p \in P$ has more than one incoming and one outgoing edge, represented by Equation 5.3.

F and J are sets of fork and join nodes that express parallel execution paths. A fork node starts two concurrent threads, which are joined in a corresponding join node, therefore each fork node has two outgoing edges, see Equation 5.4 and each join node two incoming edges, stated in Equation 5.5. Finally, Equation 5.6 expresses that there is exactly one matching join node for each fork node, though fork-join-structures may be nested. From this characteristic also follows that $|F| = |J|$.

Definition 5.9 (Node WCET, Node Best Case Execution Time (BCET))**.** Let v be a vertex in an ITAG. We define $\texttt{wcet}(v)$ and $\texttt{bcet}(v)$ as follows:

$$\texttt{wcet}(v) : \begin{cases} \text{WCET of basic block represented by } v & v \in B \\ 0 & \text{else.} \end{cases}$$
$$\texttt{bcet}(v) : \begin{cases} \text{BCET of basic block represented by } v & v \in B \\ 0 & \text{else.} \end{cases}$$

Definition 5.10 (Transition WCET, Transition BCET)**.** For $e = (u, v) \in E$ in an ITAG $G = (V, E)$, we define *transition WCET* $\texttt{twcet}(e)$ and *transition BCET* $\texttt{tbcet}(e)$:

$$\texttt{twcet}(e) : \begin{cases} \text{WCET of transition (branch) from } u \text{ to } v & u \in B \\ \text{WCET of forking parent thread of } v & u \in F \\ \text{WCET of represented join} & u \in J \\ 0 & \text{else.} \end{cases}$$
$$\texttt{tbcet}(e) : \begin{cases} \text{BCET of transition (branch) from } u \text{ to } v & u \in B \\ \text{BCET of forking parent thread of } v & u \in F \\ \text{BCET of represented join} & u \in J \\ 0 & \text{else.} \end{cases}$$

Definition 5.11 (Weighted ITAG)**.** An ITAG $G_w = (V, E, w, b)$ with two edge weight functions $w, b : E \to \mathbb{R}$, in which $w(e) = \texttt{wcet}(u) + \texttt{twcet}(e)$ and $b(e) = \texttt{bcet}(u) + \texttt{tbcet}(e)$ for all $e = (u, v) \in E$ is called *weighted ITAG*.

An example weighted ITAG is illustrated in Figure 5.1. The graph has six basic block nodes b_1 to b_6 and seven TPP nodes, P_1 to P_5 as well as

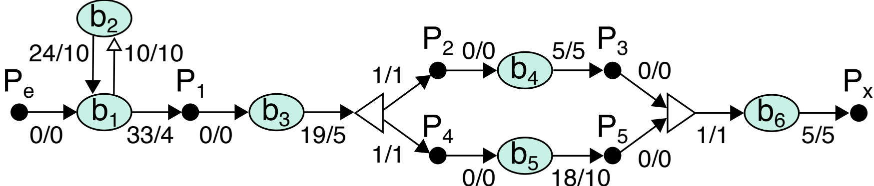


Figure 5.1. A drawing of a weighted example ITAG. The graph comprises representations of six basic blocks, labeled b_1 to b_6, one fork and one corresponding join node, depicted as white triagles, pointing left and right, respectively, and seven TPP nodes, P_e and P_x as source and sink of the graph and P_1 to P_5. Arrows with black heads denote control flow and false branches, arrows with white heads signify true branches. Edge weights are denoted as labels for an edge e with the following legend: $w(e), b(e)$.

the two TPPs P_e and P_x at the source and sink of the graph, of which the latter represent markers for the beginning and end of the function. The edge weights are denoted as labels on the edges, showing the WCET value on the left, separated by a slash from the BCET value. Outgoing edges of TPP nodes have the value 0 for both weight functions. Note that the thesis contains drawings of ITAGs with only one time value denoted in the edge labels. If not stated otherwise, this value is the WCET value. Control flow and false branches, for example for the representation of loop conditions or if-else-constructs, are depicted as arrows with black heads, while true branches are denoted as white headed arrows.

Definition 5.12 (Execution Time of a Path)**.** Let $\bar{v} = \langle v_1, v_2, \ldots, v_n \rangle$ be a path in a weighted ITAG. Then $etime_w(\bar{v} = w(v_1, v_2) + \ldots + w(v_{n-1}, v_n))$ is called the *WCET of* $\bar{v}$ and $etime_b(\bar{v} = b(v_1, v_2) + \ldots + b(v_{n-1}, v_n))$ is called the *BCET of* $\bar{v}$. For an empty path $\bar{v}$, $etime_w(\bar{v}) = 0 = etime_b(\bar{v})$ holds and if $\bar{v} = \perp$ then $etime_b(\bar{v}) = \perp\ = etime_w(\bar{v})$.

Definition 5.13 (Timing Program Point Graph (TPPG))**.** Given an ITAG $G = (V, E)$, its derived Timing Program Point Graph (TPPG) is $G' = (V', E')$ with $V' = P$ and $E = \{(u, v), u, v \in V' \colon\ reachable(u, v)\}$.

Definition 5.14 (Simple ITAG)**.** An ITAG is called *simple*, if the corresponding TPPG is a DAG.

The example graph in Figure 5.1 is a simple ITAG, as the only loop in the program contains no TPP nodes.

Definition 5.15 (Subpath by TPP ($spath_{p_1,p_2}(\bar{v})$))**.** For a path $\bar{v} = \langle v_1, ..., p_1, v_n, ..., v_{n+m}, p_2, v_{n+m+1}, ..., v_{n+m+k} \rangle$ in an ITAG with the TPPs p_1 and p_2, $spath_{p_1,p_2}(\bar{v})$ is the subpath $\langle p_1, v_n, ..., v_{n+m}, p_2 \rangle$ of $\bar{v}$. For a path $\bar{v}$ that does not contain p_1 or p_2 or both, $spath_{p_1,p_2}(\bar{v})$ returns the empty path and if $\bar{v} = \perp$, then $spath_{p_1,p_2}(\bar{v}) = \perp$.

Definition 5.16 (Timing Program Point Path)**.** Let $\bar{v}$ be a path in an ITAG G. If we remove all vertices $v \notin P$ from the sequence of $\bar{v}$, the resulting sequence of nodes is called the corresponding *timing program point path* of $\bar{v}$, denoted as $tpath(\bar{v})$.

Definition 5.17 (Common ancestor fork nodes, Least common ancestor fork node)**.** For two vertices $u, v \in V$ in an ITAG $G = (V, G)$, the set of *common ancestor forks* of u and v is defined as:

$$C_{u,v} = \{f \in F : ((\exists\, \bar{u} = f \rightarrow_E^* u) \wedge (\exists\, \bar{v} = f \rightarrow_E^* v))\}.$$

The *least common ancester fork node* is defined as:

$$lca_{u,v} = l \in C_{u,v} : \forall c \in C_{u,v}, c \neq l : length(c \rightarrow_E^* u) > length(l \rightarrow_E^* u).$$

This accordingly holds for v:

$$lca_{u,v} = l \in C : \forall c \in C_{u,v}, c \neq l : length(c \rightarrow_E^* v) > length(l \rightarrow_E^* v).$$

Part II

Interactive Timing Analysis

Chapter 6

Concepts

To our knowledge there does not exist a published general interface for the integration of reactive system modeling and timing analysis. This chapter contributes the formal definition of such an interface. This definition is based on a number of general concepts which are introduced in the following sections.

The design flow of the introduced interactive timing analysis approach is explained in Section 6.1. Section 6.2 is dedicated to the method of specifying code parts for detailed timing analysis of model elements with the help of TPPs and tracing. In Section 6.3, the possible meanings of time values for model elements are explored and categorized, while the aggregation of retrieved partial time values is detailed in Section 6.5 in relation to these categories. At the background of the introduced aggregation methods, it is possible to investigate the scale of the detailed timing analysis with regard to code part specification, which is elaborated in Section 6.6.

An important aspect of the interactive timing analysis interface is the seperation of concerns between the analysis of called functions and the tick function itself. This concept is the topic of Section 6.7. While this approach aims to enable fast interactive analysis performance, the communication of value related timing assumptions supports the tightness of the estimated time values, as detailed in Section 6.8.

As the interface is designed to be able to deal also with state based modeling languages, we introduce a method to convey information for a state based analysis in Section 6.9. The chapter concludes with the complete interface definition in Section 6.10.

6.1 Interactive Design Flow

As the survey of related work in Section 3.3 shows, the endeavor to integrate the programming or modeling process with timing analysis is not new in itself. However, previous work concentrates on concrete tool chains and does not focus on interchangeability of tools. We follow the different approach of proposing a general integration interface. This approach serves two main purposes:

Flexibility of Toolchains: With the help of the integration interface we can connect a number of different analysis tools to one modeling tool and thus can for example support interactive timing analysis for different hardware platforms or make use of different analysis strategies for a single platform. This facilitates the migration of model design implementation to a different architecture and also makes it possible to customize the interactive timing analysis in a versatile way. Also, the development of a timing analysis tool to this interface instantly establishes usability for all modeling tools that use this integration port.

Comparison of Tools: An important aspect of the evaluation of timing analysis tools is the comparison to other tools that perform analysis for the same architecture. With the help of a general timing analysis interface, any number of timing analysis tools could connect to the same modeling tool. Thus, in a project that combines a modeling tool connected to the interface as well as a benchmark collection, different analysis tools could be compared in this setting, including the possible graphical representation of timing values in the model image. Comparability would also be given for the modeling tool side, for the exploration of the qualities of generated code. This is not only interesting for the comparison of different modeling tools, but also for the evaluation of different compilation approaches in a single modeling tool.

The general interface connects a modeling tool to a timing analysis tool. The design flow of our approach to interactive timing analysis is shown in Figure 6.1. The modeling tool on the top level includes an editing interface for the design of reactive models in a high level modeling language. Also

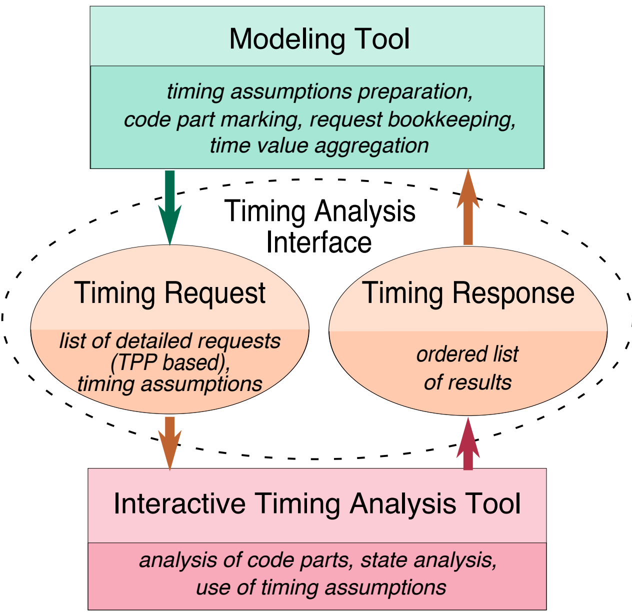


Figure 6.1. The design flow of interactive timing analysis. TPP denotes Timing Program Points.

it includes a compiler for the generation of host code, for example C code. Thus, in this part of the design flow, knowledge about model parts and their representation in the host code exists. Therefore, the modeling tool is in charge of the code part specification for detailed timing analysis and the backmapping of retrieved time values.

Furthermore, on the modeling tool side there may exist additional information on the generated code, for example concerning the value ranges of variables or the representation of state in state based systems. For this reason, the modeling tool has the task to collect and prepare such information for the timing analysis tool in form of timing assumptions.

Also, a notion of different types of time values is necessary on the modeling tool side to differenciate which kinds of timing values are to be displayed to the modeler.

For the display of time values on behalf of the model designer, aspects of modeling pragmatics have to be considered on the modeling tool side as well. For example this concerns in which way timing hotspots may be highlighted, how timing values can be fed back graphically, and how best to convey timing information for large models. These aspects are detailed in the context of the example implementation in Section 7.2 and further in Section 7.8 of this thesis.

The information prepared by the modeling tool has to be conveyed to the interactive timing analysis tool via the interactive timing analysis interface. Aside from the code that is to be analyzed, the modeling tool generates a timing request file that contains the following information:

1. Functions to be analyzed,
2. additional information in form of timing assumptions, and
3. detailed requests for the retrieval of timing information of a certain type for a specified model element.

For the specification of code parts we introduce special markers to be placed in the generated code, the TPP. This concept is explained in more detail in Section 6.2.

The timing analysis tool analyzes the specified code parts and computes a timing value of the requested type for each. The analysis tool uses the timing assumptions to allow for a tighter analysis result. It returns a list of results, one for each timing analysis request, to the modeling tool. The results are returned in the order of the requests, the bookkeeping for the backmapping is assigned to the modeling tool.

6.2 Timing Program Points and Tracing

In our approach to interactive timing analysis, we offer feedback for specified model elements. In our motivational example of Chapter 2 as well as in our example implementation introduced in Chapter 7, the time value feedback is implemented for regions, but this is just an exemplary choice. In general, our interface allows to communicate timing requests and responses

for arbitrary model elements. The core of this concept are the TPPs, which are used to tag code parts that represent the model elements that are subject to analysis. This section introduces the TPP based approach in Section 6.2.1 and provides a conceptual view on the technical challenge of tracing model elements in the compilation process in Section 6.2.2. Special aspects of TPP placement and its semantics, in particular with respect to loops and parallel language constructs, are treated in the seperate Section 6.4.

6.2.1 The general concept of TPP

In the center of the communication between the modeling tool and the timing analysis tool is the generated host code for the model, not its highlevel language representation, whether graphical or textual. Therefore all knowledge of the relation between the original model elements and the generated code lies with the modeling tool. It is part of the technical challenges of interactive timing analysis to communicate as much of this knowledge to the timing analysis tool as needed.

In our interface, this information is represented by a finite set of markers, TPPs, that are inserted into the generated code between each pair of statements, of which the first belongs to another element under analysis than the second. The purpose of this insertion is to enable the modeling tool to express a request for a time value for the code part between a pair of TPPs. See for example the generated C code in Listing 6.1. This code was automatically generated by the KIELER tool for our motivational example from Chapter 2, the improved robot example shown in Figure 2.2b on page 26.

At this point it is not important to capture the exact meaning of the generated code. For a quick orientation it is helpful to know that the code generation approach for this example is sequentializing and netlist based, thus the tick function comprises a large number of boolean guard evaluations that control the execution of certain code parts. Aspects of state representation in the C code are detailed in Section 6.9. The complete example implementation in the KIELER tool is elaborated in Chapter 7.

As in this example implementation the granularity for detailed timing analysis is on region level, a TPP is inserted at each context switch between

```
1  void tick() {
2     // Main, implicit TPP entry
3     if (_PRE_GO == 1) {
4        _GO = 0;
5     }
6     g0 = _GO;
7     // HandleMotor
8     TPP(1);
9     g7 = PRE_g6;
10    _cg7 = accelerator;
11    g9 = g7&&(!(_cg7));
12    if (g9) {
13       writeLog();
14       motor = 0;
15    }
16    g8 = g7&&_cg7;
17    if(g8){
18       getImage();
19       motor = 1;
20    }
21    // Main
22    TPP(2);
23    g1 = g9||g8||g0;
24    _cg1 = bumper;
25    // HandleEmergency
26    TPP(3);
27    g2 = g1&&_cg1;
28    g3 = PRE_g2;
29    if (g3) {
30       errorLog();
31       motor = 0;
32    }
33    // Main
34    TPP(4);
35    g5 = PRE_g4;
36    g4 = g3||g5;
37    // HandleMotor
38    TPP(5);
39    g6 = g1&&(!(_cg1));
40    // SCChart in general
41    TPP(6);
42    PRE_g2 = g2;
43    PRE_g4 = g4;
44    PRE_g6 = g6;
45    _PRE_GO = _GO;
46    return;
47    // implicit TPP exit
48 }
```

Listing 6.1. The tick() function for the improved robot model in Figure 2.2b with TPP.

the representations of two regions. Additionally, we assume that there are always implicit TPPs at the beginning and at the end of the function under analysis. We denote them as *entry* and *exit* or P_e and P_x. These TPPs are needed in most cases, as the request for the overall WCET is expressed as a request for the time value between P_e and P_x. Those implicit TPPs are indicated in Listing 6.1 by comments in lines 2 and 47. The TPPs actively and automatically set in the code generation process can be found in lines 8, 22, 26, 34, 38, and 41. The tick function starts with four lines that are attributed to the Main region, while lines 9 to 20 belong to the representation of its child

region HandleMotor. They include the conditional decision based on whether the accelerator button is pressed or not. The condition is represented by the conditional guard in line 10, which is used via intermediate conditional guards to control whether a logfile is written and the motor is stopped in lines 13 and 14 or a picture is taken and the robot is ordered to drive in 18 and 19. Lines 23 and 24 are attributed to the Main region. Here, the condition that relates to the bumper input is set up as a conditional guard _cg1. Then we switch to the representation of the region HandleEmergency, which spans lines 27 to 32. This code handles the call to the error log file write and the emergency motor stop. After that there is a switch back to region Main for synchronization tasks in lines 35 to 36, followed by a guard in line 39 that sets up information on the bumper input from the current tick for the execution of the HandleMotor region in the next tick, thus this line of code is automatically attributed to the HandleMotor region. In lines 42 to 45, the information needed in the next tick is stored in PRE guards, which resembles register updates and belongs to the state representation, which is more closely explained in Section 6.9. This bookkeeping for the next tick is attributed to the SCChart in general. This means that the execution time for this code part will be included in the overal time value for the model, but not attributed to one of the region values. Note that this code, generated with the default settings of the KIELER tool, has potential for optimization with regard to the number of region context switches. This issue is elaborated in Section 7.4.

The modeling tool can now pose time value requests for pairs of these TPPs with the meaning that the analysis is requested for the code part that starts with the first TPP in the pair and ends with the second. If the second TPP is not reachable from the first, then the analysis will return a time value of zero, which also means that the order of TPPs in the request matters. A TPP can be named in a pair with itself in a request. Such a request will typically yield a zero value, unless the TPP is located in a loop structure. The semantics of TPP placement in loops is detailed in Section 6.4.2. If the program path between the two TPP cannot be proven finite, the time value is reported as undefined.

If for our example the modeling tool now needs information on the time value for the region HandleEmergency, it can request the time value

for the code between TPP(3) and TPP(4). The request with reversed TPPs, i.e. the pair TPP(4) and TPP(3), would result in a zero time value. Like in the example for region Main and region HandleMotor, the representation of model elements may be split in the code. The approach to requesting and aggregating time values for those model elements as well as for nested model elements is elaborated in Section 6.5.

Furthermore, the TPP are also used in the interactive timing analysis interface to request and convey information on the *worst-case path* and *best-case path* itself, respectively. If the modeling tool is interested in information about the path as such instead of its time value, for example for path highlighting, it can use the TPP in the familiar fashion to mark the code part for which the path should be returned. Additionally, the response of the timing analysis tool is defined in our approach with the help of TPP as well. The answer of the timing analysis tool consists of a list of all TPP on the critical respectively least critical path, the least critical path referring to the path associated with the best-case execution time.

6.2.2 Tracing and Backmapping

The concept of working with TPPs involves two principles:

1. TPPs are placed automatically by the modeling tool, and
2. any modeling tool can connect to the interactive timing interface that is able to trace model element representations down to the generated code.

These aspects are regarded in more detail in the following.

The placement of TPPs is done automatically by the modeling tool in the code generation process, so that the modeler is not concerned with placing markers or annotations by hand. This requires on the one hand the tracing of model element representations in the compilation process and on the other semantic-related mapping decisions that are partly language specific. The general concept of tracing and backmapping for the interactive timing analysis is introduced in this section, while the concrete timing related tracing in the KIELER tool and specific issues in context of SCCharts and their netlist based compilation approach are explained in Section 7.3.

Any modeling tool can connect to the interactive timing interface proposed in this chapter, as long as it is able to specify which parts of the generated code belong to the representation of which model elements. As illustrated in Figure 6.2, the code generation process can consist of a direct step from modeling language to host code or it may involve a number of model transformations, which means that the model representation of the system changes without alteration of the overall model semantics.

Model transformations can take place within the domain of the same modeling language metamodel. For example, if the modeling language has a core or kernel language part, the model representation might be transformed to translate all statements to this basic language part. This is done to simplify the following compilation process, as a smaller number of constructs has to be processed. A tradeoff effect of this model transformation is typically an enlargement of the overall model representation, but the tracing of element representations is in most cases straightforward. Constructs are typically replaced in a fixed scheme in which new model elements are directly related to original ones and thus inherit their model element representation relation. For example, if a complex statement X belongs to the representation of a region R and is replaced in the model transformation by the sequence of simple statements A, B, and C, then we can record A, B, and C as belonging to the representation of R as well.

In contrast, there are model transformations in which the original model is expressed in a modeling language with a different metamodel than the target model representation. A prominent example is an intermediate representation in form of a CFG variant. These representations can be harder to trace, as general language concepts like for example hierarchy might not be part of the target language.

Furthermore, constructs might be added to the representation that are part of the compilation concept and not naturally related to original language statements. Whether model transformations of this kind are implemented with a change of the metamodel or not, the allocation of these constructs to model element representations has to involve an explicit implementation decision in the creation of the compilation process. An example for this are guards that are added in a netlist based compilation approach. Concrete examples of tracing-related challenges in our example implementation are detailed in Section 7.3.

For all model transformations, the implementation of the model element representation tracing can be regarded as an integral part of the implementation of the model transformation itself. For metamodel-crossing model transformations as well as those transformations that add compilation related constructs, the implementation requires explicit assignment decisions for constellations that demand strong insight into the language semantics and the compilation process. In conclusion it is possible to add a tracing implementation as a subsequent addition.

However, it is desirable to treat the aspects of traceability already in the design of the compilation process. For the KIELER implementation, the general tracing implementation has been added by Schulz-Rosengarten [Sch14] to an existing compilation process. However, the the SLIC approach in the KIELER project SCCharts compilation by Motika et. al. [MSH14] already had traceability in mind and facilitates it by making the sequence of model transformations in the compilation explicit and noncyclic as a single pass process. As illustrated in Figure 6.2, if the necessary tracing decisions are made in the design and implementation process of the compilation and code generation process, it is possible to derive tracing maps between model elements and code lines for each step of the computation process. From these tracing mappings, an overall tracing between the original model elements and the generated code parts is computed, from which a region mapping can be derived and used for the TPP placement.

6.3 Categorization of Timing Values

In accordance to the information the modeler needs and expects, the modeling tool has to be able to request the right kind of timing information, so that the way the result is calculated and the way it is understood match. Additionally to the common categories of WCET and BCET values, we introduce the following categorizations:

> ***Flat*** time value: The time value relates to this model element, excluding nested model elements, opposed to
>
> ***Deep*** time value: The time value includes the timing values for contained model elements.

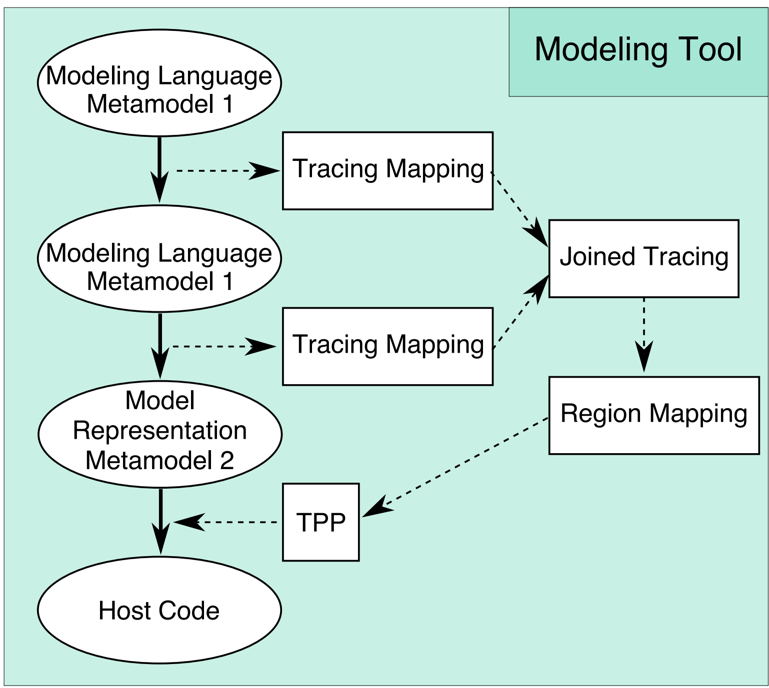


Figure 6.2. The general concept of tracing for interactive timing analysis

Fractional time value: The time value denotes the share of the overall WCET that is attributed to this model element, opposed to

Local time value: The time value is to be interpreted as the amount of time which can possibly be spent executing this model element, without restriciton to the overall WCET execution path.

All notions operate analogously for BCET values. The difference between flat and deep time values has already been explained in Chapter 2 with the help of the motivational example. Recall Figure 2.2. For the regions, in the timing labels in the right upper corner of the model view, there are two time values, separated by a slash. The first value is the flat time value and the

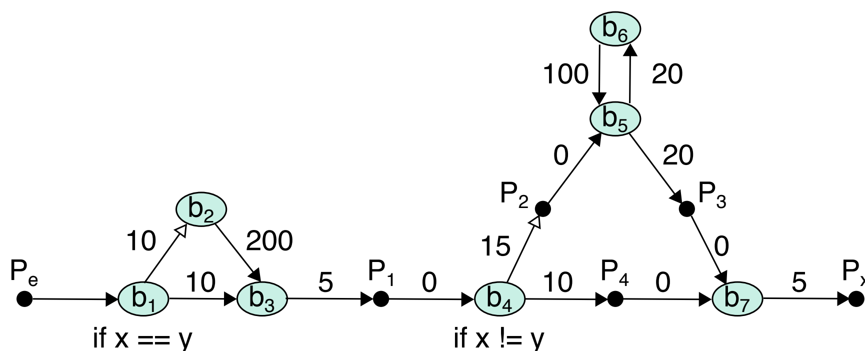


Figure 6.3. An ITAG that includes the basic blocks b_1 to b_7 and the TPPs P_e, P_x, and P_1 to P_4. The basic blocks b_1 and b_4 are left based on conditionals with the condition denoted below the basic blocks. Recall that arrows with white heads signify true branches, while arrows with black heads represent false branches as well as general control flow. The time values for basic blocks are denoted on their outgoing edges, outgoing edges of TPP have a time value of 0. We assume that analysis can determine that the loop is taken at most 10 times under all circumstances. The denoted time values are WCET time values in clock cycles. Figure adapted from [FBH+16].

second value is the deep time value. For region Main this means for example that in the first time value the times for the regions HandleEmergency and HandleMotor are not included, while the second value includes the time spent in the child regions on the critical path. Note that the time values shown in the figure are fractional time values.

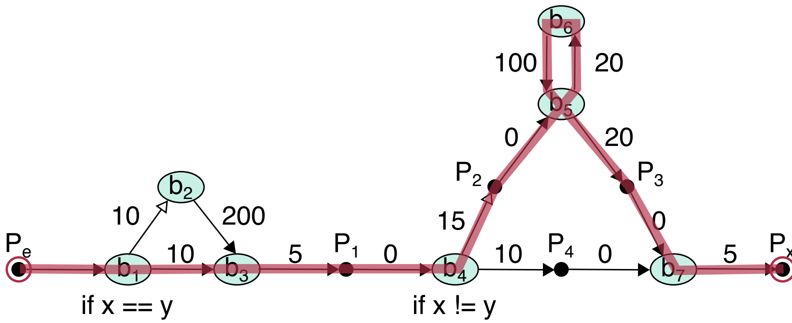
In the following, the difference between fractional and local time values is further elaborated and illustrated with the help of an example ITAG, shown in Figure 6.3. The figure shows this special sort of CFG for a function f. The function consists of seven basic blocks, b_1 to b_7. Also, six TPP are present: the two implicit ones, P_e and P_x, and P_1 to P_4. The function contains two conditional branches, one at the end of basic block b_1 and one at the end of basic block b_4. The condition of the first branching reads if x == y, the second condition is if x != y. Thus, the conditions are mutually exclusive. If we assume that the timing analysis tool is able to detect this, the analysis will determine a critical path that does not take both true branches. The

timing costs of the basic blocks are denoted on the outgoing edges. For the TPP, zero time values are given in likewise manner. The time values are given in clock cycles.

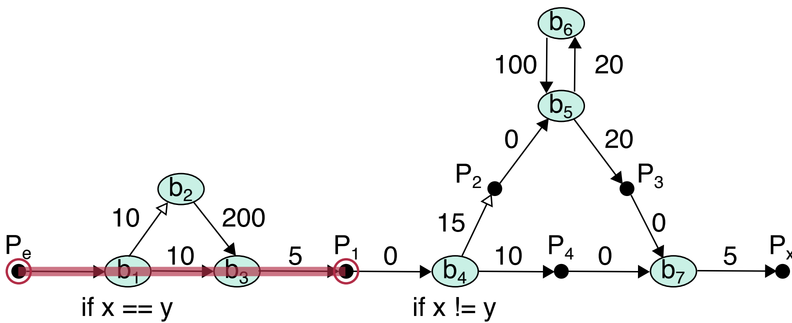
If we assume that the timing analysis ascertains that the loop in the function is taken at least 2 and at most 10 times, the overall WCET path, i.e. the critical path between P_e and P_x, takes the false branch in b_1 and the true branch in b_4 as illustrated in Figure 6.4a, yielding a time value of 1255 clock cycles. In this setting, we now ask for the fractional time value between the TPP P_e and P_1. The timing analysis is expected to return the time value in relation to the part of the overall WCET path. This part of the overall critical path is shown in Figure 6.4b. It involves taking the false branch at b_1 and corresponds to a time value of 15 clock cycles.

In contrast, the timing response for a local time value between P_e and P_1 yields the time value for the most costly path between P_e and P_1, without regard on whether it is part of the overall critical path or not. This is illustrated in Figure 6.4c. The local critical path between P_e and P_1 takes the true branch at the conditional at b_1 and thus equates to a time value of 215 clock cycles. Note that for P_e and P_x as TPP pair, local and fractional time value are identical. A formal definition of the timing response to requests for local and fractional time values is given in Section 6.10.

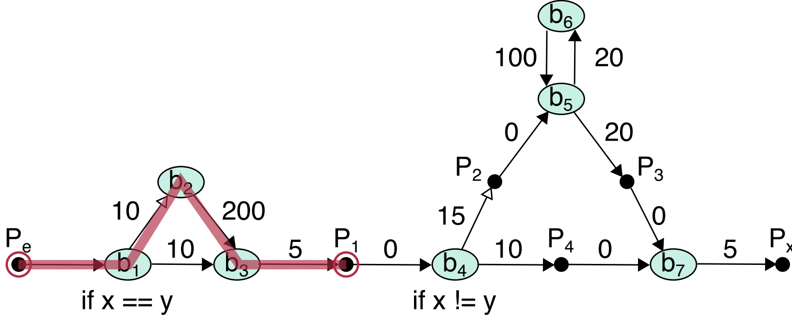
Note that the timing analysis tool is expected to prune out infeasible program paths it has detected even in the case the analysis request is for a local time value and the infeasible path information is derived from the analysis of code parts outside the borders set by the TPP. A simple example of this is shown in Figure 6.5. Assume that in this small ITAG, basic block b_1 performs the assignment x=20 and the function represented by the ITAG includes no other assignment to x. Then this constitutes that the true branch of basic block b_2 is an infeasible path, because the related condition if $x < 10$ cannot evaluate to true. If the timing analysis tool gets a request for the local time value between P_1 and P_x, the information constituting the infeasible path lies outside the code part marked by the TPP borders, as basic block b_1, where the assignment to x takes place, is located sequentially before TPP P_1. Nevertheless, even for a local time value request, the time value should be estimated under consideration of the infeasible path, as otherwise significant overestimations of the local timing potential might occur. Thus in



(a) The overall critical path of the ITAG in Figure 6.3.



(b) The path for the fractional time value between P_e and P_1.



(c) The path for the local time value between P_e and P_1.

Figure 6.4. Fractional and local WCET time value for the example of Figure 6.3.

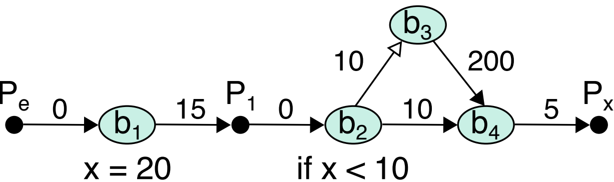


Figure 6.5. Infeasible path detection derived from a global analysis of the function is leveraged for local time value estimation. In this setting, the true branch of basic block b_2 is not feasible because of the assignment to x in basic block b_1 that sets x to 20. Having detected this, the tool returns a value of 15 clock cycles in response to a request for the local time value for TPPs (P_1,P_x). The given time values are the WCET values.

the example of Figure 6.5, the timing analysis tool should return a time value of 15 clock cycles, if it is able to handle infeasible paths in general. From this follows that even local time values cannot be calculated in isolation, meaning for the specified code part without regarding the context. This reinforces the need to employ fast methods for interactive analysis.

6.4 Special Aspects of TPP Placement and its Semantics

In this section, specific aspects of TPP placement and its effect on the meaning of timing analysis requests are discussed in an intuitive way. For a formal definition please refer to Section 6.10.

In general, a TPP is placed between any two statements in the code that belong to the representations of different specified model elements, which in case of our example implementation are regions. It follows that in the context of a corresponding ITAG for the generated function, TPP nodes will never be placed adjacent to each other. Another aspect is that sequential code parts that could be represented by one single basic block node might be split into two or more basic blocks, because the code consists of parts of different region representations. These smaller basic blocks are then separated by TPPs.

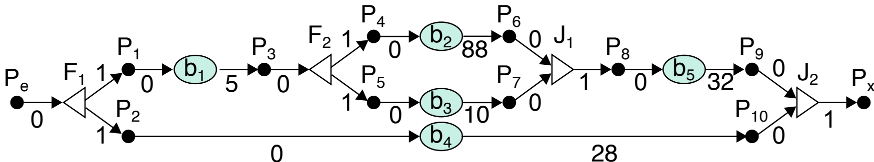


Figure 6.6. An ITAG with nested concurrency. The white triangles pointing left are fork nodes, the triangles pointing right are join nodes. The edge weights signify WCET time values in processor clock cycles.

As we assume implicit TPPs at the beginning and end of the generated function, any complete ITAG always shows the nodes $\mathsf{P_e}$ and $\mathsf{P_x}$ as source and sink of the graph, as can for example be seen in Figure 6.6. In interactive timing analysis there will typically be a timing analysis request for the code delimited by this pair of program points, as the result is the overall WCET or BCET, respectively. Furthermore, TPPs will often be placed around one basic block, like for example $\mathsf{P_1}$ and $\mathsf{P_3}$ in Figure 6.6, which frame basic block $\mathsf{b_1}$. A timing request related to this pair of program points results in a timing value for $\mathsf{b_1}$ and its outgoing transition.

Apart from these basic aspects there are two more interesting constellations concerning TPP placement and the associated semantics. The relation of TPP placement to parallel structures is considered in Section 6.4.1, followed by a discussion of TPP placement inside and in the vicinity of loops in Section 6.4.2.

6.4.1 TPP Placement and Parallel Structures

For the example implementation detailed in Chapter 7 the generated code is sequential, parallel structures are not part of the ITAGs corresponding to the generated tick functions. However, this is not a general requirement of the interactive timing analysis interface. This interface can also be implemented by modeling tools that generate code with parallel structures. In the ITAG, these are represented in an abstract way by fork and join nodes starting and ending parallel code parts. Note that in case of more than two parallel code threads, the concurrency is expressed by nested fork and join nodes in the ITAG. Nested fork and join structures also express hierarchy of threads.

The typical placement of TPPs in the context of concurrent programs depends on the code generation strategy of the modeling tool as well as the kind of model elements that are specified for detailed timing analysis requests. Consider now more closely Figure 6.6 that shows the drawing of an ITAG with nested fork and join structures. For the general notation of ITAGs, please see Chapter 5, especially Figure 5.1 on page 65. Figure 6.6 shows a graph structure with two nested fork/join vertex pairs, five basic blocks and twelve TPP nodes which illustrate the possible placements of TPPs in relation to concurrent structures.

If detailed timing values are to be retrieved for regions, corresponding fork and join nodes typically are attributed to the same region, while the code threads between the nodes are matched to different regions each. For example, the placement of TPP nodes P_4 to P_7 are characteristic for a region based interactive timing analysis. Whether TPPs are placed directly before a fork or after a join depends on whether these nodes belong to the same model element representation as the preceding or successive block, respectively. For example, P_3 will only be placed if the following fork statement does not belong to the same model element representation as basic block b_1. Likewise, P_8 will be placed if the join statement does not represent the same model element as basic block b_5. Additionally, the modeling tool might place these points if a timing value for the overall parallel structure is requested.

The timing analysis interface does not dictate a method how timing values for parallel constructs, for example the result for a request of the WCET value between the TPP P_3 and P_8, are to be retrieved. The choice of an approach happens on timing analysis tool side and depends on the way the generated code is further processed, compiled and how it is actually executed on the processor. For example, if concurrent threads in the program are sequentialized after the code is communicated to the analysis tool, but before execution, the times consumed by the execution of concurrent threads might be added, while in case of actual parallel execution, the maximum of parallel thread execution times is calculated. Furthermore, code parts might be mapped for actual parallel execution in a way that does not correspond to the concurrent threads in the model, which would also have to be respected by the timing analysis.

Note that in our example implementation introduced in Chapter 7 concurrent threads are sequentialized before the code is communicated to the analysis tool, so that no fork-join structures are contained in the code.

In the following, assume for example that concurrent threads are actually executed in parallel. Then the analysis tool would compare the two paths between between P_3 and P_8, to find the worst case execution time path between the two. One path is passing basic block b_2 and the other instead is involving basic block b_3. The maximum of the two including associated transitions would be considered the critical path, in this case that is the one involving basic block b_2 with the edge weight sum $0+1+88+0+1=90$.

For the WCET path between P_e and P_x, the calculation becomes more complex. Still assuming, the two threads will be executed in parallel, then the maximum of their execution times would be taken into account. One of the two threads is sequential and involves only the single path $\langle P_e, F_1, P_2, b_4, P_{10}, J_2, P_x\rangle$. Of the other thread, only the subpaths $\langle P_e, P_1, b_1, P_3, F_2\rangle$ and $\langle J_1, P_8, b_5, P_9, J_2, P_x\rangle$ are sequential and the related timing values including the basic blocks b_1 and b_5 might be added. However, the thread forks two child threads with F_2, and for them again the concrete implementation decides which time value is significant. Lets assume, this is again the maximum of the thread execution times. In this case, $\langle P_e, F_1, P_1, b_1, P_3, F_2, P_4, b_2, P_6, J_1, P_8, b_5, P_9, J_2, P_x\rangle$ would be determined as the critical path with a WCET value of $0+1+0+5+0+1+0+88+1+0+32+0+1=129$.

However, if in a TPP pair, one of the TPPs is located outside the fork-join node pair, like for example for the TPP node pair P_3 and P_6, a timing request for the path between the two points does not involve a maximum calculation for the threads related to this fork. The thread for which the timing value is requested is expressed by this choice of TPP placement, as there is only one path from P_3 to P_6, independent on the fork node that is part of this path. The critical path in this case is $\langle P_3, F_2, P_4, b_2, P_6\rangle$ with a timing value of $0+1+0+88=89$. Similarly the critical path between P_4 and P_8 is $\langle P_4, b_2, P_6, J_1, P_8\rangle$ yielding a time value of $0+88+0+1=89$, without the need to determine a maximal thread.

An interesting question is now, what time value we expect if we specify the path from P_3 to P_7 for our request assuming we are calculating with the respective maximum execution time in case of concurrent threads. We have

chosen a path, $\langle$P$_3$, F$_2$, P$_5$, b$_3$, P$_7\rangle$ on one of the threads of F$_1$, but it is not a subpath of the overall critical path. Thus, for a request of the fractional time value between the points, we expect a time value of zero, because this path is irrelevant for the overall WCET. Note however, that P$_1$ to P$_7$ yields a time value of 5, as the chosen path is partially on the critical path so that the time value on the outgoing edge of b$_1$ counts. For a request for the local time value the result of P$_3$ to P$_7$ is to be given as 0+1+0+10=11, because this represents the local sum of edge weights.

The placement of a TPP p can fix the thread or threads under analysis only for each pair of related fork and join nodes ($f \in F, j \in J$ with $j = join(f)$), in which it is placed, meaning for which there is a path $f \rightarrow_E^* p$ as well as a path $p \rightarrow_E^* j$. For example, for the TPP pair (P$_e$, P$_6$) there is only one fixed path, while the calculation of a timing value for the pair (P$_e$, P$_8$) would leave a maximum calculation to be determined, as there are two possible candidates for the critical path between the two TPP.

6.4.2 TPP Placement and Loops

Depending on the type of analysis request, the placement of TPPs and its semantics with regard to loops in the program can be sufficiently complex to make restrictions for TPP placement sensible. Intuitively, we ask the same question as for acyclic code, namely for the accumulated weight of the most (least for BCET) costly path between a specified pair of TPPs. However, paths in code with loop structures can contain several instances of the same node and a specific definition is needed that determines which pair of instances is significant. Furthermore, loops can have complex structures, can be nested and may contain conditionals, which also means that they do not have to be passed in the same way with each loop iteration. The same node may be executed with every single iteration, but this is not always the case. This makes it difficult to generally define the semantics of a request for the timing value between two program points, unless both of them are placed outside the loop structure. This is the reason for the restriction in [FBH+16], where we confine TPP placement by demanding that the derived TPPG for an ITAG has to be a DAG. However, this restriction is not mandatory. I will explain this in the following and suggest an interface definition expansion

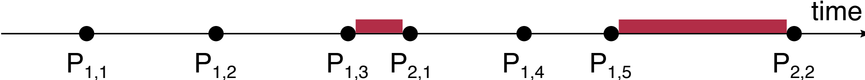


Figure 6.7. Abstract view on the timeline for the execution of a critical path with a loop structure. The fractional time value request is for the TPP pair (P_1, P_2). Only the instance pairs marked with the red rectangles will be considered as candidates for the requested value.

that handles general ITAG structures, dropping the restriction to simple ITAGs. The definition expansion is proposed informally in this section. It starts with details on TPP placement in loops and their effect on timing value calculation and then discusses TPPs in loops and path request responses. The extended formal definition follows in Section 6.10.2.

TPP Placement in Loops and Related Timing Values

The approach to the interface definition expansion is based on one general principle: The response to any timing analysis request for a pair of TPPs (P_1, P_2) should always be based on a path in the ITAG that does not contain P_1 in any other position but as the start node and also does not contain P_2 in any other position as the target node of the path. In case P_1 and P_2 are identical, this node is start and target node of the path, but the path should not visit it apart from that. This condition is illustrated in Figure 6.7. The illustration concerns a fractional time value request. This example is chosen without loss of generality, for the guiding principle can be recognized most straightforwardly in a time line view for an overall critical path in a function with a loop structure. Assume the analysis tool gets a fractional WCET time value request for the TPP P_1 and P_2, both of which lie within a loop structure in the function. We see a timeline for the critical path in the drawing, the occurrences of the two program points denoted in order and at the time of their appearance. There are five instances of P_1 and two of P_2. Though there are a number of possible instance pairs for the two points, the approach is only to consider pairs that are sequentially ordered in time and also consecutive without any other instances of any of the two points inbetween. In the example, this leaves only two candidates for the requested time value:

($P_{1,3}$, $P_{2,1}$) and ($P_{1,5}$, $P_{2,2}$). Of the two, the instance pair with the maximum time value is selected, which is ($P_{1,5}$, $P_{2,2}$).

The approach works similarly for local time values. However, they are not defined in relation to an overall critical path and thus the overall timeline abstraction is not feasible. Instead, the definition refers to the ITAG representation of the model and asks for the most costly path between the two TPP nodes that does not cross one of the two nodes twice.

The guiding notion behind this approach is that timing information that involves general information on the loop like iteration bounds should only matter if the timing analysis request asks to analyze the loop as a whole by placing both TPP outside the loop. As soon as at least one TPP is placed inside the loop, the analysis tool should only be concerned with the specified part of the code itself without regarding the number of iterations. Otherwise it would not be possible to get timing information on code encapsuled in loops in an isolated fashion. A related advantage of this approach is that timing values for requests of this kind often can be retrieved even when general bounds for the iteration of the loop cannot be determined by the analysis tool. Thus, timing information for parts of the loop can be collected, even if the loop itself cannot be proven to be bounded. This is not possible for the main alternative approach, which would be to always use the instances of the two TPP that are furthest apart in the execution for WCET calculation. Note that the advantage of determining time values for parts of the code of a function that might not be proven to terminate can be leveraged best with the notion of local time values, as they have no relation to an overall WCET path. However, even if a timing analysis tool is not able to determine an overall timing value, because the function contains a loop that cannot be proven to terminate, the analysis could still be able to identify a periodic critical path. Thus even for fractional time values, a partial analysis is thinkable with this approach, as only the analysis of one period is needed for a sufficient reference. But even for less sophisticated systems, the definitions yielded by this approach hold, are unambiguous and offer the possibility to differentiate between information containing loop bound information and detailed timing information on code parts within the loop.

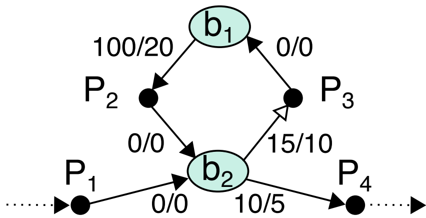


Figure 6.8. Part of an ITAG with a simple loop structure in the control flow. The image illustrates the basic TPP node placement possibilities in a simple loop setting.

Furthermore, this definition has the advantage that it facilitates tool chains with timing analysis tools that are not able to determine loop bounds either in general or without additional flow fact information. A modeling tool that has information on the loop bounds, for example by programmer annotations or by highlevel language analysis, can request the time value from start to exit(s) of a loop and then perform the time value aggregation according to the loop bound information on the modeling tool side. For this purpose, it is vital to place the first TPP directly and unconditionally at the beginning of the loop so that it is definitely passed with every iteration of the loop.

In addition to the discussed advantages, this approach is compatible with the definition in [FBH+16], as its condition clearly holds there due to the restriction to simple ITAGs in which no TPP might occur twice in a path, as loops may not contain TPPs.

Figure 6.8 shows a section of an ITAG with a simple loop. The four TPPs illustrate the different possible node placements in this setting. Assume we apply the approach proposed above and also assume an upper loop iteration bound of 10 and a lower loop iteration bound of 2. Let the loop be part of the most as well as least critical path of the function. Then the timing values for the TPP pairs are derived as shown in Table 6.1. To increase conciseness, TPP pairs in which the second TPP is not reachable from the first are omitted from the table and zero edge values are left out of the calculations. As we have assumed that the loop is on the critical as well as the least critical path of the function, the local time values are identical with the fractional values and are also omitted.

Table 6.1. Timing values for the example in Figure 6.8. The zero values of TPP node edges are omitted to trim the expressions. The local WCET and local BCET values are in this example identical with the local time values for these TPP combinations, as we assume that the loop is on the critical and least critical path.

TPP Pair	fractional WCET	fractional BCET
(P_1, P_2)	$15 + 100 = 115$	$10 + 20 = 30$
(P_1, P_3)	15	10
(P_1, P_4)	$10x15 + 10x100 + 10 = 1160$	$2x10 + 2x20 + 5 = 65$
(P_2, P_2)	$15 + 100 = 115$	$10 + 20 = 35$
(P_2, P_3)	15	10
(P_2, P_4)	10	5
(P_3, P_2)	100	20
(P_3, P_3)	$100 + 15 = 115$	$20 + 10 = 30$
(P_3, P_4)	$100 + 10 = 110$	$20 + 5 = 25$

The only TPP combination for which the loop bounds are important is the pair (P_1, P_4), due to the approach defined here. Note that the sum of the respective time values for pairs (P_1, P_3), (P_3, P_2), and (P_2, P_4) is not equal to the time value for (P_1, P_4), as the sum does not include the iterations.

The impact of this conceptual design decision for local time values is illustrated in Figure 6.9. When a local time value for the TPP pair (P_3, P_2) is requested, the calculation of the BCET value is straightforward, based on the path $\langle P_3, b_3, b_1, b_2, b_4, b_6, P_2 \rangle$ the timing weights amount to 15+10+10+5+20=60.

For the WCET value and the approach proposed here, we regard the path $\langle P_3, b_3, b_1, b_2, b_4, b_5, P_2 \rangle$ with a time value of 20+10+10+10+100=150. The timing analysis tool is, in contrast to requests for the pair (P_1, P_4), not expected to determine how often the execution might circulate the subpath $\langle P_3, b_3, b_1, b_2, P_3 \rangle$ before it reaches P_2.

However, a complex loop structure can contain program paths through a loop iteration that contain neither of the two TPPs named in the time value request. This situation is illustrated in Figure 6.10. Regarding a local time value request for the TPP pair (P_3, P_2) and assuming the timing analysis tool finds that it is a feasible path to reach P_2 from P_3 passing the loop branch B

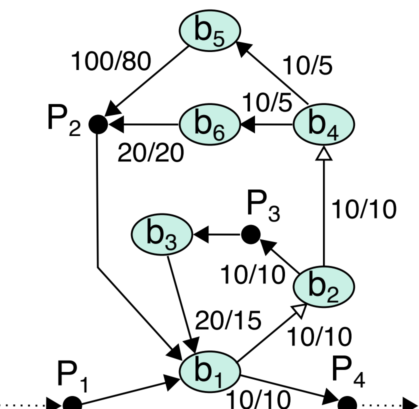


Figure 6.9. Part of an ITAG with a loop structure containing a conditional branch. The zero edge weights for outgoing edges of TPP nodes are left out for readability. The path relevant for the local WCET value for the TPP pair (P_3, P_2) passes P_3 only once—as the start node.

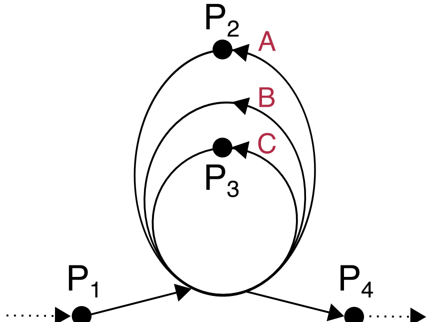


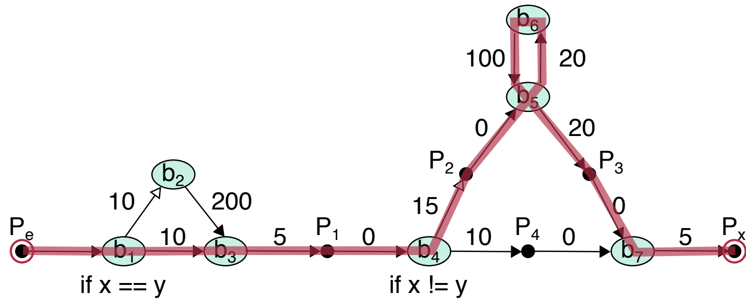
Figure 6.10. Aspect for the definition of a timing value for the TPP pair (P_3, P_2): The loop structure of the function can involve branches that contain neither of the two program points.

several times, the timing analysis tool has to bound the number of iterations of branch B to determine the time value with the approach proposed here. A modified approach might be considered in which it is requested that not only P_3 and P_2 but in general no node might be passed twice for the time value path in the graph. However, such an approach is not suitable, as it would not detect for example a path that connects P_3 to P_2 and passes branch B twice, even if such a path is feasible.

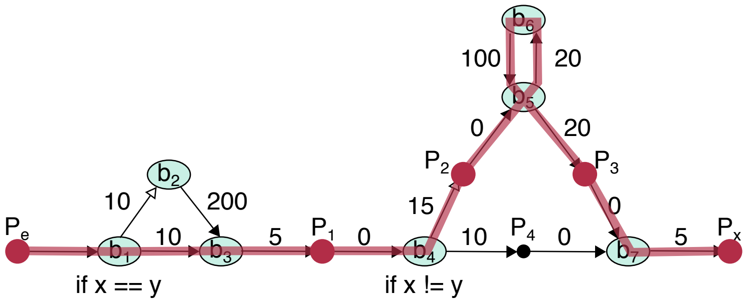
TPP Placement in Loops and Path requests

Finally, there is another aspect of TPP placement in loops, which concerns the response to a timing request for the critical or least critical path between two TPPs. As explained above in Section 6.2, the timing analysis tool communicates the path in form of a list of TPPs in passing order. As Figure 6.11 illustrates, without TPP in the program loop, the analysis tool response carries no information on whether the loop is taken or not. In the example, no TPP have been placed in the loop structure, so the timing analysis tool response for a request of the overall critical path will not reflect whether the loop has been taken or not. If however we place TPPs in the loop, we have to define what this means for the analysis tool answer to a worst-case or best-case path request. Assume for example we place a TPP P_5 between b_5 and b_6. Assuming again that the loop is taken at least 2 and at most 10 times, P_5 would be repeated two times in the list in the best-case path response and ten times in the worst-case path response to represent the (least) critical path closely. This has the advantage that the information conveyed to the timing analysis tool in this case includes some information on how often the loop is taken. A disadvantage is that the length of the response list would correspond to the number of loop iterations, which will be finite and bounded in case of a regular response, but can still in worst case lead to a considerable length of the list. However, as the extent of the list is grounded in iterative behaviour, the list contains periodic patterns which in an implementation can be handled by simple compression methods. Thus, a dynamic interaction can be maintained.

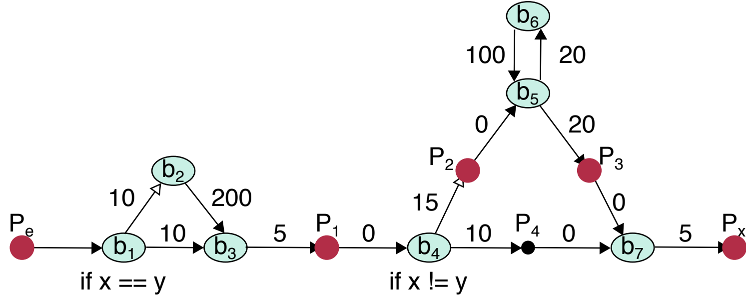
It is thinkable to use an alternative approach that enters repeated patterns only once to the list, but this would be of the same complexity as



(a) The overall critical path of the ITAG in Figure 6.3. The path contains a loop.



(b) TPPs on the overall critical path.



(c) The TPPs to be returned as a list.

Figure 6.11. The response to a request for a (least) critical path is answered with a list of TPPs. In this case, no TPPs have been placed in the loop, so the analysis tool answer does not contain information on whether the loop is entered.

simple compressions that write for example 13448x(P_1, P_3, P_1, P_2) for a path $\langle P_{1,1}, P_{2,1}, P_{3,1}, P_{1,2}, ..., P_{1,26895}, P_{3,13448}, P_{1,26896}, P_{2,13448}, \rangle$. Therefore, it is not necessary to drop information in order to make an implementation feasible.

6.5 Aggregation of Timing Values

On the side of the modeling tool, typically time values for regions of the model have to be aggregated from the time values returned by the timing analysis tool in the timing response. The main reason for this is that in the code generation process the code parts that represent one region are often scattered, so that more than one code part belongs to the representation of a region. This can affect flat as well as deep time values. Figure 6.12 shows an abstracted view of a model part with four, partly nested, regions A to D. On the right side of the illustration, generated sequential code is displayed represented by black lines, which are mapped to the regions as discernible from the labels. Code parts that belong to different regions are separated by TPPs, from TPP(1) to TPP(6). The code belonging to region B has been split in two parts in the scheduling and code generation process. Inbetween the two code parts lie not only the representations of B's two parallel child regions C and D, but also the code attributed to the region parallel with B itself, region A. With the help of the inserted TPPs, the modeling tool now has the task to determine the right set of single requests to send to the timing analysis tool and afterwards to compute the region time values by aggregating the retrieved response values. In the following, Section 6.5.1 introduces the straightforward concept of fulfilling this task for fractional time values. Afterwards, Section 6.5.2 explains why the aggregation of local time values is more complex. Section 6.5.3 elaborates on an aggregation approach for local values, which I have investigated, but abandoned, and finally suggest the concept for an interface expansion for tight local time value aggregation in Section 6.5.4.

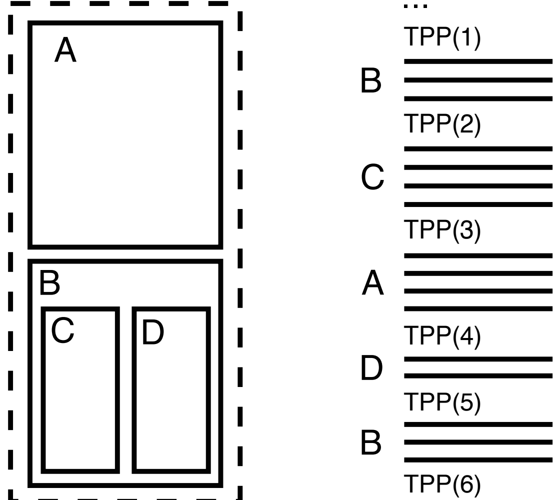


Figure 6.12. Schematic example of a model part with four regions, in which `A` and `B` are parallel regions on the outermost level and where `B` has two parallel child regions `C` and `D`. On the right side of the picture, the black lines sympbolize code parts, which are labelled with the region attributed by mapping. The code parts of different regions are separated by the TPPs `TPP(1)` to `TPP(6)`.

6.5.1 Aggregation of Fractional Time Values

For fractional time values, flat as well as deep time values, this process is fairly straightforward. As all time values are a fraction of the overall BCET or WCET and all relate to the same least or most critical path, the fractional time values for the five code parts that are separated by the TPPs add up to yield the overall timing value. It follows that it is sufficient for the modeling tool to ask exactly for the fractional time values for the following code parts:

1. Timevalue 1: For part 1 from TPP(1) to TPP(2)
2. Timevalue 2: For part 2 from TPP(2) to TPP(3)
3. Timevalue 3: For part 3 from TPP(3) to TPP(4)

4. Timevalue 4: For part 4 from TPP(4) to TPP(5)

5. Timevalue 5: For part 5 from TPP(5) to TPP(6)

Then the modeling tool can compute the desired time values for the regions as follows: The overall fractional time value for the code part shown is computed as tv1+tv2+tv3+tv4+tv5 with tv as a short notation for timevalue. The flat fractional timing value for region B has to be aggregated as a sum of tv1 and tv5. Its deep fractional timing value is calculated as tv1+tv2+tv4+tv5, which is the sum of all timing values for codeparts attributed to B or its child regions C and D. For region A, flat as well as deep fractional timing value are identical with t3, as its code part is not split and it contains no child regions. In a similar fashion, the flat and deep fractional time value of regions C and D correspond to single time values: tv2 for C and tv4 for D.

6.5.2 Aggregation of Local Time Values

The aggregation of local time values is more complex, because time values of different code parts have no uniform reference to a (least) critical path. This also means that code parts do not necessarily all reach their local (least) critical timing behaviour with regard to the same environment and state.

This problem is illustrated with an example in Figure 6.13. This shows the abstracted code example from Figure 6.12 with time values in relation to an exemplary boolean input I. For each code part there is a worst case time value under the assumption that this input takes a certain value in the tick. Part 1, which is mapped to region B, reaches a higher timing value when input I is true, while the other code section attributed to B, part 5, has worst-case behaviour with input I being false. Thus, if the modeling tool requests time values for each of the two code parts, it will receive a value of 500 for part 1 and a value of 100 for part 2. The modeling tool could sum the two up to get a flat local time value for region B, but the two partial values have been derived under the assumption of different environments. Thus, the resulting time value carries additional overestimation. However, the result is safe value, if both time values are safe local time values, as the analysis tool then considers the worst possible time for each code part. Nonetheless, the two code parts will never be executed with different environments, as I

	Local time values in relation to input I:	
...	I = true	I = false
TPP(1)		
B	500	200
TPP(2)		
C	100	10
TPP(3)		
A	10	400
TPP(4)		
D	10	100
TPP(5)		
B	10	100
TPP(6)		
...		

Figure 6.13. For the abstracted example of Figure 6.12, time values for the two values of boolean input I are shown on the right sight. The two code sections mapped to region B reach their worst-case timing behaviour for different values of I, as well as the first code section of B and the sections for its child regions.

is a pure input and will not be written throughout the tick. So the highest added time value that will occur is 510, which results for an environment, in which I is true. A modeling tool that simply adds up partial local worst case values would compute this value to be 600, which shows the overestimation. Similarly, adding up local time values for the four code parts attributed to B or its children C and D to compute the deep time value of region B yields a safe value of 800 instead of the tighter and safe value of 620 that results for the situation that input I is true.

This problem can be tolerable in general for a code generation with a good optimization with regard to context switches, where region code parts are not scattered heavily, under the condition that detailed local values are only used as a coarse indicator and interactive timing analysis is combined with a classic timing analysis at the end of the modeling process. However,

there can be model constellations and code generations that force the code parts of a region to be separated, like models with heavy reciprocal writer-reader communication between parallel threads, which force the threads to be scheduled in parts alternatingly. Also, the use of hierarchy typically leads to the distribution of the parent thread code. Thus, having multiple code parts for one region is not a phenomenon that can be avoided totally.

These problems on one hand favor fractional time values for interactive timing analysis, as their uncomplicated aggregation keeps the number of requests down and reduces the calculation requirements. This facilitates a fast round-trip-time for timing information feedback.

On the other hand, it is of interest to develop more sophisticated retrieval and aggregation methods for local time values that allow for tighter time values. The following two sections discuss this and finally present an expansion to the timing analysis interface for local time value aggregation.

6.5.3 Investigated Subtraction Approach

One approach that I have investigated but finally abandoned is to request time values for the whole code area that contains code parts for the region in question and later subtract on modeling tool side time values for code parts of other regions situated inbetween. In the example of Figure 6.13 this would mean to request a local time value for the code from TPP(1) to TPP(6) and then for example, to get the flat local time value of region B, subtract time values for code parts 2 to 4. However, it can lead to unsafe time values to subtract local time values for these code parts, as these local values might have been estimated for a different environment than the one that leads to a worst-case behaviour of the code part from TPP(1) to TPP(6). This means that the subtracted time values might exceed the amounts that were calculated in for these code parts for the overall value of the code part TPP(1) to TPP(6). Consider again Figure 6.13. When the modeling tool asks for the local time value from TPP(1) to TPP(6), the timing analysis tool will calculate a time value of 810 for the constellation, in which input I is false. If we now for example subtract the local time value of code part 2, which is attributed to region C, we subtract a value of 100 for an input environment with I = true. This is however not the value that was taken into account for

code part 2 in the overall value of 810, as part 2 only reaches a time value of 10 when I is false. By subtracting a value that is too high, we risk getting an unsafe time value. Note that subtracting the time value for the whole block from TPP(2) and TPP(5) at once works correctly in this case, but is no general solution. Instead, we would have to force the time values for the subtracted parts to be retrieved for the same environment than the value for TPP(1) to TPP(6).

In order to obtain such a guarantee, I have investigated to expand the definition of fractional time values to relate not to the overall worst case path of the model, but to an arbitrary path identified with a start and end program point. Such a modeling tool could request the local worst-case time value for the code from TPP(1) to TPP(6) and also request the fractional time value for the code part from TPP(2) to TPP(5) in relation to the critical path from TPP(1) to TPP(6).

However, even subtracting this time value does not always lead to the desired result and can even introduce unsafe values, which can also be seen in the example: For the overall local value from TPP(1) to TPP(6), we get a time value of 810 for the constellation, in which I is false. We subtract the fractional value between TPP(2) and TPP(5) with the same interface, which is 510. Then the result yielded for the flat local time value of B is 300, which is well shy of the correct value of 510, arising from an environment with I = true.

Thus, the approach with time value subtractions on modeling tool side had to be considered unsafe and was relinquished.

6.5.4 Concept Expansion for Local Value Aggregation

One basic recognition on the aggregation of local time values is that the environment and state in which the WCET behaviour of a code part occur might change, if this code part is expanded or reduced. Such approaches that work with additive or subtractive aggregation methods on the modeling tool side are bound to add overestimations or even lead to unsafe values as explained above. It follows that the modeling tool has to specify exactly the code part for which the timing value should be determined. Code parts that consist of split sections in the code have to be specified with the help of more than one pair of TPP.

This approach shifts the task of calculating the aggregated time values for split code parts to the timing analysis tool, while the tracing and book-keeping on which code parts belong together stays on the modeling tool side.

Concretely, the modeling tool names all sections that have to be taken into account in the analysis, for example in form of a list of TPP pairs. In terms of the ITAG, for the determination of the according time value, only edge weights on paths between the respective program point pairs will be taken into account, all others are taken to be of zero value. For this constellation, the timing analysis tool has to determine the (least) critical path to derive the overall local time value of the distributed code part.

For example, if the modeling tool wants to ask for the flat local WCET value of region B in the example of Figure 6.13, it would specify a request for the local WCET value of the code part given by the list of the following two TPP pairs: [(TPP(1),TPP(2)),(TPP(5),TPP(6))]. The analysis tool would be expected to determine the execution path that is most critical with respect to the determined two code regions together. In this, it would consider the time values for the code parts 2, 3, and 4 in the example to be of zero value in any case. Thus it would determine that the worst-case execution for this distributed code part happens in relation to an environment in which input I is true. The analysis then yields a time value of 510.

In addition to the list of TPP pairs, the interface allows to specify start and end TPPs for the whole section to enable partial code analysis where feasible.

6.6 Scale of TPP Combinations

For an assessment of the feasibility and scalability of interactive timing analysis in general or with respect to the usage of specific types of timing values, it is an important issue how many combinations of TPPs the timing analysis tool may have to investigate, especially, how the number of combinations under analysis grows with a higher number of TPPs.

For fractional time values, the modeling tool can perform all necessary calculations with a set of timing responses with fractional values for each

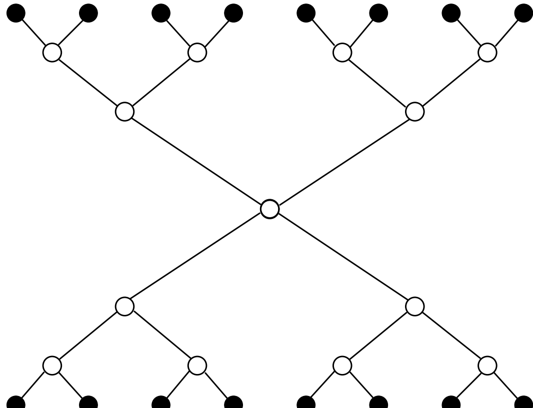


Figure 6.14. An abstraction of the worst case constellation for consecutive TPP constellations. Filled circles denote TPPs, non-filled circles signify join, fork or basic block nodes. The straight lines denote edges whose direction is from top to bottom.

pair of consecutive TPPs, as detailed in Section 6.5.1. This means that the analysis tool will only have TPP pairs to consider in which the second point is reachable from the first without crossing another TPP in the ITAG.

This means that first the number of possible consecutive TPP pairs is at least as big as the number of TPPs, minus one for t_x, which is never start point of a request pair. However, also the structure of the ITAG representing the code is of influence. According to Definition 5.14 on page 65, there are two types of nodes in the ITAG, which can have more than one incoming edge, basic block nodes and join nodes. A basic block node can have multiple incoming edges and thus can connect multiple pair-starting TPPs with the following code, the join node can connect two, in the case of nesting of branches or parallel structures accordingly more. Also, there are two types of nodes that have two outgoing edges and thus can connect a pair-starting TPP with possibly two consecutive TPPs per nesting level, the basic block nodes and the fork nodes. These considerations result in a worst-case combination of TPP placement and partial ITAG structure, which is illustrated for a number of 16 TPP in Figure 6.14. The illustration displays TPPs as filled black circles and denotes join, basic block and fork nodes

as circles with white filling. All edges are drawn as straight lines, each with the direction from top to bottom of the image, therefore arrow heads are omitted. The eight pair-starting TPPs at the top of the illustration are connected by a flipped tree structure of basic block or join nodes that merge branches or threads. Thus, all are connected to the root of a tree structure of fork or basic block nodes that offers reachablility of eight pair-closing TPPs. Note that the upper part of the structure could for example also consist of one basic block node with eight incoming edges from TPP node sources. In any case, 64 pairs of consecutive TPP exist in the worst case example. Though neither the TPP placement nor the code structure are likely to ever reach the worst-case constellation, this leads to a worst-case upper bound of a quadratic growth of TPP pairs for analysis in relation to the number of TPPs for fractional time values.

Typically, an actual implementation will reach a much better rate. In our example implementation introduced in Chapter 7, the number of request TPP pairs grows linearly with the number of TPPs.

As explained in detail in Section 6.5.2, to aggregate time values for consecutive TPPs pairs on the modeling tool side is not feasible for local time values.

For the following considerations, I assume that the interface expansion for the aggregation of local time values is implemented. Thus, a request for a local time value includes the specification of a TPP pair that names the overall code section in which the code parts of a region are situated. Additionally, it includes a list of TPP pairs that further specify these scattered code parts. In the following, I will consider the list pairs as well as the general request pair as TPP pairs under analysis.

Each TPP is considered to start a code part that is mapped to a specific region. For an example, refer again to Figure 6.12 on page 98. There, TPP(1) is attributed to region B, TPP(2) is mapped to region C and so on. For each region there will be two TPP pairs to specify the code section in which scattered code parts of this region are situated, one for the flat value, considering only code parts that are directly related, and one for the deep value, which takes also the code parts for child regions into account. Additionally we have the TPP pairs specified in the request lists. Assuming the modeling tool requests flat as well as deep timing values, the times one

TPP is mentioned as starting TPP in one of the lists depends on the number of hierarchy levels of its attributed region. Each TPP will be named once for the list in its flat value request, once for the list in its deep value request and once for the deep value request of each region which contains it.

6.7 Function Analysis

One of the technical challenges in interactive timing analysis is to enable a fast update of timing values for a model change. This challenge is addressed in the design of the interactive timing analysis interface by the key concept of the separation of concerns for the analysis of the generated tick() function on one hand and the analysis of external, called functions on the other.

The tick() function itself has to be analyzed with every request/response interaction between modeling tool and analysis tool, for example with every saved change of the model, as in our example implementation. However, external functions that are called by the tick() function are typically not part of the model changes, and often refer to a library. For this reason, the interface permits to compute WCET and BCET numbers for these functions offline by a traditional timing analysis tool and to use the values as assumptions in the interactive timing analysis. With these assumptions, only the tick() function values need to be recomputed in each feedback cycle, while the called function numbers are fetched from the assumption information. This enables a faster interaction performance of the timing analysis feedback. As the calling context of the external functions cannot be respected by an offline calculation, this option can lead to more pessimistic, though sound, approximations. Therefore it can be interesting to combine an interactive timing analysis leveraging this assumption option, which is used for fast feedback during the modeling process, with a classical analysis tool, which is called at the end of the modeling process to improve the tightness of the final timing value without the need for fast analysis performance. This is illustrated in Figure 6.15.

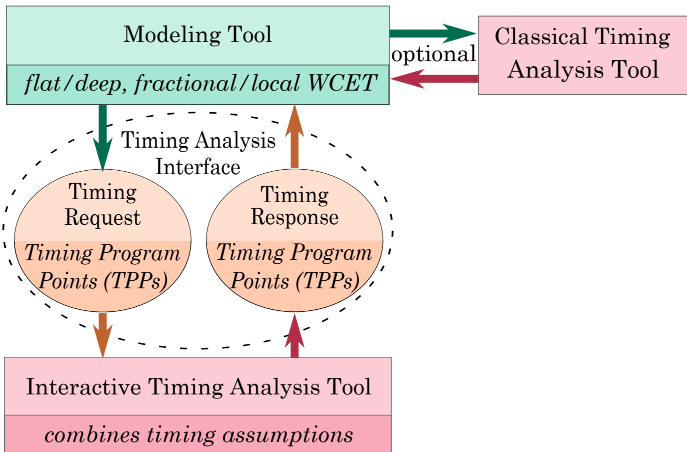


Figure 6.15. In some settings it can yield additional benefits to combine the interactive timing analysis with the usage of a classical timing analysis tool. Especially if timing assumptions for external function calls are used, the application of a traditional timing analysis tool for a single pass at the end of the modeling process can be leveraged to ensure tightness of the timing value.

6.8 Value Assumptions

Another form of additional information that can optionally be communicated by the modeling tool to help the interactive timing analyzer to retrieve tight timing estimations are value assumptions. These may concern function arguments as well as global variable values. For example, if a global variable is an integer, but on the modeling tool side there exists the information that the value of this global variable will never be another value than zero or one (for example, because the variable is used to represent a boolean input), then this information can be conveyed to the analysis tool along with the other types of assumptions, for example the timing bounds for external functions. This happens by associating the argument with a value, which in the formal definition is represented as an abstract value, for which we do not prescribe a domain. This depends on the type of the parameter or variable and is left to implementation. For example, for an integer value,

the abstract assumption value could be an integer interval. For a concrete implementation example please refer to Section 7.7.

With the information that only a stated set of values can be supplied as arguments to the function under analysis or that a global variable is restricted to a specific range of values, the timing analysis tool may rule out infeasible paths and thus offer tighter timing estimations. However, these assumptions are not prerequisite for interactive timing analysis. If the implementation does not make use of them, the timing values may just have to be more pessimistic estimations.

6.9 State Representation

Another source of overestimations for the analysis of the tick() function can arise for state based systems, if the timing analysis tool simply computes a time value for one single traversal of the tick() function in isolation. For an example we revisit the improved robot model from Chapter 2, which is shown as an SCChart in Figure 6.16. Here, the two inner regions HandleEmergency and HandleMotor can never be executed in the same tick, as the state Emergency, which contains the HandleEmergency region, is only entered when the boolean input bumper is true, while the surrounding state Normal of the region HandleMotor is entered when the bumper input is false. However, a key point in this constellation is that the polling for the value of bumper happens in one tick of the model execution and the execution of the region contents inside the target states takes place one tick later. This is the case, as both transitions from state Init to the states Emergency and Normal are immediate transitions, but the inner transitions of the regions HandleEmergency and HandleMotor are delayed transtions, which means that in contrast to immediate transitions they consume one tick. Thus, there is state information that has to be conveyed from one tick to the next. The decision made in the first tick based on the value of bumper must be preserved for the execution of the next tick. Such a state preservation can escape a timing analysis tool that just looks on one execution of the tick function without knowing which parts of the code carry state information.

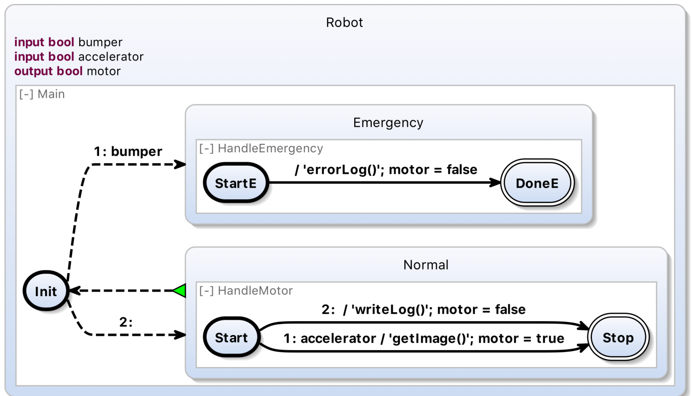


Figure 6.16. The motivational improved robot example involves mutually exclusive execution of the two inner regions, as HandleEmergency is only executed when the bumper input has been true and Handlemotor only when it has been false in the previous tick.

See for example again the code generated by our example implementation for the improved robot model in Listing 6.1 on page 76. The execution of code parts in this generated `tick()` function is controlled by boolean guard conditions. The value of the input bumper is assigned to a conditional guard variable `_cg1` in line 24. This variable is used directly respectively in negated form for the assignments of the guard variables `g2` in line 27 and `g6` in line 39. Only one of these guards can evaluate to true at any given execution of the tick function, as `_cg1` is not changed during the execution. Guard `g2` controls the execution of the HandleEmergency region, while the HandleMotor region execution is guarded by `g6`. However, these guards are not used directly, as the execution is to continue only in the next tick. Thus, the values of `g2` and `g6` are stored in PRE guards `PRE_g2` and `PRE_g6` in lines 42 and 44 for the next pass of the `tick()` function.

The value of `g2` from the previous tick is then read from the variable `PRE_g2` into the guard `g3` in line 28. Guard `g3` is then used to control the execution of the content of region HandleEmergency with the help of the conditional statement in line 29.

For g6 and therefore the control of the HandleMotor region, the procedure is similar, though more complex, as HandleMotor has two inner transitions, of which only one is taken based on the value of the input accelerator. The previous value of g6 is read from PRE_g6 into the guard variable g7 in line 9 to be used in combination with the conditional guard for the accelerator input, _cg7, which is prepared in line 10. The guards that combine these conditions are g9 and g8 in lines 11 and 16. These guards control the execution of the inner transitions of the HandleMotor region in the conditional statements. Line 12 represents the case that bumper was not true in the previous tick and accelerator is false in the current tick. The conditional statement in line 17 indicates the case that input bumper was false in the previous tick and input accelerator is true in the current tick.

A timing analysis tool that investigates the tick() function with respect to only one run might be able to determine that g9 and g8 will not both be true in the same tick due to their inverse dependence on the accelerator input in the same tick. However, it will not detect that each of those two guards is mutually exclusive with guard g3 in the sense that when g3 evaluates to true, neither g8 nor g9 will. The tool can only figure out that both g9 and g8 are based on PRE_g6 on the one hand and that g3 is based on PRE_g2 on the other, but it fails to derive that PRE_g6 and PRE_g2 can never be true in the same execution of the tick() function. This mutual exclusivity is inherited from the guards g6 and g2 that were stored in the corresponding PRE guards in the previous tick, an information that is lost when we only look at one pass of the tick() function. Thus the timing analysis tool fails to prune out a special kind of infeasible paths, which are state based and which we call *reactive infeasible paths* to distinguish this problem from the general problem of detecting infeasible paths [GES+06].

To defuse this source of overestimation for state based systems, our interface offers the possiblity to name the state representing variables, in this example the PRE guards, to the timing analysis tool in the form of state assumptions. This enables the timing analysis tool to perform a state based analysis to find reactive infeasible paths for a tighter timing estimation.

The state information can only be effective if also the initial setting of the state variables is conveyed to offer the starting point for analysis. In general this is done by naming the initial values together with the variables. In case

the practical implementation contains an initialization function for state variables, which for example is the case in the example implementation described in Chapter 7, the conveyance in the timing request file can be accomplished compactly by naming the initialization function.

6.10 Interactive Timing Interface

In this section, the chapter concludes with the formal definition of the interactive timing analysis interface based on the concepts introduced in the preceding sections. The interface specifies the communication between a high-level modeling tool and an interactive timing analysis tool. For an example implementation of this specification see Chapter 7.

Section 6.10.1 contains the basic interface definition adapted from the publications [FBS+14a; FBS+14b; FBH+16], which are presented in more detail in the introduction. In Section 6.10.2, I suggest an expansion of the interface definition to accommodate general ITAG structures, i.e. to allow for TPP placement in loops, as explained in Section 6.4.2. Another interface expansion, addressing the problem of timing value aggregation for local time values, as discussed in Section 6.5, is introduced in Section 6.10.3.

6.10.1 Basic Interface Formalization

We define the interactive timing analysis interface based on the notion of a program as a set of functions and global variables of primitive type. The term of function is not limited to a specific programming language. In our example implementation, which is regarded in detail in Chapter 7, the functions are C functions that are generated from model representations in SCCharts. However, the interface is not limited to C, but suggested as a general specification. We define interactive timing analysis as follows.

Definition 6.1 (Interactive Timing Analysis)**.**
Given a program consisting of a set of functions F, and a set of global variables G, interactive timing analysis is a communication between a modeling tool and a timing analysis tool, in which repeatedly, triggered by

modeling tool user actions, the timing analysis tool poses a timing analysis request t_{req} and the timing analysis tool answers with a timing response t_{res}.

The kind of modeling tool user action is not prescribed by this interface, it can for example consist of saving a modeling step or pushing a dedicated button for timing analysis. A central term in this is the timing analysis request, which we define next.

Definition 6.2 (Timing Analysis Request)**.** A *timing analysis request* is a 7-tuple $t_{req} = (f, a, g, S, e, P, R)$ with $f \in F$ being the function to be analyzed, a, g, S, and e are the assumptions communicated to the timing analysis tool, P is the set of timing program points in f, and R is the set of requested analyses.

Details for the formal definitions of the assumptions (a, g, S, e), the program points (P) and the analyses requests (R) are given below. For the informal discussion of the underlying concepts please refer to the links offered for each aspect respectively in the following.

Assumptions

Assumptions represent additional information conveyed automatically by the modeling tool to the timing analysis tool to support the tightness of the timing analysis. The general concept of assuptions is introduced in Section 6.1. A key characteristic of assumptions is their optionality. The interactive timing interface is not limited to systems in which the modeling tool can provide these assumptions. Also, it is not limited to state based systems, state variables just offer a means to implement the specification for this kind of system.

Assumption $a : \mathbb{N} \rightarrow A$ is a function that specifies assumptions for the *arguments* that may be applied to function f. That is, expression $a(n)$ returns, for argument $n \in \mathbb{N}$, an abstract value $v \in V_a$. This is explained in detail in Section 6.8, together with the related concept of assumptions for global variables: $g : G \rightarrow V_a$ $g : G \rightarrow V_a$ which specifies the assumption for a value $g(x)$ of a *global variable* $x \in G$. In this V_a means the set of abstract values whose representation depends on the type of the variable or parameter. The basic representations consist of sets and where applicable intervals

of possible values. However, the interface does not stipulate an obligatory domain. If specific types call for other abstract representations, these can be implemented accordingly. Also, abstract value domains can be defined in form of coupling constraint relations between arguments or states. It is thinkable to extend the interface in the future to replace the latter usage by an additional section of assumptions expressed as constraint equations on variables and arguments. To investigate this, including the aspect of ideal representation to achieve adaptability for existing analysis tools, is left to future work.

S is the set of state variables with their initial setting, as discussed in Section 6.9.

Finally, the assumptions on the execution time of called external functions is represented by the function $e : F \rightarrow \mathbb{N}_\perp \times \mathbb{N}_\perp$, where $\mathbb{N}_\perp$ represents the domain for execution time values, whith $\perp$ as a notation for a missing safe upper or lower bound, whether due to non-termination or because a safe timing value has otherwise not been determined. We write $e(f_c) = (t_b, t_w)$ for $f_c \in F$ and with t_b and t_w denoting the safe lower and upper bound value, respectively. Note that this definition does not stipulate that both values have to be conveyed in form of a pair together in every implementation of this interface. They can also be communicated separately, and it is also possible to use only one of the two types of function timing assumptions, for example, if BCET information is not needed. For example, in the implementation introduced in Chapter 7, timing assumptions for functions are noted separately for WCET and BCET values.

Analyses Requests

With the timing analysis request t_{req}, the modeling tool specifies the information it wishes to receive from the timing analysis tool. As the modeling tool typically needs a number of detailed timing values, a timing analysis request consists of a set R of *single requests* $r \in R$, each being a triple y, p_a, p_b, where $y \in Y$ is the type of the desired timing value and p_a and p_b are TPPs that tag the start respectively end of the code part that the modeling tool wishes to be analyzed. The two markers may be any pair of TPPs specified in the finite set P of the timing request tuple t_{req}, including the implicit TPPs

p_e and p_x at the entry and exit of the function. Detailed information on the concept of TPPs is given in Section 6.2.

The interface differenciates six request types, which are:

$$Y = \{\texttt{WCP}, \texttt{BCP}, \texttt{LWCET}, \texttt{LBCET}, \texttt{FWCET}, \texttt{FBCET}\} . \tag{6.1}$$

With the help of the types WCP and BCP, the timing analysis tool can specify a single request for the worst-case path or best-case path of the code part given by the TPP markers, respectively. The concept of this type of request is explained in detail above at the end of Section 6.2.1.

The types LWCET and LBCET denote requests for local time values, for the worst-case and best-case respectively. Accordingly, FWCET is the request type for a fractional WCET timing value and FBCET denotes a request for the fractional BCET value.

The meaning of these different timing value categories and their associated requests is explained above in Section 6.3 and defined by a formal specification of what is expected as a response in the following.

Timing Response

For the set of analysis requests R, the timing response t_{res} is a function

$$t_{res} : R \to \mathbb{N}_\perp \cup \mathcal{P}() \tag{6.2}$$

where for each analysis request $r = (y_r, p_{a,r}, p_{b,r})$ we receive a value whose domain depends on the analysis request type of r. If $y_r \in \{\texttt{LWCET}, \texttt{LBCET}, \texttt{FWCET}, \texttt{FBCET}\}$ the result is a time value $\in \mathbb{N}_\perp$, with $\perp$ denoting that the code part could not be proven to terminate or a time value could not be retrieved for another reason. If, however, $r \in \{\texttt{WCP}, \texttt{BCP}\}$, the result is a finite path $\bar{p} = \langle p_1, p_2, \ldots, p_n \rangle \in \mathcal{P}()$, with $\mathcal{P}()$ denoting the set of all possible finite paths. If the term is used with relation to an ITAG G, it signifies the set of all possible finite paths in G.

In the following, let G be a simple, weighted ITAG as defined in Chapter 5, Definition 5.11 and Definition 5.14 on page 64 and 65.

Though we do not stipulate how the timing analysis is to be implemented, we make two assumptions for the following specifications: The first is that the timing analysis tool is in general able to derive basic block

and transition costs through analysis, where applicable under consideration of timing assumptions. Thus, we can base the following definition on the abstract representation of the weighted ITAG with its cost functions representing these basic time values. The second assumption is that the timing analysis tool is in general able to determine the best-case and worst-case execution flow between two TPPs in the program under analysis, to be represented as the (least) critical path in the ITAG. We denote the finding the worst-case execution time path as a function $\bar{v}^{w}_{p_1,p_2} = \langle p_1, v_1, v_2, \ldots, v_n, p_2 \rangle$ and name the corresponding function for the best-case execution time path $\bar{v}^{b}_{p_1,p_2} = \langle p_1, v_1, v_2, \ldots, v_n, p_2 \rangle$ for paths between two TPPs p_1 and p_2. Both functions return the empty path, if p_2 is not reachable from p_1, and $\perp$, if the analysis tool cannot determine a finite path between the two program points. The paths returned by these functions contain nodes of all categories. For the response to a `WCP` or `BCP` request, the corresponding timing program point path, as defined in Definition 5.16 on page 66, is calculated.
Finally now, we can define the response time function t_{res} for the different types of single requests, which also relates the precise formal meaning of the different timing value types:

Definition 6.3 (Response function t_{res})**.**

$$t_{res}(r) = \begin{cases} tpath(spath_{p_1,p_2}(\bar{v}^{w}_{p_e,p_x})) & \text{if } r = (\texttt{WCP}, p_1, p_2) \\ tpath(spath_{p_1,p_2}(\bar{v}^{b}_{p_e,p_x})) & \text{if } r = (\texttt{BCP}, p_1, p_2) \\ etime_w(\bar{v}^{w}_{p_1,p_2}) & \text{if } r = (\texttt{LWCET}, p_1, p_2) \\ etime_b(\bar{v}^{b}_{p_1,p_2}) & \text{if } r = (\texttt{LBCET}, p_1, p_2) \\ etime_w(spath_{p_1,p_2}(\bar{v}^{w}_{p_e,p_x})) & \text{if } r = (\texttt{FWCET}, p_1, p_2) \\ etime_b(spath_{p_1,p_2}(\bar{v}^{b}_{p_e,p_x})) & \text{if } r = (\texttt{FBCET}, p_1, p_2) \end{cases}$$

6.10.2 Interface Expansion for General ITAGs

In this section, I suggest an expansion of the basic interactive timing analysis interface that makes the interface applicable to general ITAGs instead of being restricted to simple ITAGs. This means that the expansion allows for TPP placement in loop structures. The concept behind this expansion is elaborated in detail in Section 6.4.2.

At the core of the suggested expansion is a strengthened requirement for the functions $\bar{v}^{w}_{p_1,p_2}$ and $\bar{v}^{b}_{p_1,p_2}$, which represents the approach that the critical path between two program points might not include these program points, but as start and end point of the critical path.

Definition 6.4 ((Least) Critical Path Function For General ITAGs)**.** Let G be a general ITAG, and let $\mathcal{P}_{\overline{p_1,p_2}}$ denote the set of all finite paths in G that start with an instance of p_1 and end with an instance of p_2. Then:

$$\bar{v}^{w}_{p_1,p_2} = \bar{v} \in \bar{U} : etime_w(\bar{v}) = \max_{\bar{u} \in \bar{U}}\{etime_w(\bar{u})\}$$
$$\text{with } \bar{U} = \{\bar{u} \in \mathcal{P}_{\overline{p_1,p_2}} :$$
$$\bar{u} = \langle v_1, ..., v_n \rangle : \forall\, v_i,\ 1 < i < n : v_1 \neq v_i \wedge v_n \neq v_i\}$$

and accordingly with identical $\bar{U}$ for the best-case path:

$$\bar{v}^{b}_{p_1,p_2} = \bar{v} \in \bar{U} : etime_b(\bar{v}) = \min_{\bar{u} \in \bar{U}}\{etime_b(\bar{u})\}$$

These new definitions already handle the local request types. However, they do not yet cover the changes necessary for the subpath based request types `WCP`, `BCP`, `FWCET`, and `FBCET`, for the determination of the subpath between two TPP might still be ambiguous if several instances of the TPPs in the request are existent in the reference path defined by $\bar{v}^{w}_{p_1,p_2}$ and $\bar{v}^{b}_{p_1,p_2}$. Thus, also $spath_{p_1,p_2}$, see base definition Definition 6.5, needs additional conditions to be clarified. Also, as the choice between different candidates for the subpath is made based on which candidate results in the longest (shortest) execution time, the definition has to be differenciated for worst-case and best-case timing:

Definition 6.5 (Subpath by TPP ($spath^{w}_{p_1,p_2}(\bar{v})$ and $spath^{b}_{p_1,p_2}(\bar{v})$)) for General ITAGs)**.** Let $\bar{v}$ be a path in an ITAG with possibly multiple instances of the TPPs p_1 and p_2. Let S be the set of all subpaths $\bar{s} = \langle p_1, v_n, ..., v_{n+m}, p_2 \rangle$ of $\bar{v}$ with $\forall\, v_i \in v_n, ..., v_{n+m} :\ v_i \neq p_1 \wedge v_i \neq p_2$ and let $Cand$ be the set of all pairs of instances of p_1 and p_2 that are start and end node respectively of a path in S. Then

$$spath^{w}_{p_1,p_2}(\bar{v}) = \bar{v}' \in S :\ etime_w(\bar{v}') = etime_w(max\{ \underset{(p_{1,i},\ p_{2,j}) \in Cand}{\bar{v}^{w}_{p_{1,i},p_{2,j}}} \})$$

and

$$spath^{b}_{p_1,p_2}(\bar{v}) = \bar{v}' \in S : etime_b(\bar{v}') = etime_b(min\{ \underset{(p_{1,i},\ p_{2,j}) \in Cand}{\bar{v}^{b}_{p_{1,i},p_{2,j}}} \})$$

With this, the definition of t_{res} itself does not change, it is defined as in Definition 6.8.

6.10.3 Interface Expansion for Local Value Aggregation

In this section, I define the interface expansion proposed conceptually in Section 6.5.4 for the aggregation of local time values.

First, I redefine the set R of analysis requests so that each analysis request $r \in R$ has the form $r = (y_r, p_{a,r}, p_{b,r}, L)$ with the additional L that signifies a list of pairs of program points in G.

Then I define execution time functions for a given path $etime_{lw}$ and $etime_{lb}$ to adapt the definitions in Definition 5.12. The new definitions consider time values of edges for a path only in the case that they are on a path between any pair of TPP in a given TPP pair list:

Definition 6.6 (Partial Execution Time of a Path)**.** Let $\bar{v} = \langle v_1, v_2, \ldots, v_n \rangle$ be a path in a weighted ITAG G. Furthermore let E_s be the set of edges $e = (u, w)$ with $u, w \in \bar{v}$ and $u = v_i$ and $w = v_{i+1}$, so that there is a subpath of $\bar{v}$, $\bar{v}' = \langle v_l, ..., u, w, ..., v_m \rangle$ with $(v_l, v_m) \in L, 1 \leqslant l \leqslant m \leqslant n$.

Then $etime_{lw}(\bar{v}, L) = \sum_{e \in E_s} w(e)$ and $etime_{lb}(\bar{v}, L) = \sum_{e \in E_s} b(e)$. For an empty path $\bar{v}$, $etime_{lw}(\bar{v}, L) = 0 = etime_{lb}(\bar{v}), L$ holds and if $\bar{v} = \bot$ then $etime_{lw}(\bar{v}, L) = \bot\ = etime_{lb}(\bar{v}, L)$.

Recall that the treatment of parallel structures is included in the definition of the functions for finding the (least) critical path, see also Section 6.4.1.

These functions are now also adapted to respect the information communicated in L as $\bar{v}^{lw}_{p_1,p_2,L}$ and $\bar{v}^{lb}_{p_1,p_2,L}$:

Definition 6.7 ((Least) Critical Path Function for Enhanced Local Time Value Aggregation)**.** Let G be a general ITAG, L denote a non empty list of TPP pairs, and let $\mathcal{P}_{\overline{p_1,p_2}}$ denote the set of all finite paths in G that start with

an instance of p_1 and end with an instance of p_2, then:

$$\bar{v}^{lw}_{p_1,p_2,L} = \bar{v} \in \mathcal{P}_{\overline{p_1,p_2}} : \max\{etime_{lw}(\bar{v}, L)\}$$

and accordingly for the best-case path:

$$\bar{v}^{lb}_{p_1,p_2,L} = \bar{v} \in \mathcal{P}_{\overline{p_1,p_2}} : \min\{etime_{lb}(\bar{v}, L)\}$$

With $\{\}$ as notation for the empty list then the definition of t_{res} is changed for local time value requests as follows:

Definition 6.8 (Response function t_{res} for Enhanced Local Time Value Aggregation)**.**

$$t_{res}(r) = \begin{cases} tpath(spath_{p_1,p_2}(\bar{v}^{w}_{p_e,p_x})) & \text{if } r = (\texttt{WCP}, p_1, p_2, \{\}) \\ tpath(spath_{p_1,p_2}(\bar{v}^{b}_{p_e,p_x})) & \text{if } r = (\texttt{BCP}, p_1, p_2, \{\}) \\ \boxed{etime_{lw}(\bar{v}^{lw}_{p_1,p_2,L}, L)} & \text{if } r = (\texttt{LWCET}, p_1, p_2, L) \\ \boxed{etime_{lb}(\bar{v}^{lb}_{p_1,p_2,L}, L)} & \text{if } r = (\texttt{LBCET}, p_1, p_2, L) \\ etime_{w}(spath_{p_1,p_2}(\bar{v}^{w}_{p_e,p_x})) & \text{if } r = (\texttt{FWCET}, p_1, p_2, \{\}) \\ etime_{b}(spath_{p_1,p_2}(\bar{v}^{b}_{p_e,p_x})) & \text{if } r = (\texttt{FBCET}, p_1, p_2, \{\}) \end{cases}$$

Note that this interface expansion can be combined with the expansion suggested in Section 6.10.2 by replacing $\mathcal{P}_{\overline{p_1,p_2}}$ with $\bar{U}$ in Definition 6.7, with $\bar{U}$ defined as in Definition 6.4 for `LWCET` and `LBCET` requests and using the expansion for general ITAGs as in Section 6.10.2 without changes for the remaining request types.

$$\begin{gathered} t_{req} = (f, a, g, S, e, P, R) \\ f \in F \\ a : \mathbb{N} \to V_a \\ g : G \to V_a \\ S \\ e : F \to \mathbb{N}_\perp \times \mathbb{N}_\perp \\ P \\ r \in R : (y, p_a, p_b),\ y \in Y, p_a, p_b \in P \\ Y = \{\texttt{WCP}, \texttt{BCP}, \texttt{LWCET}, \texttt{LBCET}, \texttt{FWCET}, \texttt{FBCET}\} \end{gathered}$$

$$V_a : \text{abstract value domain}$$

Chapter 7

Time for SCCharts

We have validated the interactive timing analysis interface with an example implementation in the KIEL Integrated Environment for Layout Eclipse Rich-Client (KIELER) project[1], which is a scientific open source project dedicated to the investigation of modeling tool technologies, modeling languages and their semantics, and the pragmatics of Model Driven Engineering (MDE). The project is introduced in detail in the following Section 7.1, including a description of the SCCharts modeling environment, which is the parent component of the interactive timing analysis implementation. The section closes with an introduction to the used technologies. Section 7.2 then introduces the GUI for user actions and time value feedback. This is followed by an explanation of the TPP placement with the help of the KIELER tracing framework in Section 7.3. Interrelations of interactive timing analysis and code generation are discussed in Section 7.4. Section 7.5 then contains a discussion of the potential influence the preserving TPP information can have on compiler optimizations. This is followed by an explanation of the identification of state variables for the KIELER SCCharts tool in Section 7.6. Next, Section 7.7 introduces the *timing request file*, which is used to convey analysis requests and assumptions, and its automatic generation. The chapter closes with an elaboration on the modeling pragmatics aspects implemented in the KIELER interactive timing analysis in Section 7.8.

[1] http://rtsys.informatik.uni-kiel.de/kieler

7.1 At home in KIELER—Technical Background

The following introduces the KIELER project with its goals and the current components in Section 7.1.1. The environment for modeling in the SCCharts language is detailed together with its used technologies in Section 7.1.2. This modeling tool is the host component of the interactive timing analysis implementation. Technologies used particularly for the interactive timing analysis are introduced in Section 7.1.3. This includes the experimental timing analysis tool, which is not part of the contributions of this thesis, but is connected to the implemented interface for the validation and evaluation of the concepts of interactive timing analysis.

7.1.1 Aim and Scope of the KIELER project

Where complex and often safety critical systems are developed with the help of graphical or textual modeling languages, the modeler is interested in the efficiency of his or her own work with the respective tools and also needs tool chains that facilitate the design of safe systems that guarantee deterministic results as well as timeliness. According to this, the research work of the KIELER project comprises two main investigation areas. First, the project is concerned with improving the *pragmatics* of graphical modeling work [FH09c; Fuh11]. A key feature of this component is the graphical layout, which is a continuous research topic of the project since the introduction of the first own layout algorithm implementation of the project, which was developed by Spönemann [Spö09] and adapted for dataflow graphs with ports by Schulze et al. [SSH14; Sch11b]. *KIELER Layout Algorithms (KLay) Layered* is a layer-based, hierarchical layout algorithm orignially based on the approach introduced by Sugiyama et al [STT81]. Automatic layout is not only used to enhance the readability of diagrams, but also to enable a number of approaches for the improvement of model design and maintenance. It unburdens the modeler from editing graphical model representations by hand, and allows for lightweight graphical display for textual notations, as detailed by Schneider et al. [SSH13] with the possibility to coordinate different *views* of the model, directed by *meta layout* and view management, introduced by Fuhrmann et al. [FH09c]. As stated by the

authors, this facilitates the integration of graphical information for analysis and debugging, but also of different strategies for the browsing of large models, like *focus-and-context*, where currently viewed or important model parts are shown in more detail, or the visualization of *Model to Model (M2M) transformations*. Originally, the KIELER Infrastructure for Meta Layout (KIML) and KLay were part of the KIELER project. Meanwhile, their functionality was moved into the official Eclipse project Eclipse Layout Kernel (ELK)[2].

The investigation of modeling language semantics constitutes the second research area of the KIELER project. A special focus lies on synchronous languages, which are dedicated to the design of safety critical systems, as elaborated in Section 1.1.2. In the last years the project has expanded from the investigation of existing modeling languages to additional research concerning the design of the new modeling language SCCharts [HMA+13b; HDM+14] that follows the SC MoC, which is explained in detail with related references in Chapter 4. For this language, the KIELER project offers the modeling environment *KIELER SCCharts* [HDM+14; MSH14; Mot17], which is described in the next section. Other research fields have been an editor and simulation environment for Esterel and a framework for the intermediate language *S* for compiling SynchCharts to Synchronous C or Synchronous Java, as well as integrations of KIELER with the Ptolemy project [Mot09; FH09b; SFH09; Han10; Mot10; TAH11; Mot11; RSS+13; MH14].

The current KIELER SCCharts modeling tool was suitable for an example implementation, as SCCharts are a graphical modeling language, which allows the implementation of graphical timing feedback. Furthermore, it can model state based systems, which was a prerequisite for testing the state based timing interface part. Also, there exists a growing collection of test models, which not only helped in the evaluation described in Chapter 8, but could also in the future be employed as a benchmark suite for the comparison of different timing analysis tools connected to the timing analysis interface. Also, the tight integration of automatic layout into the KIELER tool was a strong aspect, as it facilitates the incorporation of timing information display. The SCChart modeling support framework is decribed more closely in the following Section 7.1.2.

[2] `http://www.eclipse.org/elk/`

7.1.2 SCCharts in KIELER

The current SCCharts modeling environment is a successor of older projects, which were concerned with the development of an Eclipse-based editor for the language SyncCharts [Sch09], textual and structure based modeling of SyncCharts or general Statecharts [Wis06; Bay09; Sch11a], and simulation and code generation from SyncCharts [TAH11].

Currently, in the KIELER project SCCharts are modeled textually, using textual SCCharts language (sct). The detailed textual syntax is not relevant for this theses, an in-depth description can be found in [Mot17]. Though it is possible to use a command line compiler or an online compiler for sct files, the main SCCharts modeling component is part of the KIELER tool, which is integrated into the rich client platform Eclipse[3] and thus implemented in form of plugins, the basic Eclipse components. The SCCharts modeling environment consists of a number of Eclipse editors and views, which are currently organized into a choice of two perspectives. The SCCharts Modeling perspective consists of the textual sct editor, which is based on Xtext[4]. Xtext is a framework for the development of programming languages and Domain Specific Languages (DSLs), for which grammars can be defined in a dedicated grammar language. Xtext offers editing, parsing and code generation support as well as typechecking and the possibility to define validation rules for the language. The editor is combined with a diagram view, in which a graphical representation of the textual model is automatically generated with the help of the KIELER Lightweight Diagrams (KLighD) project [SSH12; SSH13]. KLighD is a framework for the synthesis of transient lightweight representations of models, whether textual or graphical. In this context, a graphical representation was chosen as an alternative to the integration of a complex graphical editor. The modeling perspective with activated timing analysis is shown in Figure 7.1. The textual editor for the sct file is shown on the left, while the automatically generated KLighD diagram view is situated to its right and has a number of *Diagram Options* that can be set by the user to customize the view. The interactive timing analysis user interface is part of these options. The corresponding

[3] https://eclipse.org

[4] http://www.eclipse.org/Xtext/

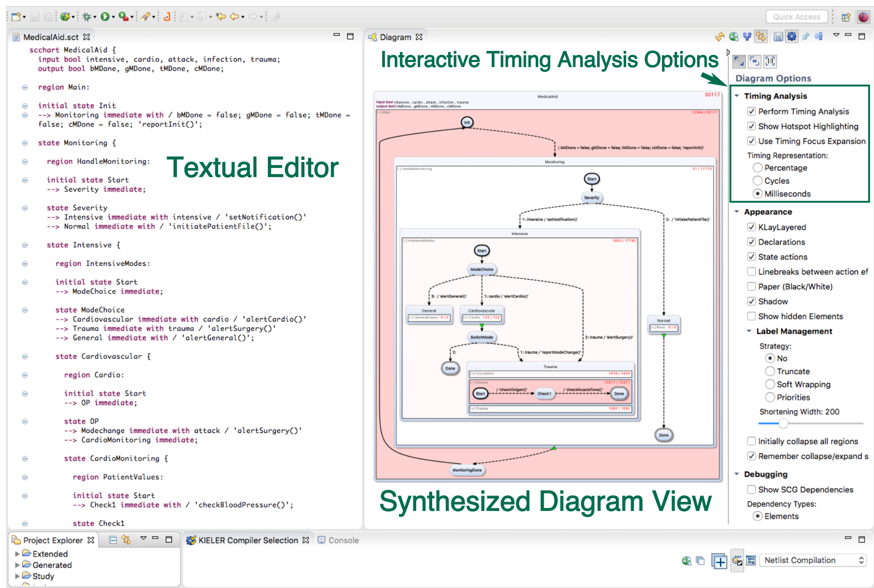


Figure 7.1. The SCCharts modeling perspective of the KIELER tool with the textual editor, the generated lightweight graphical view and activated timing analysis. The interactive timing analysis options are explained in Section 7.2.

section is marked in the screenshot. The interface is explained in more detail in Section 7.2. Apart from these parts that are the most prominent in connection with interactive timing analysis, the modeling perspective by default includes the project explorer, in which the user can manage the model files, the console, and the KIELER Compiler Selection, which serves as a user interface for the browsing of different states of the compilation process.

The second perspective is for SCCharts simulation. Its basic component in addition to the editor and diagram view is the KIELER Execution Manager (KIEM), which is an execution framework for the stepwise simulation of models with highlighting of active elements in the diagram view. The KIEM simulation comes with its basic configuration view, a data table, where

the user can actively set input signals and a signals view with a graphical display of present and absent signals. A detailed description can be found in [Mot17]. As the simulation and interactive timing analysis apply to different usecases, the interactive timing analysis is used in the default modeling perspective.

The compilation concepts for KIELER SCCharts have been introduced by von Hanxleden et al. [HDM+14]. There are two main approaches, the first is a dataflow based method which is suitable to compile SCCharts to either software, especially C code, or Hardware, as also elaborated by Rybicki et al. [RSM+16]. During compilation, Extended SCCharts are transformed to a normalized core language version, which is translated into its SCG representation. The SCG is devided into basic blocks, whose execution is controlled by boolean guards. This ensures control flow in a netlist, which is characterized by the continuous activity of all components.

The concept of SC, which is introduced in detail in Chapter 4, demands that sequential assignments to the same variable are allowed. In a netlist based compilation approach this meets the problem that only a single wire is assigned to the variable. The KIELER compilation approach solves this with a Single Static Assignment (SSA) [App98] related approach, were different instances of the variable are created for the different assignments. These instances correspond to wires in the hardware. A multiplexer selects the correct version for the final variable value in the sense of the Φ nodes of SSA. Both for the hardware sythesis and for compilation to C code, the ordering and selection of the variable copies must be consistent with the sequential ordering given by the program and also respect the iur-protocol for concurrent variable accesses. This is ensured by a control and dependency analysis, which leads to program rejection, if a cycle of control and dependency edges is detected. In the following, the netlist is either translated to hardware, or the SCG is sequentialized for software generation with the help of iur-scheduling of the basic blocks in accordance to the dependency analysis. The resulting sequential SCG is then translated into C code.

The second compilation approach is a priority based method, which is intended for software generation. It resembles the Syncronous C approach introduced by von Hanxleden [Han09], but incorporates the newer concepts

of the SC MoC. The updated priority based aproach has been proposed together with the netlist-based compilation approach [HDM+14], but the netlist-based compilation was implemented as the first main approach of KIELER SCCharts and has undergone more practical evaluation, while the priority-based approach is currently under investigation. One of its main advantages is that it can handle immediate control flow cycles. This approach uses *prioIDs* in SCL code to schedule concurrent threads. The concept assigns priorities to SCG nodes, which determine a scheduling order. Thread segments, which are defined by fork-join node pairs, are assigned ordered *thread segment IDs*. The prioIDs consist of a combination of the priority of a node and the ID of its thread segment. The overall ordering over prioIDs is such that different priorities decide the precedence and the thread segment IDs decide cases of identical priorities. Priority changes whithin a thread are allowed. The scheduler always chooses the thread with the highest prioID. Thus the algorithm that assigns priorities and thread segment IDs must respect the constraints given by the iur-protocol for the induced scheduling order.

For the example implementation of the interactive timing analysis we chose the netlist based compilation approach, which is well established in the KIELER tool, facilitates element tracing with the help of the SLIC compilation concept [MSH14] and had an already implemented general tracing approach [Sch14]. The tracing is explained further in Section 7.3.

7.1.3 Used Technologies

Besides the introduced general background in the KIELER SCCharts modeling environment, two main technology decisions characterize the example implementation of the interactive timing analysis. The first is the usage of KLighD technology to integrate the timing value feedback, hotspot highlighting and focus-and-context expansion into the Diagram view synthesis. This enables us to offer direct feedback within the visual model representation without the need for another view or window that would distract the developer from the central modeling process or simply crowd the screen.

The second aspect concerns the analysis tool side. The interactive timing interface has been developed together with David Broman from the KTH

Royal Institute of Stockholm, who engaged in the analysis tool perspective and in ongoing research develops a tool named KTA that satisfies the interface. In the course of our collaboration, he has implemented a simple experimental version of the tool to enable the evaluation of the interface [FBH+16]. This experimental tool is available under the MIT license[5]. This experimental version of KTA was the timing analysis tool connected to KIELER for all evaluation tests described below as well as for the user study. The timing analysis is done for the MIPS32 ISA. Its analysis approach is exhaustive-search and simulation-based. The obvious downside of this is that the tool is not scalable for large models with a high number of inputs, as all input combinations are tested. Thus, the tested models are restricted to models with boolean input and no more than 20 input variables. On the other hand, the retrieved timing values can be a useful comparison set for the assessment of the finished timing analysis tool version and, not regarding compiler optimizations, present a tight timing estimation for the interactive timing analysis validation, as the simulation is cycle-accurate [FBH+16]. A possible introduction of overestimation due to the implemented approach to TPP semantics is explained in Section 7.5.

The experimental timing analysis tool is designed to accept the C code, generated with inserted timing program points, and a timing request file from the modeling tool and performs analysis with the aim to retrieve timing information for code parts between TPPs instead of functions. As described in [FBH+16], the analysis employs a GCC-based cross-compiler for a MIPS32 architecture to compile the generated C code to ELF-binary. The problem of how to preserve the TPP location information in the binary is solved by inserting assembly labels for the timing program points, which is more closely discussed in Section 7.5. The timing analysis tool uses an internal data structure representation synthesized from the binary file as a basis for its exhaustive simulation approach. During the simulation, the tool searches for the WCET paths and memorizes the TPP that are on the path.

An important feature for the validation of the interface is that the tool is able to perform a state based analysis. It uses the initialization function of the generated code to retrieve the state variable initializations. Then it uses

[5] https://github.com/timed-c/kta

a fixed-point approach to compute and explore all possible combinations of input and state. In the first step all input combinations are explored to determine the set of possible states, which then is used for the next step of function exploration, generating the next set of possible states for the following step, until every constellation is explored. The analysis tool is able to retrieve local and fractional WCET values, fractional BCET values and the worst case execution time path.

7.2 User Interface

Though the example implementation allows also the programmatical polling of local WCET and fractional BCET values, for the user interface we concentrate on fractional WCET values, as the prominent usecase we have in mind is the timing related model revision in case of excessive WCET. Consequently, the proposed interface expansion for the aggregation of local time values is not implemented. Tight local time values are consequently only available for flat time values. Deep time values are only available with the help of a simple, addition based aggregation, which potentially introduces considerable overestimation. See Section 6.5 for a detailed explanation on this subject. Also, the fractional timing values are most suitable for timing hotspot highlighting, as they reflect the timing contributions to the actual WCET behaviour of the model. Thus, in its default configurations, our system requests fractional time values for each pair of consecutive TPPs. Also, it aggregates the deep timing values as described in Section 6.5.1. Additionally the modeling tool asks for the critical path of the model with a WCP request to prepare hotspot highlighting.

In the default preference setting of the KIELER modeling tool, the interactive timing analysis user options are deactivated. The reason is that the analysis only works with a connected timing analysis tool or in a testing mode with hand-provided response files, so that only systematic and prepared usage is feasible. Thus, the interactive timing analysis should be activated in the preference page of the SCCharts modeling environment, found in the KIELER preferences under KIELER SCCharts → Interactive Timing Analysis. This opens the preference page shown in Figure 7.2. As the tim-

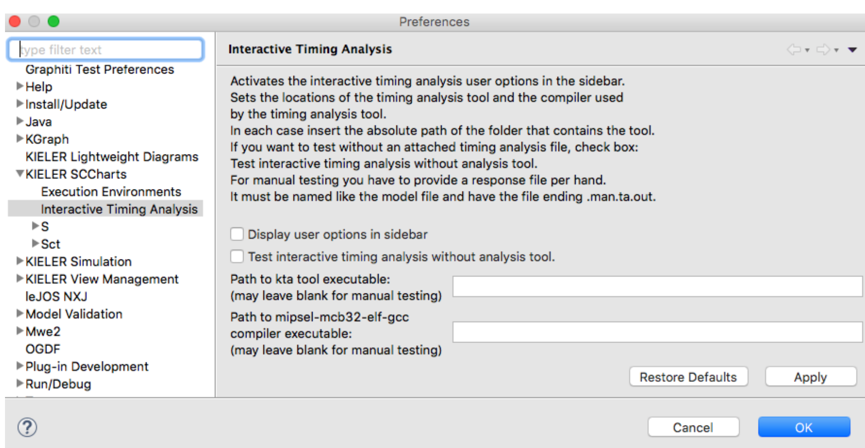


Figure 7.2. Screenshot: Preferences for the interactive timing analysis.

ing information display is integrated as a part of the synthesized diagram view, the user options are situated in the diagram options side bar. The first checkbox of the preferences page with the label Display user options in sidebar will trigger the display of the user options with the next opening of the diagram view. The next checkbox allows the developer to switch to an exceptional testing mode that does not require the timing analysis tool to be connected. This mode is intended for a developer who wants to test the timing value display without testing the timing analysis as such. It can be used for example in development work for the modeling tool itself. In that case, a manually written response file has to be provided in the same folder as the model file. The model sct file has the file ending .sct, for example a model file could be named model01.sct. The timing response files automatically generated by the timing analysis tool would be named model01.ta.out. A developer who wants to test the interactive timing analysis without the KTA tool for this example has to provide a file named model01.man.ta.out, which will be automatically polled instead of analysis tool invocation, if the checkbox labelled Test interactive timing analysis without analysis tool is activated. Note that the file has to correspond to the response

file format and contain exactly one answer for each timing request that is posed in the timing request file generated by the modeling tool, which would be named model01.ta in the example. Thus, the modeler must make sure the response file matches the current .ta file, either with the help of breakpoints or by using more than one analysis run.

If the timing analysis is used in the normal mode, the path to the KTA tool executable as well as the path to the mipsel-mcb32-elf-gcc compiler executable have to be inserted into the corresponding text fields below the check boxes. These paths have to name the folder that contains the executable, not the file itself. For example, if the compiler executable with its full path is /Applications/mcb32tools.app/Contents/Resources/Toolchain/bin/mipsel-mcb32-elf-gcc, then the path should be inserted as /Applications/mcb32tools.app/Contents/Resources/Toolchain/bin/.

With the activated interactive timing analysis, the timing diagram options are displayed in the diagram option bar as marked in the modeling perspective screenshot Figure 7.1 in Section 7.1.2 on page 123 and shown in detail in Figure 7.3. The options are displayed under the headline Diagram Options in the category Timing Analysis, which is opened in Figure 7.3. The first of the three checkboxes that has the textual description Perform Timing Analysis activates the interactive timing analysis and the display of the overall WCET value in the right upper corner of the root state as well as the display of detailed fractional flat and deep timing values for all regions. The first analysis is triggered with the activation of the checkbox, and as long as it is checked, the analysis will be repeated with every saving of model changes and can also be triggered by pressing the Refresh Diagram button pointed out with a red arrow in Figure 7.3. The region timing values are displayed in the right upper corners of the regions with the legend Flat Fractional WCET / Deep Fractional WCET. To display detailed timing values for regions is an exemplary choice. The concept would also have allowed to trace the timing for other model elements or to give the choice of the model element to the user. To offer timing values for regions was chosen because it is very suitable to be combined with hotspot highlighting and especially with a focus and context view that is based on collapsing and expanding model parts, which is natural to do in region units, as they represent element groups a modeler might want to focus at, especially a

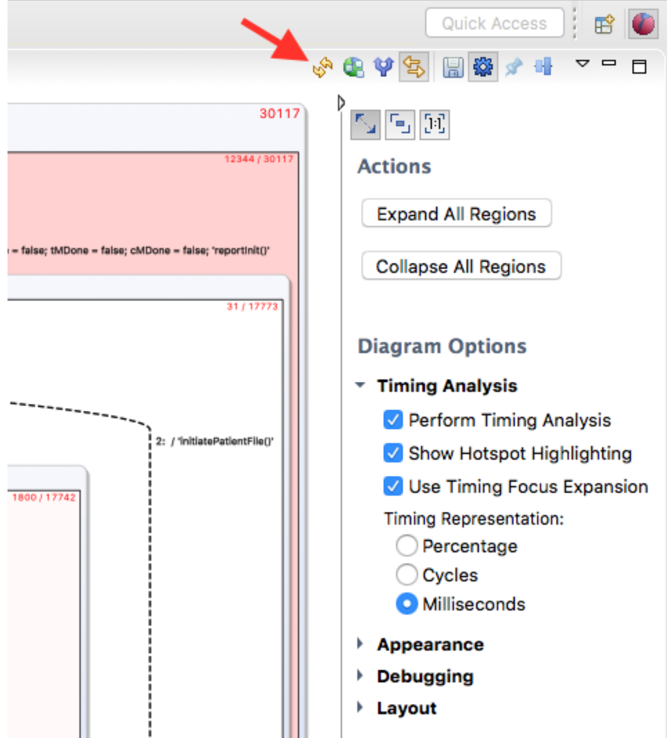


Figure 7.3. Screenshot: The user interface for the interactive timing analysis.

thread in concurrency or inner behaviour of a state.

The second checkbox Show Hotspot Highlighting activates the hotspot highlighting together with the timing value display. The hotspot highlighting is accomplished by coloring the background of every region on the critical path with a red color shade corresponding to its level of timing contribution, so that when a region is colored a deeper shade of red than another, it contributes more to the overall critical timing amount than the other region. Regions that contribute more than 50 percent of the overall timing additionally are shown with a thick red borderline rim. The highlighting is

updated in the same way as the detailed timing value display.

Finally, the third checkbox additionally activates a Focus and Context diagram display which consists of collapsing the regions that contribute less then ten percent of the overall worst-case timing value in their deep timing value. This functionality is more closely explained in Section 7.8.1.

In addition to these options it is possible to configure the representation of the timing values to be shown in the units of processor cycles, milliseconds or percent of the overall timing value. These configurations are not delegated to the timing analysis tool, it always responds in processor cycles. The necessary conversion happens on the modeling tool side, which at least for the percent display is mandatory, as the timing value aggregation for fractional values is done by the modeling tool. However, the usage of other units in the tool communication could be adjusted unproblematically with a simple programmatical configuration step.

7.3 TPP Placement with Tracing

To be able to implement the interactive timing analysis interface, the KIELER SCCharts modeling tool needed to be enabled to trace the model elements from the SCChart representation down through all compilation steps so that the corresponding parts of the C code could be marked with TPP. A strong advantage of the KIELER SCCharts tool regarding this task is its SLIC compilation approach as proposed by Motica et al. [MSH14], in which the compilation consists of a linear sequence of model transformations, of which each is performed only once. This renders the tracing of model elements more straightforward. Also, there exists a general tracing implementation for model elements through the compilation steps, which was implemented by Schulz-Rosengarten [Sch14]. This general tracing framework, called KIELER Transformation Tracing (KiTT) (formerly KIELER transformation mapping (KTM)), and how it was leveraged for the interactive timing analysis is explained in Section 7.3.1. Specific tracing problems with regard to interactive timing analysis are elaborated in Section 7.3.2.

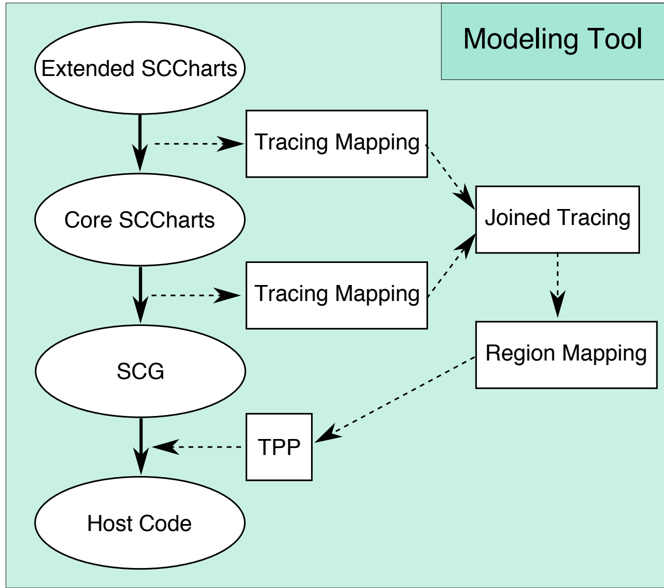


Figure 7.4. Tracing in the KIELER SCCharts tool. The model transformation from Extended SCCharts to Core SCCharts actuall is performed in a number of transformation steps. Before the translation to the SCG representation, the Core SCCharts are normalized. The retrieved SCG is sequentialized before code generation.

7.3.1 Employing the KIELER SCCharts Tracing Framework

The databased compilation approach for SCCharts in KIELER has the integrated tracing approach KiTT that was designed as a basis for different kinds of features like for example simulation highlighting, debugging visualization and feedback of analysis information [Sch14]. For each model transformation in the compilation process, the tracing framework produces a tracing mapping to store the information of which model elements in the new model represent which elements in the old model. These can be one-to-one relationships, one-to-many relationships and also many-to-one relationships. The tracing of transformation steps is illustrated in Figure 7.4 in an abstracted way. The transformation of Extended SCCharts to normalized

Core SCCharts, the transformation to the SCG and its sequentialization are in fact implemented in up to 27 transformation steps dedicated to the handling of different language features, and each transformation is documented with a tracing mapping.

As described in [FBH+16], an overall tracing that links elements of the original SCChart to the elements of the sequentialyzed SCG is computed as a transitive closure, denoted as Joined Tracing in Figure 7.4. This forms the basis for the extraction of a region mapping that tells us for every node in the sequentialized SCG, which region its represented element originally belonged to. As the sequentialized SCG is directly transformed into C code, this also indirectly links the code statement to the regions. In this example implementation, we use this region mapping to insert a TPP in form of an SCG node into the sequential SCG between any pair of adjacent nodes of which the source node of the connecting edge belongs to another region than the target node. This TPP node insertion is implemented in form of a model transformation that gets invoked after the sequentialization of the SCG. The inserted TPP nodes are assignment nodes with a text expression of the form "TPP(<number>)" which are directly passed along to the generated code.

See also Figure 7.5 for an illustration of the overall mapping relations between an SCChart, a simpler version of the robot example, and its overall tracing relation to the corresponding sequentialized SCG representation. The figure shows a screenshot generated with the help of a feature of the general tracing framework, which allows to visualize the tracing relations between model transformation stages. The mappings are visualized for the three transitions in the two inner regions of the SCChart. In the representation of the SCG it can be seen that the transition elements belonging to the region HandleEmergency are all mapped to the upper nodes until the assignment node labeled motor = false, after which the representation of region HandleMotor starts. Thus, the interactive timing analysis automatically inserts the TPP(2) node between these areas. The next TPP node, TPP(3) is inserted after a large section of SCG nodes attributed to region HandleMotor, when a context switch to the representation of another region happens.

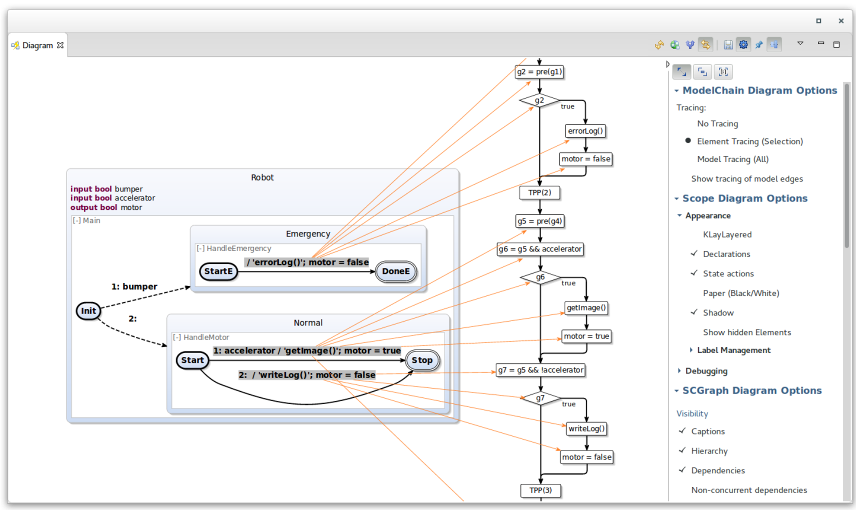


Figure 7.5. Screenshot of the view generated by the tool for visualizing the tracing mapping relation between a simple version of the Robot SCChart and its corresponding SCG. The element relations are automatically visualized with arrows. Source: [FBH+16].

7.3.2 Specific Tracing Aspects

To set up and implement the rules for model element tracing is always a task that demands careful consideration of the semantical language background. In the compilation of SCCharts, often highlevel, but rather simple, syntactical structures are replaced by elaborate compositions of lower level elements with a number of additional regions, nodes and variables. A detailed overview of transformations for Extended SCCharts is given by Motika [Mot17]. Also, especially for the creation of guards in the SCG, new elements are created that have no direct origin element in the source model. Thus, the tracing has to be implemented for each transformation with clear background knowledge of the respective semantics. Using the tracing framework of KIELER SCCharts for the interactive timing analysis did not only mean a practical test of the semantical correctness of existing mapping decisions.

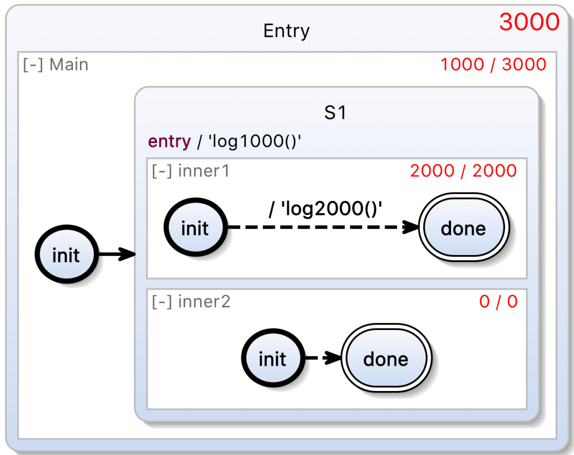


Figure 7.6. The test model Entry shows that the execution of state actions is not attributed to any inner region of the state, but to the parent region of the state.

As the detailed hotspot highlighting was implemented on region level, also the general element relations had to be revisited for a region allocation for all target model elements to original SCChart regions. For example for the entry action host code call in the model Entry, shown in Figure 7.6, it is not only important to trace which node or nodes of the SCG correspond to this call, but also to which region it should be attributed. An entry action is attributed to a state, in this example to state S1. As the example model shows, a state can contain more than one parallel region. There is no meaningful way to choose one of these regions over the other to attribute the timing value of the entry action host code call to. The alternative is to attribute the timing value in the region the node S1 belongs to, its parent region Main. This is the choice made in the KIELER tool. Its only disadvantage is that it might seem peculiar that the entry action is active together with the immediate behaviour of the inner regions of S1, but is attributed to their parent region. Nevertheless, it is the only clear assignment option that can generally be applied without regarding the existence and number of inner regions of the action state.

Another aspect is the assignment of guards. Recall the robot example shown in Figure 6.16 on page 109. The conditional that depends on the input bumper controls which of the inner regions of the model is executed, region HandleEmergency or region HandleMotor. In the netlist based approach that is reflected as boolean guards attributed to basic blocks which contain code representing the inner regions. This seems to imply an attribution to the inner regions, which is aggravated by the fact that the SCG has no expressive hierarchy as the SCChart, so that it is less easy to find the representation of enclosing structures. The tracing nevertheless must make sure that the time value for the evaluation of the condition is attributed to the main region. This does not seem crucial in this simple example, but it is possible that a condition is employed whose evaluation is costly. Such a condition could for example involve the call to a function whose return value is interpreted as boolean.

7.4 Interactive Timing Analysis and Code Generation

An interesting aspect of the implementation of the interactive timing analysis interface is the influence of the code generation approach on the timing behaviour of the model. An example of a potentially surprising timing value result is shown in Figure 7.7. We can see that in the critical instant of this model, the region getSamples is active and contributes the main part of the overall WCET. The execution of the two inner regions is mutually exclusive, as they are on different branches of a conditional depending on input dec. Thus, we would expect a fractional timing value of zero for region calibrate to be displayed, which is not the case. Looking at the timing assumptions for the function calls, we find that the timing value denoted for this region corresponds to the call to the function isLightApplication, which is not called as a transition effect but in the role of an indicator function whose return value is interpreted as boolean. The reason for the observed behaviour is that current code generation in the netlist based approach places all conditional evaluations in guard assignments on the same level in the C code and executes them without nesting of conditions. Thus, the execution of

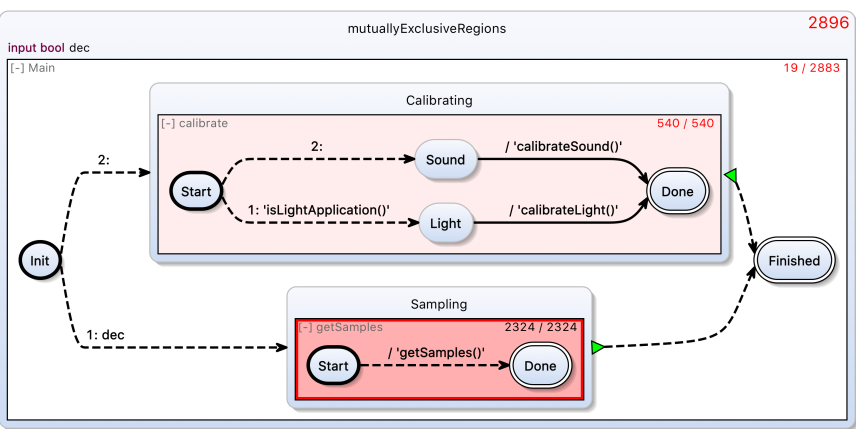


Figure 7.7. This example model shows a surprising timing result in which a region that is not expected to be active in this critical tick nevertheless shows a significant timing contribution. This effect is due to the flat conditional structure of the netlist based code generation approach that evaluates all conditionals without regard to nesting relations of conditionals in the model.

the conditional depending on isLightApplication is executed without regard to the result of the conditional depending on dec. This behaviour normally has no significant effect, as the evaluation of conditionals typically takes only a few processor cycles. In the case of heavyweight triggers as in this example, this characteristic of the code generation becomes clearly visible in the interactive timing analysis. Note, however, that this is no mistake on part of the timing analysis, its functionality is correct, as the region calibrate indeed contributes to the overall WCET path. The timing result mirrors a characteristic of the netlist based approach. A different code generation with nested conditionals would lead to a different result. Whether the implementation of such an alternative approach is in general rewarding depends on the tradeoff between an increased complexity of code generation and the optimization regarding unnecessary, but on average not costly condition evaluations.

7.5 TPP and Compiler Optimization

The interactive timing analysis tool accepts C code with inserted TPPs, but has to perform the timing analysis with regard to machine code. This means that the information on relevant code parts given by the TPPs has to be preserved in the compilation step. The experimental version of the KTA tool solves this problem by representing timing program points with the help of assembly labels which are preserved during the compilation process and denoted in the symbol table of the ELF-binary file, so that the adresses of the TPP positions in the binary can be retrieved [FBH+16]. A characteristic of this approach is that the assembly labels represent optimization barriers, so that the compiler cannot move code across these barriers for compiler optimizations. This can affect the compiler optimization and thus lead to timing overestimations in comparison with the execution times of fully optimized code. The effect can be mitigated by the development of optimized code generation processes that minimize the number of region context switches in the C code. However, due to the SC MoC iur-scheduling regime, which is introduced in detail in Chapter 4, there exist worst-case models that enforce a high number of context switches because of thread communication, for example because two threads alternate in reading and writing a number of variables. An example for this is the test model Weaver shown in Figure 7.8. This model requires the alternate repeated scheduling of three sibling regions that read and write a set of nine variables.

Thus, this approach is especially suitable for timing analysis for PRET systems, which are discussed in Section 3.4, where compiler optimizations have a small effect on the timing behaviour of the program, as the order of instructions is of little influence to the execution time. The processor targeted in this example implementation is similar to PRET architectures as it is a 32-bit single-cycle MIPS processor with a fast scratchpad memory that is assumed to hold the complete program code [FBH+16].

The task of preserving code range information without influencing compiler optimization remains as an interesting research topic for future work as it concerns general architectures, which is further discussed in Section 9.2.

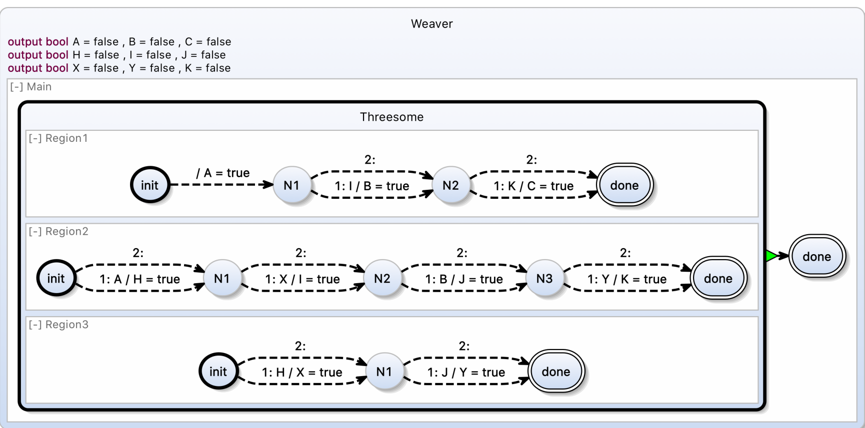


Figure 7.8. This example model enforces forth-and-back-scheduling between the regions due to high amount of thread communication. Writes to a variable have to be scheduled before the reads.

7.6 Identifying State

As explained conceptually in Section 6.9, for state-based models, the modeling tool can convey information on state variables to the analysis tool to enable a state-based analysis which is able to rule out impossible state combinations and thus reactive infeasible paths. The challenge on modeling tool side is to determine which variables in the code have to be identified as state variables. For some groups the answer is straightforward in the example implementation. Input variables are no state variables as they are decided by the environment in every tick. Also, the internal simple guard variables and conditional guard variables do not qualify as state variables, as all guards that are needed in the next tick are explicitly stored in register variables called PRE-variables by the KIELER tool. Consequently, these PRE-variables have to be listed as state variables. Additonally the _GO, which indicates wheter the system is in its first tick, is treated as a state variable. The PRE-variables and _GO are also initialized at the beginning of the program execution with the reset function.

The case is less clear for output variables and local variables. As elaborated by Motika [Mot17], in KIELER SCCharts, both are not initialized in the reset function and their values remain undefinded before the variables are first written. The KIELER SCCharts implementation makes no guarantees on output and local variables that are not properly initialized by the programmer [HDM+14]. However, both output and local variables keep their value, over arbitrarily many ticks, until a new assignment takes place. For local variables there is the special characteristic that they exist only in the scope of their declaration superstate. Unless the variable is declared static, the storage of an assigned value happens only until the corresponding state is left and reentered, upon which the local variable is reinitialized from a given constant [HDM+14; Mot17].

This means that the programmer is responsible for correct initialization on program start for output variables and on every entering of the variable scope for local variables. However, it is not requested that outputs or local variables have to be explicitly reinitialized in every tick, both are reinitialized from a register that stores the last value. For local variables this holds until the scope is left and reentered. It follows that both kinds of variables must be named as state variables, as they store program state from one tick to the next. On the other hand it is unproblematic to declare these variables as state variables, if we assume that the program has been found constructive, as we can then assume that the initialization behaviour, even concerning reinitialization of local variables on scope re-entry, is correctly decided in the code or without effect. The analysis of constructiveness, including detection of indeterminate behaviour caused by initialization faults, is an orthogonal question which is assumed to be solved in this case.

As the goal of providing state assumptions is to enable a tighter timing analysis that is able to detect a special kind of infeasible paths, as detailed in Section 6.9, it is possible to reduce the number of state variables to be respected by leaving out variables that are never read. A variable that is not read cannot decide program branching and thus is without influence on the feasiblity of program paths. For many programs this is the case for output variables. It is important however that the variable must be without read access in the generated code.

```
1  Function tick
2  InitFunction reset
3  GlobalVar bumper 0..1
4  GlobalVar accelerator 0..1
5  State motor
6  State PRE_g2
7  State PRE_g4
8  State PRE_g6
9  State _GO
10 State _PRE_GO
11 FunctionWCET errorLog 6410730
```

```
12 FunctionWCET writeLog 14483659
13 FunctionWCET getImage 16064301
14 FWCET entry 1
15 FWCET 1 2
16 FWCET 2 3
17 FWCET 3 4
18 FWCET 4 5
19 FWCET 5 6
20 FWCET 6 exit
21 WCP entry exit
```

Listing 7.1. The listing shows the contents of the generated timing request file for the improved robot model in Figure 6.16. The file names first the function to be analyzed, then the reset function, then assumptions for global variables and state variables, followed by function WCET assumptions and finally the timing requests.

7.7 Timing Request File Generation

The KIELER SCCharts modeling tool provides the experimental timing analysis tool with the generated C code with TPPs as well as a timing request file that has the file ending .ta. This request file is automatically generated. For the improved robot model shown in Figure 6.16 on page 109, the contents of the synthesized file are as shown in Listing 7.1. The listing displays a timing request file for the KIELER default setting matching the user interface, in which fractional WCET values and on demand hotspot highlighting and focus-and-context expansion are offered. In the first line, the modeling tool specifies the function that should be analyzed, which is the tick function for the model. The next line specifies the reset function, which in the KIELER tool is located outside the tick function. Lines 3 and 4 communicate to the analysis tool that the two inputs are boolean inputs and their value will be 1 or 0. This is followed by the state assumptions in line 5 to 10, which are automatically retrieved respecting the guidelines described in the previous section Section 7.6. The state variable assumption for the output motor is not mandatory, as this output is not a read variable, see the previous Section 7.6 for further explanations on this topic. A corresponding optimization of the state variable detection is currently not implemented in the KIELER tool.

Lines 11 to 13 contain the WCET assumptions for the called functions. These are read from an assumption file with the .asu file ending. Conceptually, such an assumption file can either be provided by hand or automatically with the help of a timing analysis tool. In the example implementation, these files are provided by hand as in our evaluation we made use of the possiblity to design settings by preparing characteristic models as well as specific timing assumption constellations, as explained in Section 8.1. Finally the timing analysis requests themselves are written. For this setting the modeling tool needs the fractional WCET values for each pair of consecutive TPPs, which are requested in lines 14 to 20, including the implicit TPPs at the beginning and the end of the function, denoted as `entry` respectively `exit`. Finally, a request for the critical path is posed in line 21 for the complete program, i.e. from its `entry` to `exit` points to prepare the hotspot highlighting. Time value aggregation is done on the modeling tool side according to the principles explained in Section 6.5.1. The modeling tool does internal bookkeeping on the nature and order of the requests and also of the representation relation between TPPs and regions, so that the timing analysis tool can answer with a mere sequence of numbers that will be attributed to the right requests by the modeling tool. The only rule it has to follow in this is that the order of response numbers must correspond to the order of requests they are related to.

7.8 Modeling Pragmatics for Interactive Timing Analysis

As interactive timing analysis especially aims at increasing modeling productivity in timing related revisions, it is natural to investigate combinations of this feature with pragmatics-related methodologies to further strengthen its effect on modeling efficency. A discussion of approaches in the research field of modeling pragmatics is given in Section 3.5. Two corresponding features are integrated into the example implementation of the interactive timing analysis. The first is a focus-and-context expansion view for the display of large models, which is explained in Section 7.8.1, and the second concerns the display of function timing information on mouse hover, detailed in Section 7.8.2.

7.8.1 Focus and Context for Large Models

While the hotspot highlighting offers quick orientation in timing-related revisions also for big models, to read the detailed region numbers often needs browsing in different zoom levels. To help the modeler in working with large models, the KIELER tool offers a customization of the hotspot highlighting view, in which the regions that are timing hotspots or contain timing hotspots are expanded and regions with little timing contribution are collapsed. Thus, the hotspot highlights can be regarded in their context, but this context is shown in less detail. See Section 7.2 for the corresponding user interface.

Figure 7.9 shows the synthesized timing focus expansion view for the model MedicalAid, whose fully expanded version can be compared in Figure 8.4 on page 153. All regions that have a deep timing value under 10 percent of the overall WCET of the model are collapsed automatically. Referring to the deep timing value leverages the calculations already done by the timing analysis and thus adds no further complexity. The deep timing value reflects also the timing values of enclosed inner regions and thus identifies regions that might be collapsed without hiding superstates whose regions are timing hotspots themselves. The collapsing of the regions can be reverted for a single region by clicking the [+]-symbol in the left upper corner or for all regions at once using the button labeled Expand All Regions which is situated with the action options in the diagrams view and can be seen in Figure 7.3 on page 130. A larger example can be found in Appendix B.

An alternative method to deal with the problems of timing display for large models is proposed as future work in Section 9.1.

7.8.2 Function WCET on Mouse Hover

The interactive timing analysis interface allows to seperate the timing analysis of the tick function from the in-advance-calculation of the WCET of external functions, see Section 6.7. For each model with function calls in the example implementation there exists an assumption file with timing information for the called functions. Therefore timing information on the function calls can be made directly available to the modeler without even

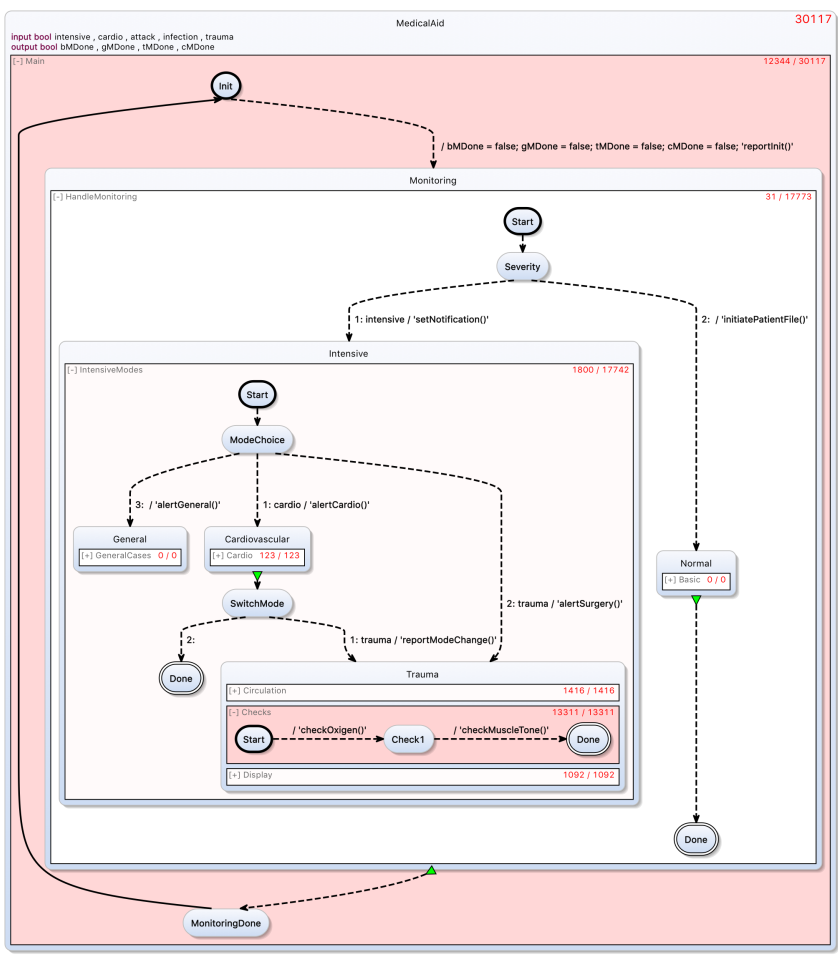


Figure 7.9. The model MedicalAid in a view with hotspot highlighting as well as a timing related focus-and-context representation, in which regions with a deep timing value under 10 percent of the overal WCET value are collapsed. The uncollapsed view can be found in Section 8.1.1 on page 153.

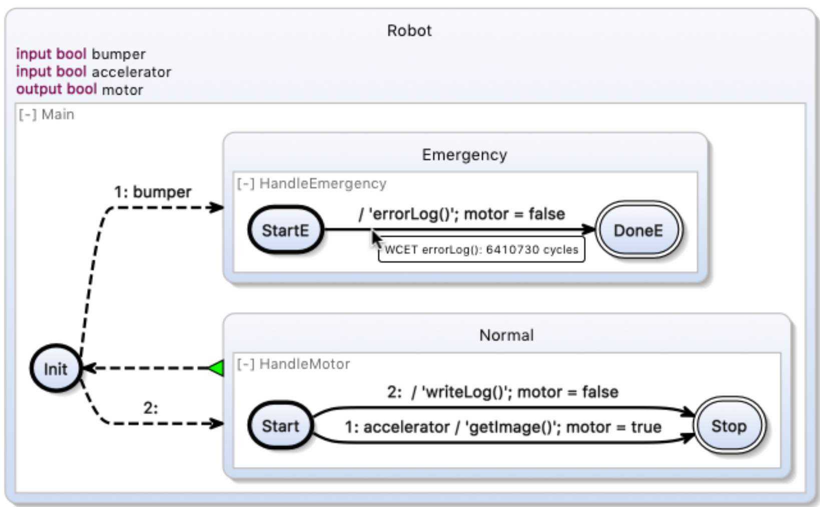


Figure 7.10. Screenshot: Function WCET on mouse hover.

invoking the interactive timing analysis cycle. The KIELER tool offers the display of function time values on mouse hover, as can be seen in the screenshot in Figure 7.10.

When the modeler hovers the host code call transition, the tool shows a text field which reports the stored timing information on the respective function.

Chapter 8

Evaluation

The evaluation of the approach described in this thesis was conducted with the help of the example implementation introduced in Chapter 7 and consists of four parts: First, the example implementation was tested with three different types of benchmarks: models that were designed explicitly to test specific characteristics, test models from the KIELER project, including models originating from industrial collaboration, and also a suite of randomly generated SCCharts of varying complexity. This part is elaborated in Section 8.1. Secondly, an artifact was successfully submitted to the International Conference on Real-Time Networks and Systems (RTNS) 2016 to gather feedback from an external peer review process, which is explained in Section 8.2. As a third part I introduce an informal validation technique for both the timing analysis and the compilation of extended language features with specifically designed SCCharts models that reflect desired semantical behaviour in an expected timing result. This approach is elaborated in Section 8.3. Finally, in Section 8.4 follows a description of the fourth part, a user study that was conducted to obtain indications on the practical value of the introduced approach.

8.1 Test Cases

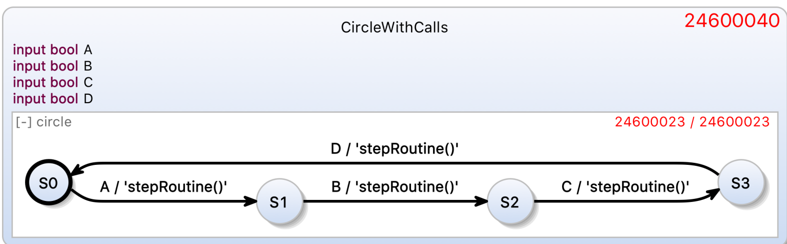
8.1.1 Specifically Designed Test Cases

To validate our timing analysis approach with the example implementation introduced in Chapter 7, we introduced a set of test cases that were modeled by hand. The host code calls in the models are fictional and the timing assumptions for the host code calls are also given by a hand-written file that

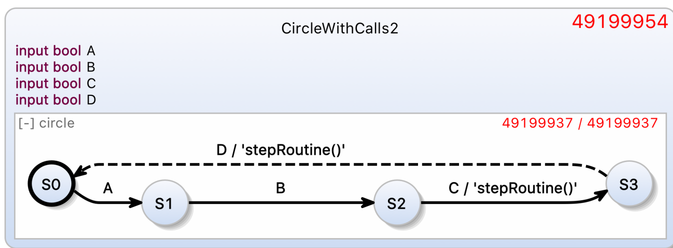
is automatically polled and integrated into the generated timing request file. The goal of this is to allow the usage of each model with a multitute of timing assumption constellations and thus to be able to reuse it for a number of test setting configurations. Each of the models was designed with a specific test focus in mind. In the following, some of these test examples are introduced to illustrate the validation concept.

For a chosen test focus, the models are built to cover different levels of complexity. For example, the small model CircleWithCalls shown in Figure 8.1a, and its variant CircleWithCalls2 in Figure 8.1b, are used as basic corner case models to test the state based analysis based on state assumptions as introduced in Section 6.9. Both models can only be analyzed correctly, if the analysis is able to handle models that are conceptually perpetually running. Both models reenter their initial state during execution. Also, in case of CircleWithCalls, the analysis must determine that the host code call stepRoutine() can contribute its WCET value only once, due to the delayed transitions. The illustration shows a successful run for a function WCET assumption of 24599908 processor cycles, For CircleWithCalls2, the worst case cannot be reached before A and B have not been perceived in ticks before, and due to the immediate edge from S3 to S0, the two host code calls belong to the same tick. Figure 8.1b shows a successful analysis run for the same assumption regarding the host code call.

After the dedicated basic models generate a satisfactory outcome, we also test more complex models, to adress more involved aspects of the current test focus. For example the model Controller, shown in Figure 8.2, whose structure is designed such that in specific timing assumption settings, worst-case behaviour is reached only after a number of ticks have passed and in which only a combination of three timing hotspots leads to a higher timing value than that of a fourth hotspot that would otherwise dominate the WCET behaviour of the model. The view in Figure 8.2 is generated by the KIELER tool with hotspot highlighting and timing value representation in milliseconds. The most costly host code call in the model is the one to writeAccidentLog(), which is executed in the first tick after the one in which the superstate Operating is entered. It has a WCET assumption of 3125ms. None of the other host code calls can surpass this value alone. However, there is a constellation that occurs four ticks later, in which the host code



(a) The test model CircleWithCalls is designed to test basic functionality of the state based analysis. The analysis has to determine that the function call can be executed only once in a tick.



(b) Model CircleWithCalls2 also is a basic test of state based analysis. The worst case constellation cannot be reached before the occurences of A and B have been registered in the ticks before.

Figure 8.1. Basic test models for state based analysis. In both cases, the analysis must be able to handle programs that are conceptually perpetually running. The time values are given in processor cycles.

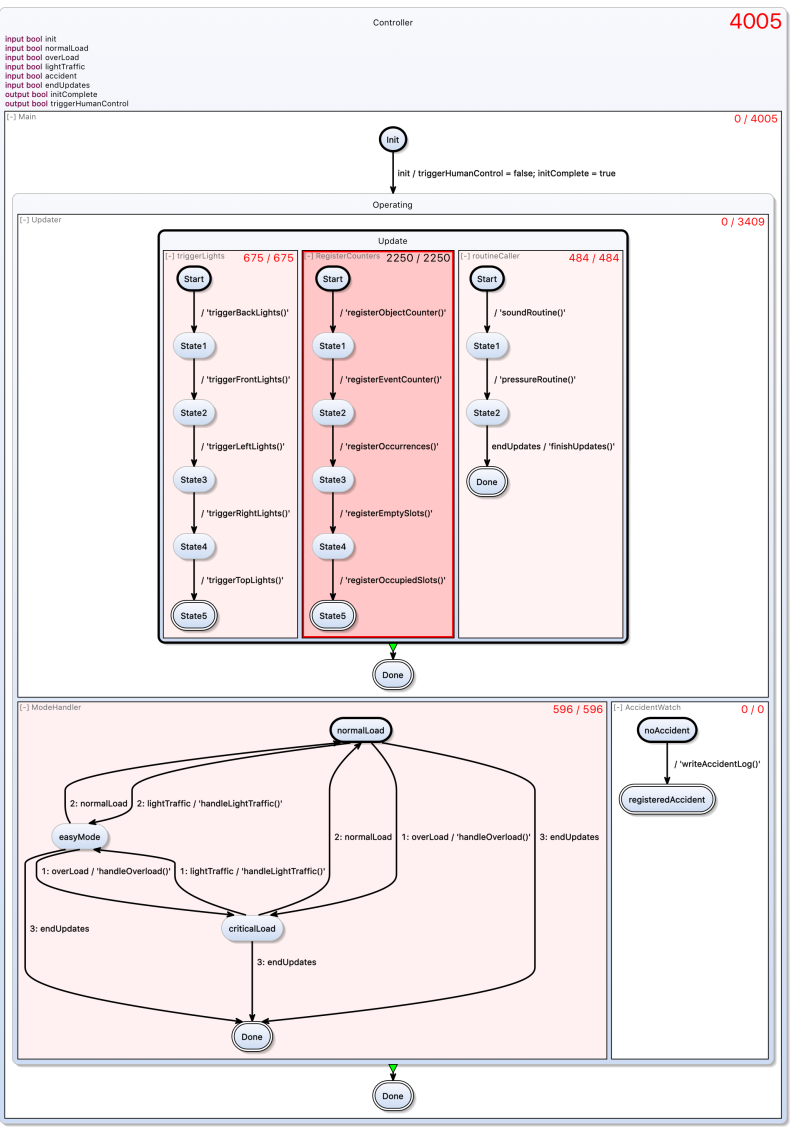


Figure 8.2. The test model Controller is designed for more involved testing of the state based analysis. The time values are given in milliseconds.

call to the function registerOccupiedSlots happens, which is next to writeAccidentLog in WCET cost and takes 2250 miliseconds to execute. Together with the timing values of two parallel regions, the region that executes this host code call surpasses the timing value of the region AccidentWatch that calls writeAccidentLog, so that these regions are on the critical path, but AccidentWatch is not. It has already terminated, when the worst-case constellation is reached. Note that the timing values of the three sibling regions are added up though the regions are parallel, because the example implementation generates sequentialized code.

Finally, we use general nontrivial models like FunPark2, which is shown in Figure 8.3 to validate that the concepts are feasible for larger models. FunPark2 is a model with 150 nodes and 40 regions that does not involve host-code calls and has no predominant timing hotspot, so that a number of regions contribute rather evenly to its worst case timing value.

Another aspect in the testing focus were the characteristics of fractional time values, which can be seen best in test models with mutually exclusive regions like MedicalAid shown in Figure 8.4. Region Basic in state Normal, displayed as the rightmost inner region, for example is not on the critical path, accordingly the interactive timing analysis shows no time value contribution in the hotspot highlighting. Though regarding the user interface, the example implementation shows fractional WCET values, which are suited best for hotspot highlighting, from developer side it is also possible to poll for the Local Worst Case Execution Time (LWCET) value of this region and thus to test the difference between local and fractional time values. The local time value result for region Basic for the same model with identical function timing assumptions is returned as 1419ms, as can be seen in Figure 8.5. This corresponds to the sum of the timing assumptions for the three host code calls that can be executed in one tick in this region. Thus the difference is successfully displayed.

The hotspot highlighting and accordingly also the WCET path analysis request is tested with models like Feeder which is displayed in Figure 8.6. This model in this assumption setting has a characteristical hot spot that concentrates on a single region and one of its child regions.

Furthermore, we used models like MultiWait to test the interactive timing analysis for a high number of TPP, caused here by the multitude of small

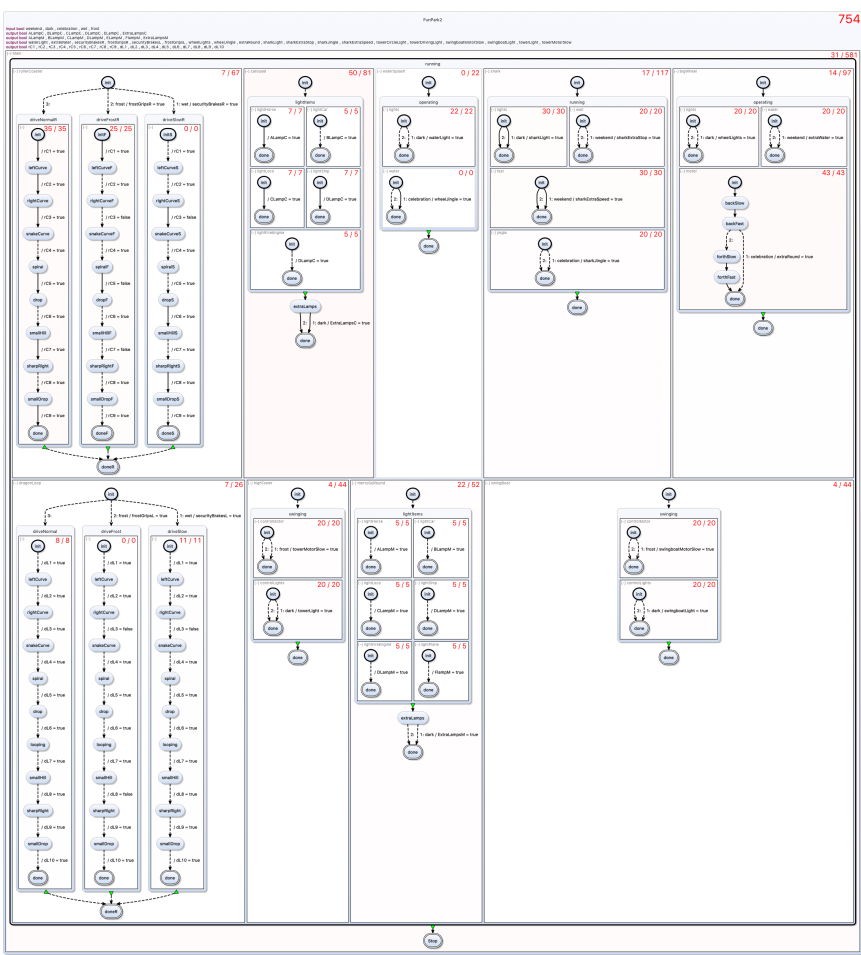


Figure 8.3. The test model FunPark2 is used as a general non-trivial benchmark. The unit of the time values is milliseconds.

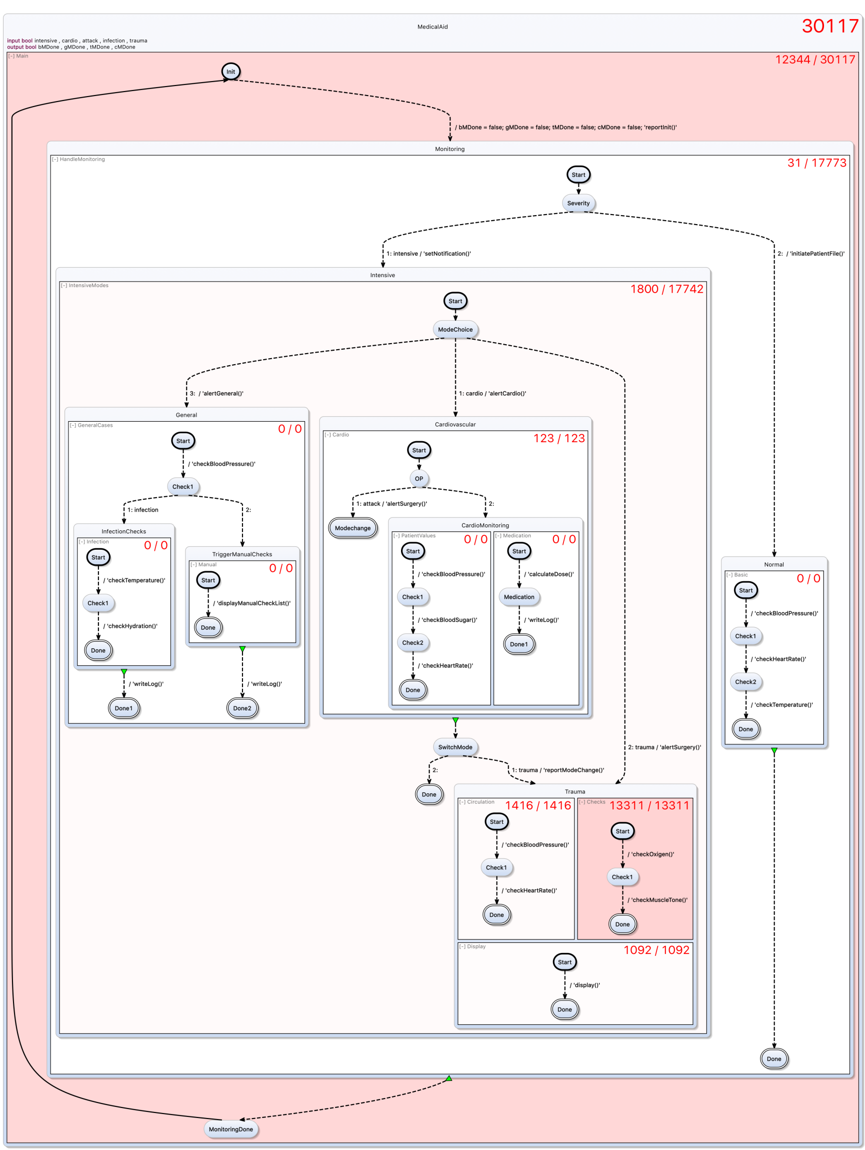


Figure 8.4. The test model MedicalAid shows the characteristics of fractional values due to mutually exclusive regions. The unit of the time values is milliseconds.

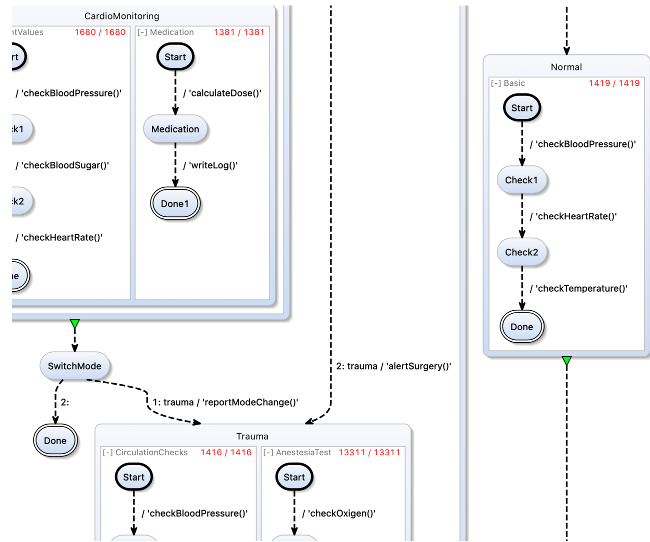


Figure 8.5. The local time value of region Basic in MedicalAid. The unit of the time values is milliseconds.

regions, shown in Figure 8.7. The worst case execution time behaviour is given when all three inputs are true. Table A.1 contains an overview covering all test models in the appendix on page 189.

Note that an evaluation of the performance time of interactive timing analysis is not feasible with the current experimental timing analysis tool, as it is not scalable for larger models due to its exhaustive search approach. Nevertheless, for all models introduced above the reaction time was well adequate for interactive usage. The system has also been interactively tried in the user study described in Section 8.4. However, it is desirable to employ performance measurements when the implementation of the KTA tool is finished.

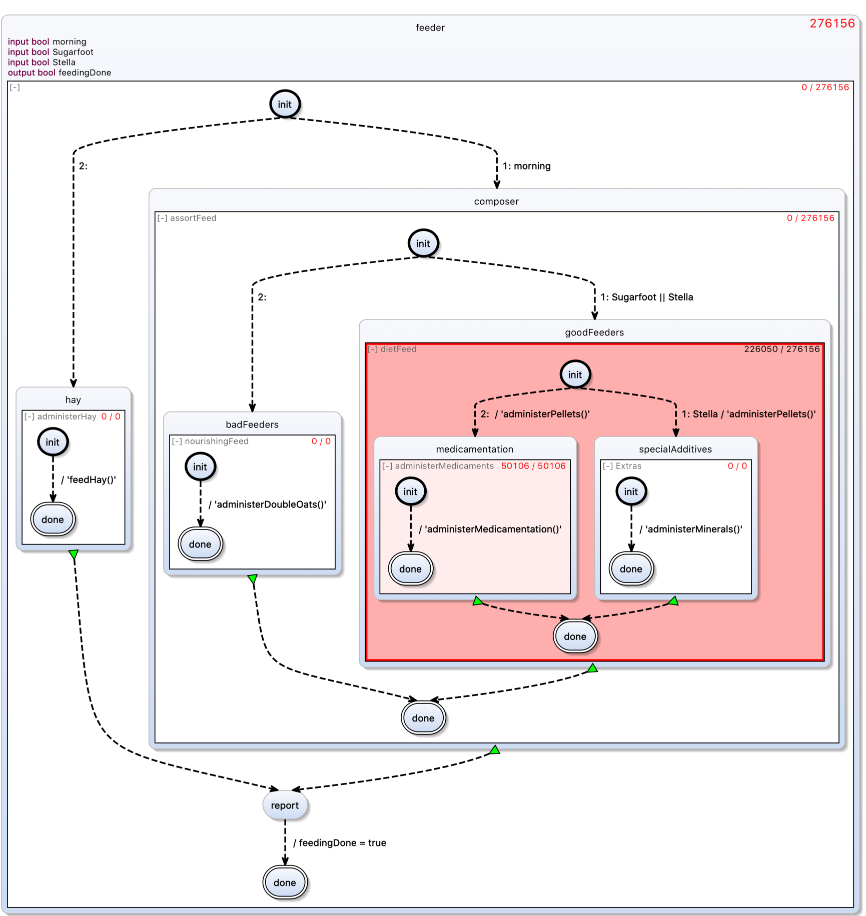


Figure 8.6. The test model Feeder for the testing of hotspot-highlighting.

Figure 8.7. The test model MultiWait provokes the generation of many TPP because of its high number of small regions. The unit of displayed timing values is milliseconds.

8.1.2 Test Models from the KIELER project

Apart from the test models that were specifically designed for the validation of the interactive timing analysis, we employed test models from the KIELER model suite. The idea behind this was not only to use a higher number of test models but also to use models that originated in practical use of SCCharts or were designed for various purposes to avoid confined testing. One example is the test model SCU_Monitor in Figure 8.8, which models an avionics system control unit in context of a flap system. The model originates from an industrial collaboration [FKR+05; FKR+06; FH09a]. It also illustrates that many of these test models use extended language features, in this case for example signals for which the compilation establishes emulation structures. An overview of the tested models is given in Table A.1 in the appendix on page 192.

8.1.3 Generated Test Models

Finally, a set of test models generated by a random model generator was tested. These models cover a range of model sizes and complexity degrees. The random generator for .sct files is part of the KIELER project and written by Steven Smyth. For tests of the interactive timing analysis I customized the generator to add with a chance of 0.25 percent for each transition a randomly chosen host code call from a given list of ten host code calls, for which there exists a timing value assumption file that can be used to test the generated models. The advantage of generated test models lies not only in the increased number of models that can be tested, but also in the chance to come across unusual constellations that are seldomly used in practice, but nevertheless valid. An example test model is shown in Figure 8.9 with timing values displayed in the unit of processor cycles. and the tested models are also listed in the overview in Table A.1 in the appendix on page 192[1].

[1] All tested models are also available in the group repository of the Embedded Systems Group of Kiel University, either in the models-repository of the KIELER project or in context with this work.

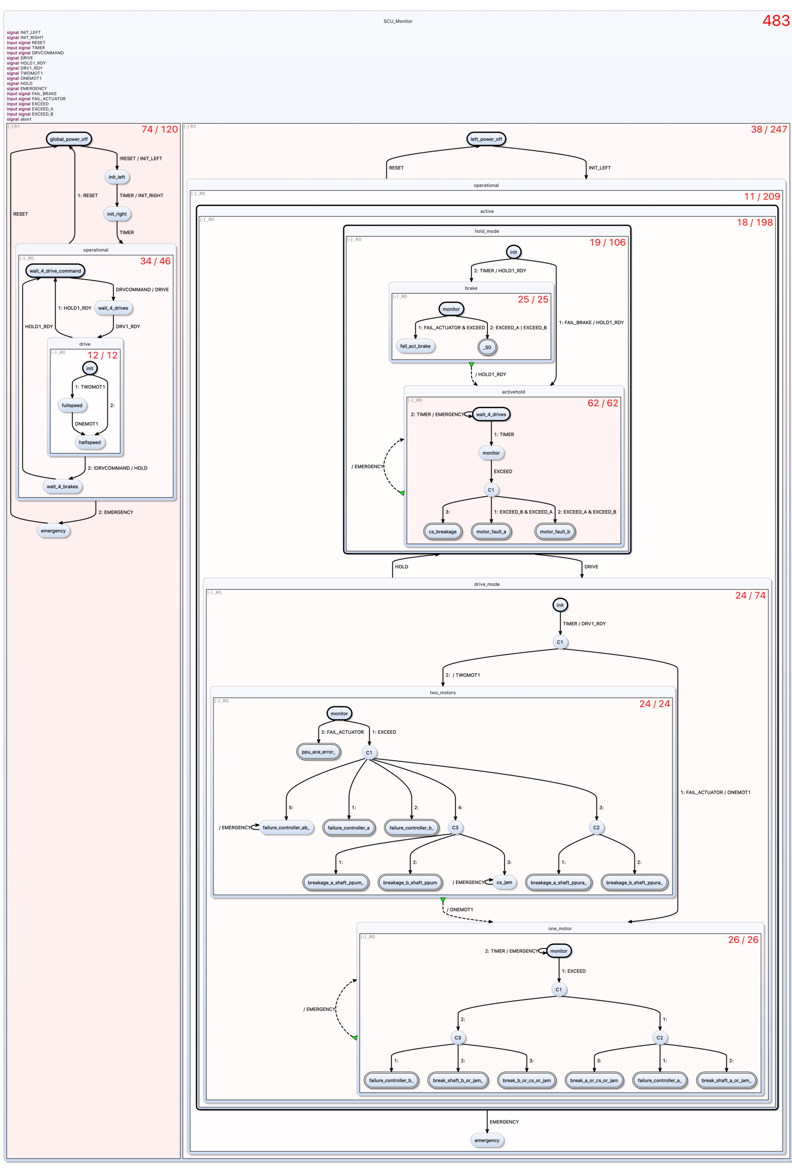


Figure 8.8. The test model SCU_Monitor originates from a collaboration with the avionics industry [FKR+05; FKR+06; FH09a].

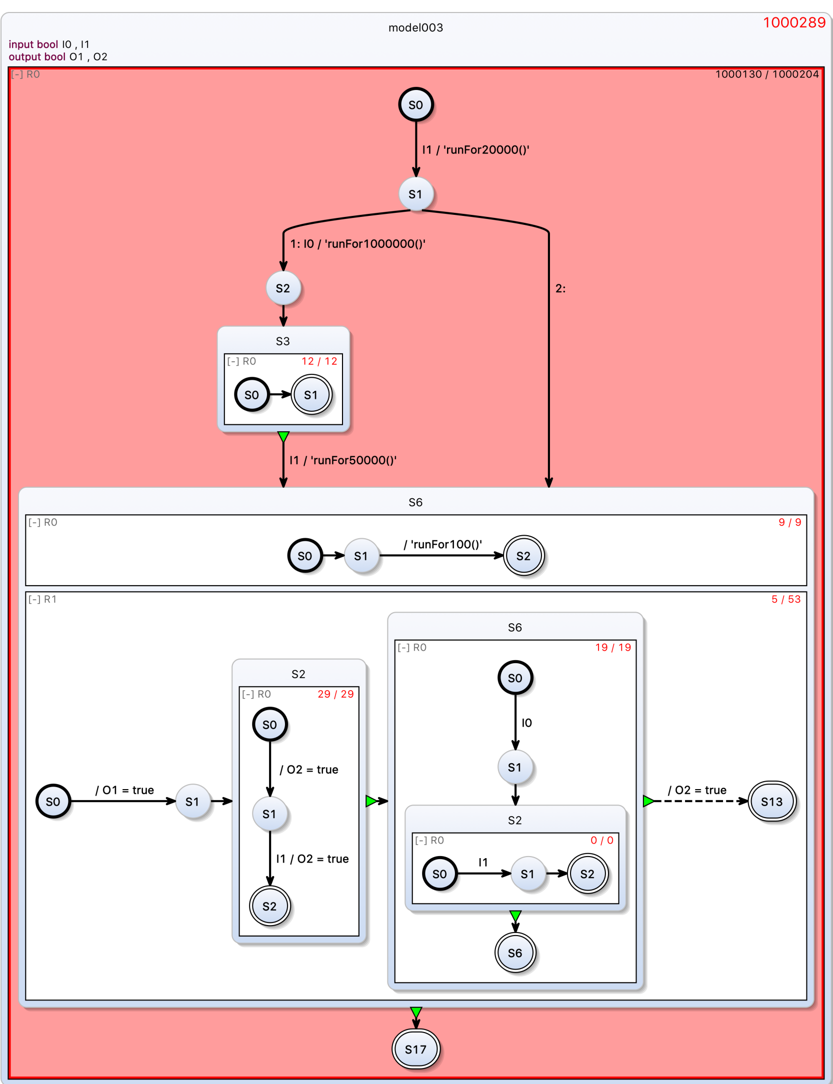


Figure 8.9. Test model model003 has been automatically generated with a customized version of the sct generator in the KIELER project.

8.2 Artifact submission

Figure 8.10. The RTNS approval seal granted for [FBH+16] .

To get feedback from a peer review, we also submitted our example tool for the artifact evaluation of the RTNS 2016 in relation to the corresponding conference paper [FBH+16]. Submitted was the example implementation which was offered in form of a VirtualBox Appliance. Additionally, the submission contained a handout on how to install the system with the corresponding versions of KIELER and the experimental timing analysis tool and a short user manual. Note that the implementation offered not yet any support for focus and context and function timing values on mouse hover in the corresponding version. A tutorial video was part of the submission. Both video and virtual box appliance have also been published on the demos page of the KIELER project. Also, a basic set of models, which was described in the conference paper, was made available for the evaluation. These are the models described in Section 8.1.1. The handout also included instructions on how to design further models. The artifact was successfully reviewed and the approval seal, as shown in Figure 8.10, was granted for the related conference paper [FBH+16] and its presentation.

8.3 Validation of Extended SCChart Features and Timing

In this section I introduce an approach to informally validate the compilation of extended SCCharts features together with the timing analysis by mapping an expected timing behaviour to the semantically correct model behaviour. For the described models, the timing analysis result is expected to also reflect the semantic impact of the language construct. Thus, these test cases have also been used in testing the transformations of the extended features. Unexpected timing behaviour serves as a warning sign for errors in the code generation itself as well as the interactive timing analysis.

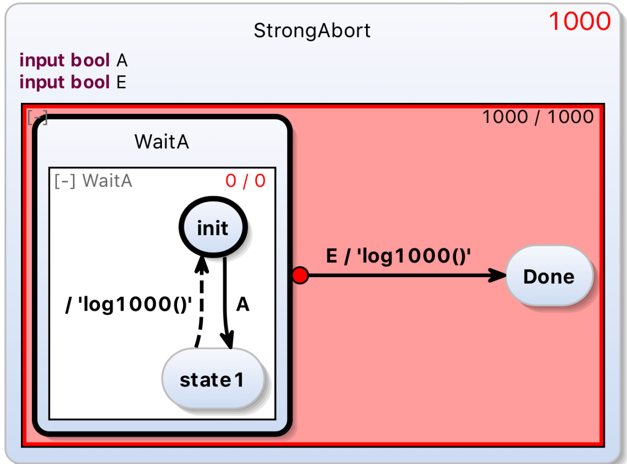
To achieve this, the models are configurated with strategically placed fictional host code calls that add a certain time value in case the execution of the call is reached. The functions are named in accordance with the associated WCET assumptions, for example a call to log1000() will contribute a time value of 1000 milliseconds to the overall WCET. The time values for these test functions are chosen to surmount the typical execution times of the surrounding constructs in the model significantly, so it can be easily determined whether the call has been triggered. All time values in this section are given in milliseconds if not stated otherwise. Recall that very small timing amounts of under one millisecond are displayed as zero values, so that the focus is drawn to the control calls.

The following descriptions cover several important extended features, but do not describe all tested models. An additional test case for actions is described in Section 7.3, as it has interesting aspects concerning tracing in the example implementation. All tested models, including those not described in detail in this section, are also listed in the model overview in Table A.1 in the appendix on page 192.

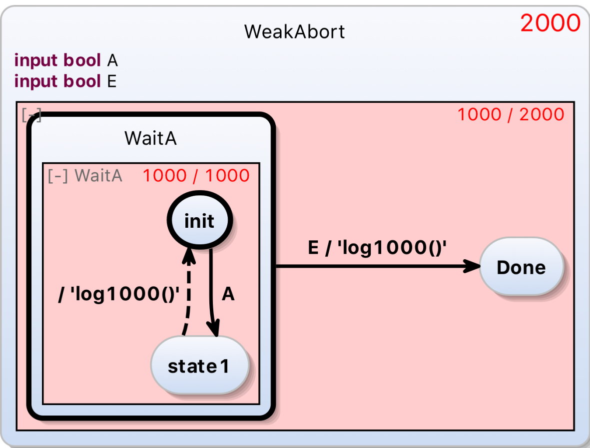
8.3.1 Weak and Strong Abort

Figure 8.11 shows the investigation of strong and weak abort with the help of two example models. The difference between the two types of aborts is that while a weak abort allows the immediate behaviour of the aborted state to be performed before abortion, the strong abort ends the execution of the state promptly. To test this difference in behaviour, I performed a timing analysis with two models that are identical but for the type of abort transition. In Figure 8.11a, the super state called WaitA is aborted strongly, depending on the boolean input E, whereas in Figure 8.11b, this abortion is weak.

In both models, a call to log1000() is placed as effect of an immediate transition within the state WaitA, which is the source of an abort transition. This abort transition also has the effect that a call to log1000() is included in the timing calculation. Thus, for the model in Figure 8.11a, we expect a timing value that includes the time value for the function call only once, while for the model in Figure 8.11b, the timing analysis should include the



(a) Strong abort: the content of the state WaitA is not executed in case of an abort.



(b) Weak abort: immediate state contents are executed before WaitA is aborted.

Figure 8.11. Test models for strong and weak abort. Time values are given in milliseconds, a call to log1000() costs 1000ms. In the case of a strong abort, the state WaitA is aborted whithout any of its contents being executed. In case of the weak abort, the immediate transition can be taken before abortion takes place, thus the timing value includes two calls to the function.

2000 ms for two function calls. The comparison of the two figures shows that the test has been successful.

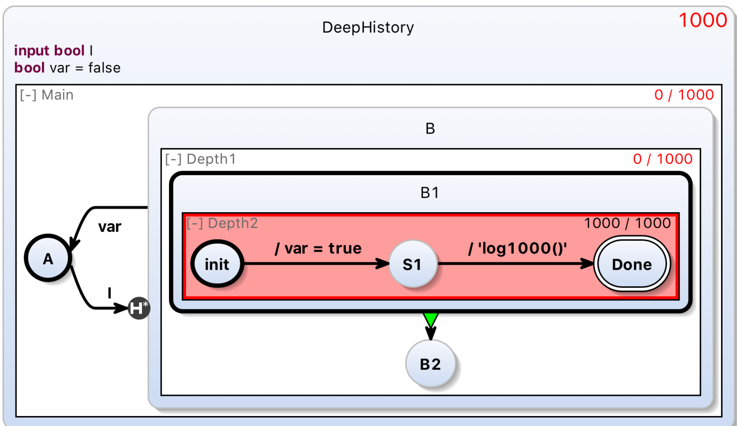
8.3.2 Deep and Shallow History

Two test models that compare the semantics of shallow and deep history with the help of interactive timing analysis are shown in Figure 8.12. Similar to the setup for weak and strong abort, the two models are identical but for the different types of history transition involved. In both models, the Main region includes two states, the simple state A and the super state B. State B itself also contains one complex and one simple state, named B1 and B2 respectively.

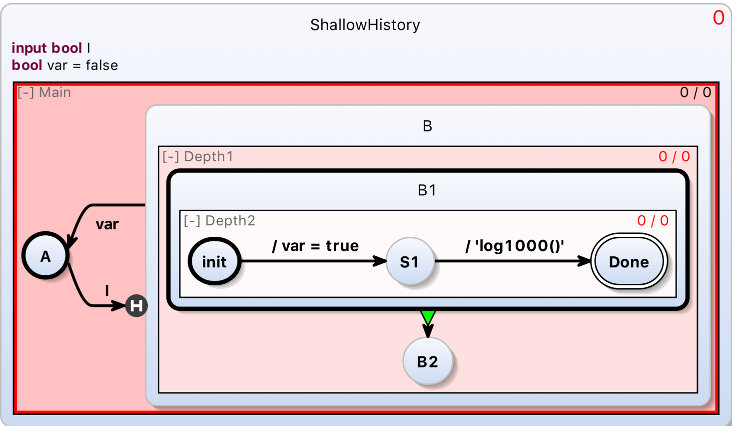
The execution starts in the state A of region Main. After the first tick, state B is entered with the first occurrence of the input I. This is performed with a deep history transition in the model in Figure 8.12a and with a shallow history transition in the model in Figure 8.12b. The deep history transition lets the control flow in state B resume in the state in which it was located when B was left last, keeping track down to the deepest hierarchylevel of the state. The shallow history transition in contrast only regards the first level of content in state B, which means, it knows only whether B1 or B2 have been active before.

For the first passage of the history transitions this makes no difference, as B has never been active before. Execution in B starts in its initial state B1 and as it is a superstate, transitively in the initial state of B1, which is init. In the next tick, the transition to state S1 is taken unconditionally, with the effect that var is set to true. This activates the weak abort condition for the superstate B, which is never reset.

The weak abort is executed with no prior execution of the transition from S1 to Done, as that transition is not immediate. The execution resides in state A, until input I is true for the next time. Now the control returns to the state B and the difference between deep and shallow history becomes evident. In the case of deep history Figure 8.12a, the execution is resumed in S1. The passage of the transition to Done is immediate behaviour now, as the tick consumed by the transition is considered to have passed. So the transition is taken, before the state can be aborted the next time and the



(a) Deep history: the call to log1000() is executed, as the execution is resumed in state S1 after the abort and deep history reset.



(b) Shallow history: the call to log1000() is not executed, as after first entering S1, state B is aborted, variable var having been set to true. With the shallow history reset triggered by input I, the execution resumes in the initial states of B, B1 and init. S1 cannot be reached again, as, var being still true, the abort is triggered again.

Figure 8.12. Deep history resumes the control flow exactly where it was when the super state B was active the last time, while shallow history only tracks the uppermost hierarchy level of the super state.

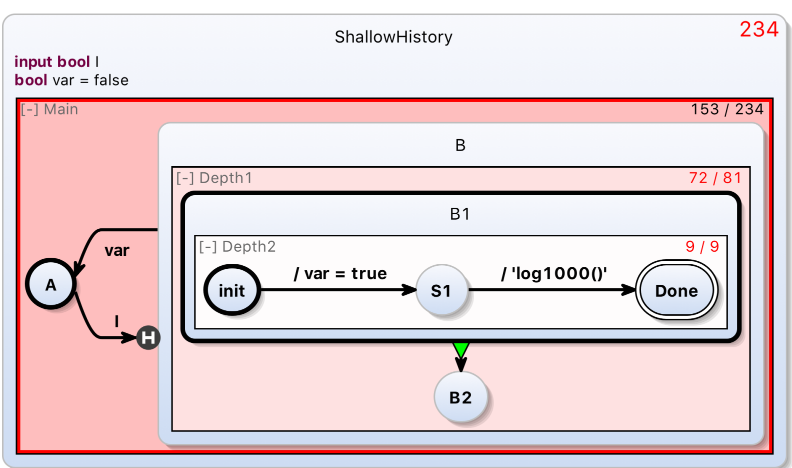


Figure 8.13. An alternative view for Figure 8.12b, showing processor cycles instead of milliseconds.

call to log1000() is triggered. In contrast, with the weak history transition in Figure 8.12b, the control resumes in state B1, but in its initial state, as the shallow history transition does not track, which state in the content of B1 has been active before. Thus, the transition from init to S1 is taken instead of the transition triggering the host code call. Note that Figure 8.12b shows timing values of zero for all regions, but has differentiated hotspot coloring. This is established by time values that are so small that they are not perceived in the milliseconds timing view of the model, but can be seen in the view that feeds back the processor cycles, see Figure 8.13.

8.3.3 Complex Final States

Figure 8.14 shows a simple test case for the semantics of complex final states, i.e. final states with outgoing transitions. Complex final states serve as normal final states when they are active, which means that they trigger a normal termination transition in case all regions of the state that is to be terminated have reached a final state in this tick. However, if the

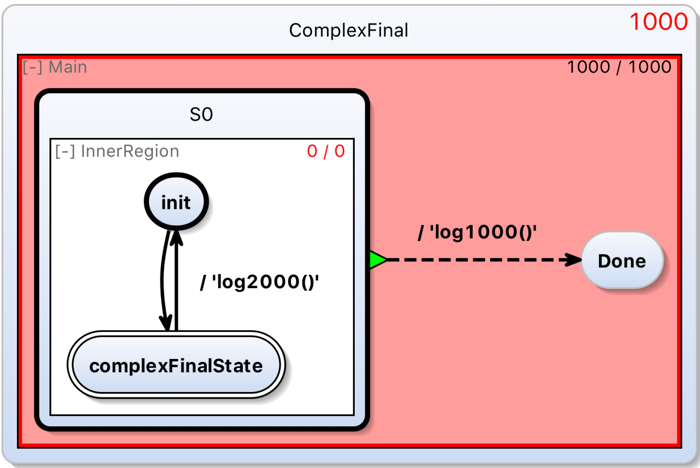


Figure 8.14. In this model, the outgoing transition of the complex final state is not taken, as the state S0 is left with a normal termination as soon as the complex final state is reached.

termination conditions are not complete when a complex final state is reached, because concurrent regions of the state have not terminated yet, the complex final state can be left on its outgoing transitions like a normal state. See Figure 8.14 for an example model, in which a termination takes place as soon as the complex final state is reached. The model complexFinal contains the superstate SO, which is left with a normal termination transition that contributes 1000ms to the overall WCET when it is taken. The state contains only one region, called innerRegion. In this region, control passes from the initial state init to the complex final state complexFinalState unconditionally and without effect. This state could be left with an unconditional transition, whose execution would contribute 2000ms to the overall WCET by a call to the log2000() function. However, in the tick before this transition can be taken, the normal termination is activated, as all regions of the state have reached a final state.

A different behaviour can be observed in the model in Figure 8.15, which is basically the same model, but with an additional region Region2

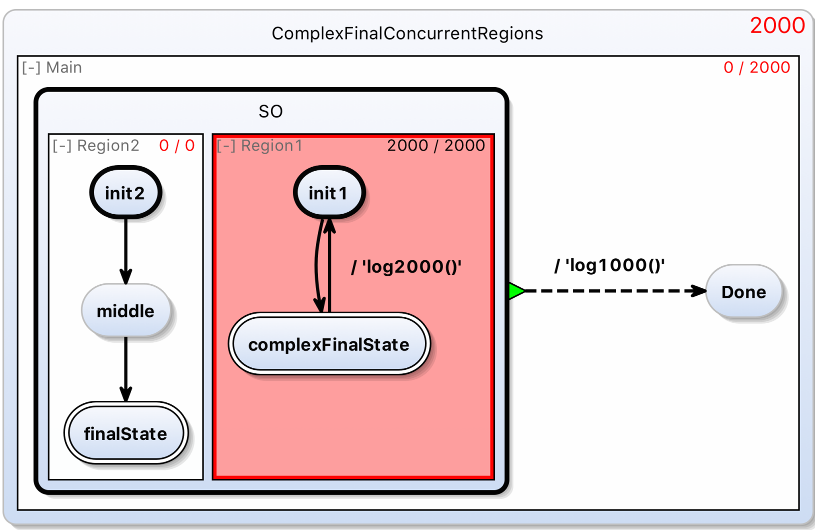


Figure 8.15. A variant of the model in Figure 8.14 with an additional concurrent region in state S0. When complexFinalState is reached for the first time, the conditions for a termination of S0 are not yet given, as Region2 has not reached its final state yet, needing one tick longer to complete its execution. Thus, the transition from complexFinalState to init1 is taken in the next tick.

within the state SO. This region has not reached its final state in the tick in which complexFinalState gets active for the first time. The reason is that it takes two non-immediate transitions to reach finalState instead of one for complexFinalState in the concurrent region. Therefore, in the next tick, the outgoing transition of the complex final state becomes active. This contributes 2000ms to the overall WCET, thus constituting a new critical path. In the related tick, the normal termination will not be triggered, as init1 becomes active, which is no final state. Termination will take place in the next tick, when all regions have reached their final states.

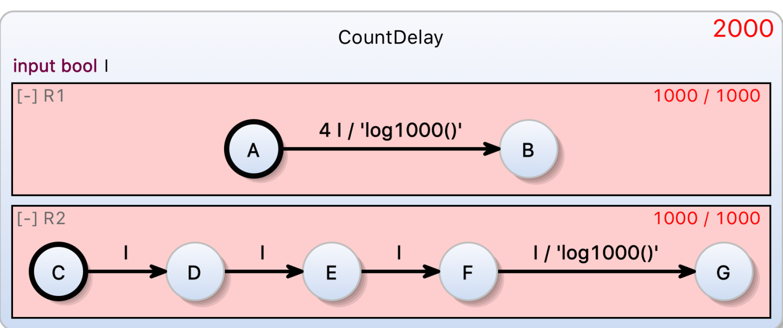


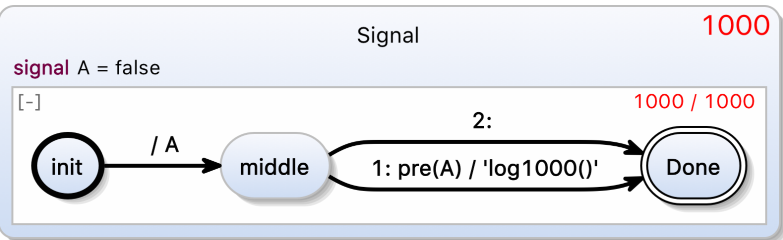
Figure 8.16. The test model CountDelay reaches worst-case behaviour after four occurences of I.

8.3.4 CountDelay

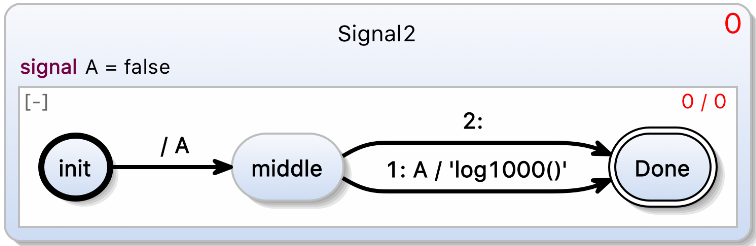
The model CountDelay displayed in Figure 8.16 tests the count delay function, which counts the occurences of an input in different ticks and triggers a transition, when a desired value is reached. The transition from A to B in region R1 is taken after four occurences of input I. The same holds true for the transition from F to G in region R2. R2 expresses equivalent behaviour without the use of a count delay transition. Accordingly, the call to log1000 is expected to happen in both regions in the same tick, thus creating an overall WCET behaviour that includes two calls to the test function.

8.3.5 Signals

As elaborated in Section 4.5, in the SC MoC, signals are emulated with variables. Consequently, this is also the case for SCCharts, and the signals used by the modeler are turned into emulation variables during the compilation process. The two test models shown in Figure 8.17 test this process. In both models, first signal A is emitted, i.e. set to true, in a first tick and in the next tick the effects of this set are tested. The pre value of A should be true, as is successfully tested in the analysis run of model Signal shown in Figure 8.17a. Model Signal2 tests for the signal value itself. Due to the reinitialization to



(a) Signal is a model for the test of the pre value functionality of signal emulation. As A is set in the first tick, the transition with the pre guard is taken in the next step and the host code call is triggered.



(b) Model Signal2 shows whether the reinitialization to false for the signal variable works correctly. In case it does, the transition with priority 2 is taken and the host code call is not triggered as shown here.

Figure 8.17. Signals are emulated in the SCCharts compilation with variables. The two test models shown here cover two different aspects of correct signal behaviour regarding pre value and reinitialization. The value of signal A from the previous tick can only be accessed with the help of pre.

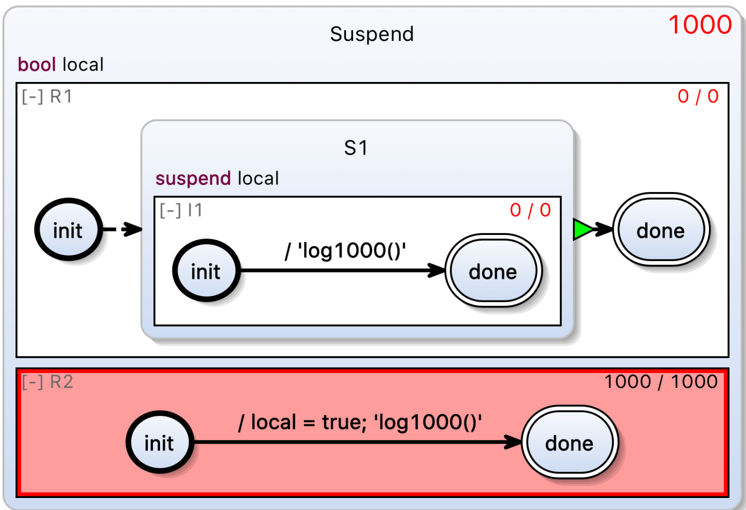


Figure 8.18. The test model Suspend shows only one timing hotspot, as the setting of the local variable triggers the suspend within the sibling region.

false of the emulation variables, it is expected to be false in the second tick, so that in this case the false branch is triggered and the call to log1000 is skipped.

8.3.6 Suspend

The model Suspend in Figure 8.18 shows the behaviour of the suspend feature. As region R2 sets the variable local, the suspend in state S1 becomes active and the execution of the host code call in this region is abandoned for the tick. Thus, only the host code call in region R2 is executed in this tick, leading to a single timing hotspot in this region, which is visualized by the interactive timing analysis.

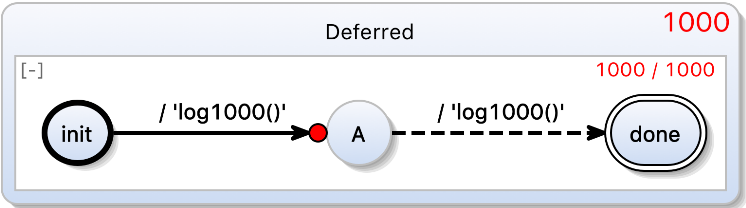


Figure 8.19. The test model Deferred displays the behaviour of the shallow deferred transition. The outgoing immediate transition is blocked by the deferred transition with which state A is entered, so that only one of the host code calls is executed.

8.3.7 Deferred

The test model Deferred shown in Figure 8.19 tests the effect of a *shallow deferred* transition, which has the effect that for the target state, all outgoing immediate transitions are treated like delayed transitions. This can be perceived in the view shown in Figure 8.19, in which the shallow deferred transition is displayed with a red circle at the target state entry point. Though both transitions in the model are unconditional and the transition from A to done is immediate, the two transitions cannot be taken in one tick, which is mirrored in the timing behaviour, as only one of the host code calls is executed in the critical tick. This behaviour would be the same for a *deep* deferred transition. Additionally, for deep deferred transitions also all inner immediate behaviour of the target state is delayed, for example entry actions or inner immediate transitions. As the deep deferred transition is under development in the KIELER tool, it could not yet be specifically tested.

8.4 User Study

To investigate the practical benefit of the example implementation of the interactive timing analysis, we conducted a study with 44 participants, which were to solve a model timing revision task. The entrants were devided evenly into four groups, assigned randomly. Forty participants were students who were in an advanced stage of their bachelor or in their master

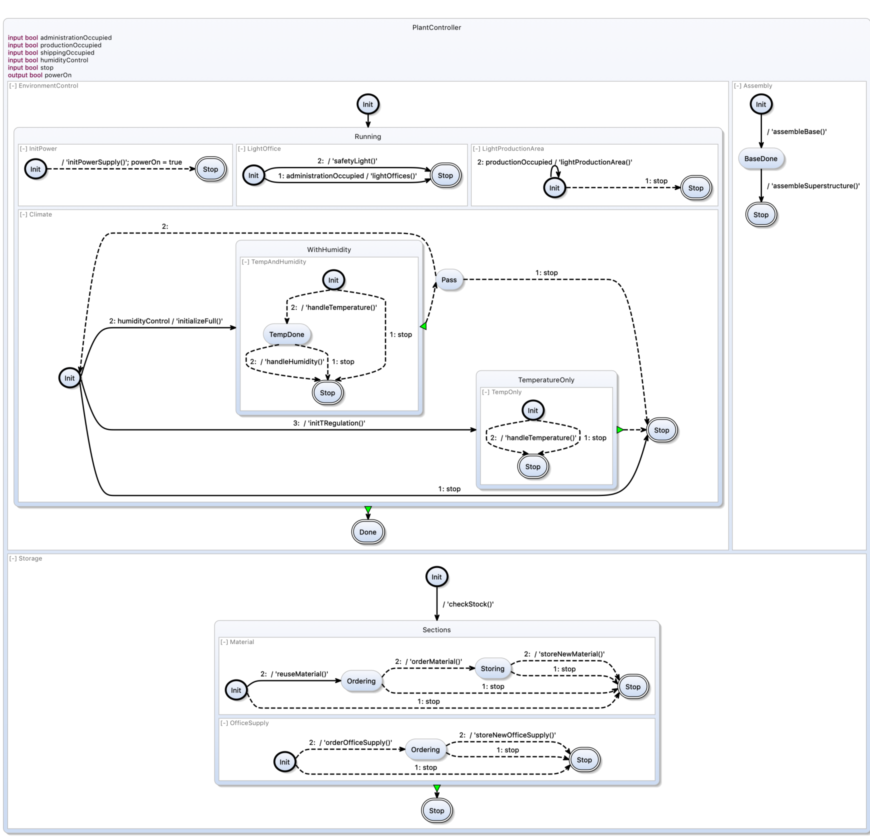


Figure 8.20. The experiment model PlantController

studies, while the other four were researchers that were not involved in the design of the study and were randomly distributed one to each group. All participants had practical experience with SCCharts as a modeling language. Also, all participants were familiar with the notion and characteristics of WCET in relation to SCCharts. By participating, the students could benefit from a partial credit for an embedded systems course, but the attendance was not a requirement for passing the exam with full score.

The basic task was the same for all participants. All were given the model PlantController shown in Figure 8.20, which contains 11 Regions, 33 States, counting also all superstates including the root state, and 17 host code calls. The participants were asked to revise the model working in an instance of the KIELER SCCharts tool, until a given overall WCET value was reached. The diagram view of the KIELER tool showed for all groups this WCET value in the right upper corner of the root state. Otherwise, the diagram views offered by the KIELER tool were different for the four groups. While group 1 saw only the display of the mentioned overall WCET value, group 2 additionally saw the detailed flat and deep fractional timing values for all regions, group 3 had the overall timing value and hotspot highlighting, but no detailed timing values, and finally group 4 had all three types of information, overall value, hotspot highlighting and detailed region value numbers. The highlighting and number display was of the same design as shown in the examples above. At the time of the study, the implementation did not support the three timing value display modes for cycles, milliseconds or percent. The timing values that were given for the fictional host code calls were chosen in a magnitude realistic for the unit of milliseconds, as this was regarded as an adequate choice for the use case of timing revision. In the study, the values were refered to abstractly as time units. The timing values were automatically updated with each saved change of the model. Additionally, the update of the feedback could be triggered by hand by clicking a refresh button. All participants received the same introduction into the interpretation of the timing information as far as it was displayed for the different groups.

In order to reduce the overall WCET value, the participants were allowed to exchange the host code calls to fictional alternative library functions that were characterized as functionally equivalent. For this, each participant

received a printed list of one exchange function for each called function in the model. Timing values for the functions were not disclosed. However, the participants were informed that the timing value of the exchange function could possibly be larger or lower than the one of the originally called function. Apart from the host code call manipulations, only temporary removal of parts of the model was allowed. Permanent changes of the model structure were prohibited.

The time the participants took to solve the task was measured. To avoid inappropriate strain for the participants, attempts that took longer than 25 minutes were aborted. The working times of the 44 participants are denoted with different markings in Figure 8.21, group 1 to group 4 in columns from left to right. Aborted attempts are denoted at the $min.25$ line with an approximated working time of 25 minutes, which would have been the minimal needed working time in case the attempt had not been aborted.

8 of the 11 participants in group 1 were not able to finish the task in below 25 minutes. In group 2, only 3 attempts had to be aborted and all participants finished in time in groups 3 and 4. The longest working time in the two groups with hotspot-highlighting was 15.08 minutes, in group 4 with additional detailed timing values, the maximal working time was 11.58 minutes, also this group shows the lowest standard deviation. This could be an indication of a more effective guidance of the modeler. The results of group 3 are similar in that they also show a large cluster, but the results deviate more in total.

A distinctive result is that all participants of the group 1 without detailed time values and hotspot-highlighting took longer than any entrant of group 4, which was equipped with all kinds of timing feedback. Thus the results of the two groups are completely disjunct. This can be seen as a strong indication that the full interactive timing analysis setup was indeed of practical benefit in this setting. Exept for a single participant result, the same holds true for group 3 that was offered only hotspot-highlighting with the overall WCET value. Though the relation is less clear for group 2, seven out of eleven working time values stay below all values of group 1, so the results still suggest an improvement.

As only a moderate number of participants and a small time frame for each of the 44 individual test runs were given, the study was naturally

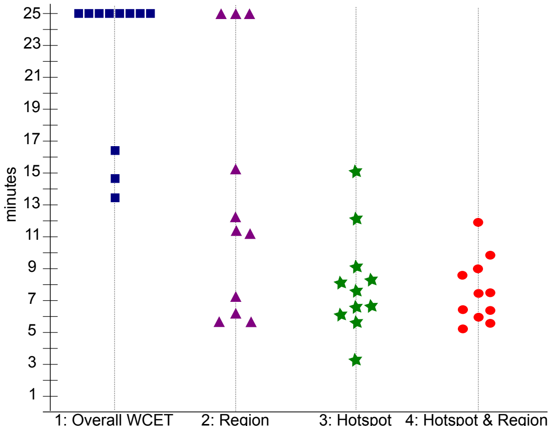


Figure 8.21. The graph shows the working times of the four different groups in the study, left to right, for different types of available timing information.

limited in its setting. An elaborate experiment in an industrial setting with more benchmark models and broader possiblities for model changes would be a helpful future reasearch endeavour to further investigate the effectiveness of interactive timing analysis in the future. Nevertheless, the results of this study make it seem likely that the interactive timing analysis can enhance the productivity regarding timing related model revisions.

Furthermore, the study motivates new research topics for future work. The only groups in which all participants were able to finish in time are groups with hotspot highlighting and also the working times show a similar profile, only slightly to the disadvantage of pure hotspot highlighting. Thus, hotspot highlighting alone might be able to approximate the effect of full interactive timing analysis with detailed region timing values in a competitive quality. This is an interesting aspect, as it might be possible to concentrate on hotspot identification instead of safe concrete values. This could possibly be joined with the idea to combine two analysis approaches for the two different use cases. A fast approximative analysis method could be employed for the enhancement of modeling productivity and a classical timing analysis to retrieve safe time values at the end of a modeling stage.

Part III

Outlook and Conclusion

Chapter 9

Future Work

In the context of the introduced approach to interactive timing analysis and its practical investigation with the example implementation, several entry points for future work opened up that are summarized in this chapter. Research topics that have emerged on the modeling tool side are described in Section 9.1, which includes an outlook on further modeling pragmatics aspects for time value display. Future work aspects on the analysis tool side are discussed in Section 9.2. Suggestions on how to employ the benefits of interactive timing analysis for compilation issues are made in Section 9.3.

9.1 Modeling Tool Side

Modeling pragmatics for interactive timing analysis constitutes a promising research topic for future work. The implemented approaches already facilitate working with large models. The hotspot highlighting described in Section 7.2 for example does not depend on the readability of timing values and thus is perceivable in large models without zooming. Furthermore, according to our user study evaluation described in Section 8.4, it seems to have a good potential for user guidance in timing related model revisions.

Readability is helped also for the detailed time values by the implemented focus-and-context timing hotspot expansion view explained in Section 7.8.1. However, to handle large models can still involve a lot of browsing with different zoom levels, as the displayed numbers get smaller with the display of their region in a large context. To solve this, it might be interesting to integrate the timing labels as a layout element of its own right instead of treating it as a parallel to the region label. This would enable to include the timing information as an element with constraints like a fixed

or minimum size and to vary its placement so that space can be used in an optimal way.

Even more promising seems an adaption of the model view to show different grades of detail on different zoom levels. For example for a model zoomed out for an overview, the region contents might be greyed out and overlayed with a big timing value display. As soon as the modeler zooms in on a timing hotspot, the detailed contents could be shown together with a smaller display of the timing label for precise revision. This approach demands involved investigations on modeling pragmatics to develop a general approach to customizing the model display in relation to zoom levels. This topic is already under research in the KIELER project. The corresponding results could be leveraged for the specialized employment for time value display.

Additionally, it would be of interest to practically investigate how combinations of analysis tools with different specializations could best support the modeling process. As explained in Section 6.7 and Section 8.4, it might be feasible to use a fast timing analysis optimized for the usecase of hotspot highlighting for interactive timing analysis, but employ a classic timing analysis tool for safe time values at the end of the modeling step and also for the in-advance-calculation of external function timing values. This is facilitated by the separation of concerns between the tick function and called function that is part of the interface concept.

Concerning the example implementation, a tighter support of deep local time values could be achieved by an implementation of the proposed expansion of the interactive timing interface for the aggregation of local time values, see Section 6.5. On the modeling tool side this would mean to generate timing requests for dispersed code parts in form of lists of TPP pairs as detailed in Section 6.10.3.

9.2 The Analysis Tool Side

On the analysis tool side it is left to future work to implement a scalable timing analysis tool that is capable of state-based analysis for arbitrary code parts specified by TPPs. The implementation of the interface expansion for

local time values proposed in Section 6.10.3 on the timing analysis tool side is also a future work topic for the practical investigation of tight local timing value aggregation.

Furthermore, it could be investigated by which means existing frameworks, for example the OTAWA tool[1], can be adapted to analysis of specified code parts instead of functions. First investigation steps in this direction have been undertaken by Banerjee [Ban12] concerning the usage of dummy functions, the analysis of sub-CFG and the usage of basic block information from a complete analysis to aggregate timing values for specified code parts.

One of the most interesting issues for future work concerns the preservation of the information on code part specification communicated with the TPPs pairs. The representation of TPPs as assembly labels implemented by the experimental timing analysis tool in the example implementation has the disadvantage of establishing compiler optimization barriers that limit the moving of instructions, which on general (especially non-PRET) architectures can lead to overestimations when compared with timing values for completely optimized code. This is further explained in Section 7.5. A solution could for example be to develop mappings of finer granularity between assembly instructions and code lines as endeavoured for example by Banerjee [Ban12]. Such, we could possibly establish and leverage a mapping from code parts marked by TPP to code lines to representing assembler instructions. This might make it possible to aggregate the timing contributions of these instructions, even when they are dispersed by compiler optimization. Corresponding investigations in the context of low level compilation which are concerned with the tracking of code parts that are not organized as functions can help to facilitate detailed timing analysis for state-based systems in general.

[1] `www.otawa.fr`

9.3 Additional Applications from the Compiler Perspective

The fast retrieval of timing information on code parts specified by TPP is not only of interest for user applications for modeling support. Also the compiler could leverage the connection to an interactive timing analysis tool, especially in the context of compilation for parallel execution.

A first use case concerns the allocation of code blocks to different hardware threads for parallel execution. As shortly explored by Weiß [Wei15], the calculation of WCET timing values for code parts could be used to calcuate a mapping of program segments to parallel threads that optimizes the overall WCET of the program. The interactive timing analysis could be used in combination with a datastructure that represents the program with a focus on its potentially parallel parts, like the SCPDG introduced by Weiß [Wei15]. The respective program parts can be annotated with their expected WCET contributions with the help of the interactive timing analysis. Based on this information code parts can be allocated to hardware threads, possibly in groups of code parts that are to be sequentially executed on their respective thread.

Another use case is the scheduling of threads on PRET architectures with the help of deadline instructions. As described in detail in Section 3.4, these architechtures provide timing instructions that allow the time triggered scheduling of threads, especially the *get_time* instruction that polls the platform clock and the *delay_until* instruction that delays a thread until a certain platform clock value is reached. This makes it possible to schedule the hardware threads that run in parallel on the processor and control inter-thread communication through shared local memories. Assume that we have an IASC program whose program parts have been mapped to three hardware threads on a PRET architecture such that each thread consists of a sequentialized code part. An advantage of the possibility of time-triggered scheduling is that the mapping does not have to avoid dependencies between the threads, as thread communication via shared memory is possible and the scheduling order can be controlled with the help of *delay_until* instructions. Figure 9.1 shows an abstract representation of an algorithm for

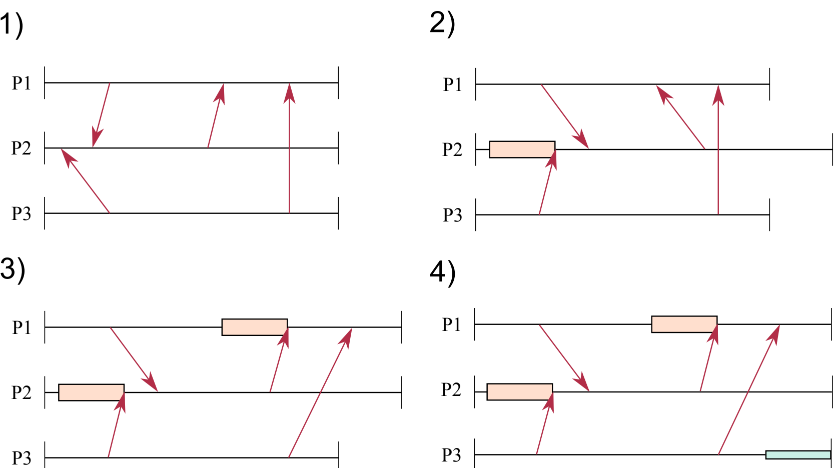


Figure 9.1. Schematic view of an algorithm for the time triggered scheduling of hardware threads on a PRET architecture. The timelines P1 to P3 correspond to three hardware threads and the red arrows signify thread communication dependencies that require that the program node at the arrow source has to be executed before the program node at the arrow end. Orange, thick rectangles denote time buffers inserted by the algorithm with *delay_until* instructions, the green thin rectangle signifies an adjustment of tick length.

the time triggered scheduling for one tick of the program. The goal of the algorithm is to guarantee for each dependency, denoted with red arrows in the drawing, that the variable access at the source of the dependency is executed in time before the variable access at the end of the dependency arrow. Thus, at the end of a pass of the algoritm, all arrows should point to the right. Thus the algorithm works with BCET values from the beginning of the tick up to the target variable accesses and WCET values from the beginning of the tick up to the source variable accesses. If the WCET that passes from the beginning of the tick up to the source of a dependency is longer than the BCET from the tick beginning to its target, the execution time of the thread with the target variable access is padded with the help of a *delay_until* before the access statement. The necessary padding amount

is calculated statically and gets its dynamic offset for the calculation of a concrete delay deadline time stamp with the help of a *get_time* instruction. The algorithm searches for dependency edges from beginning to end of the program. In the illustrated example, first the dependency shown leftmost in part 1) of the illustration is visited, it leads from P3 to P2. As its source might be executed after the target access, the thread P2 has to be padded with a delay, displayed in part 2 of the illustration, the padding denoted as a thick orange colored rectangle. While the dependency from P1 to P2 is automatically also corrected by this padding, the later dependency from P2 to P1 now points in the wrong direction in time and requires a padding as shown in part 3) of the illustration. Finally, the execution time of the tick is adapted to align all threads, shown in part 4) of the figure. Accordingly, the time triggered scheduling influences the timing behaviour of the tick and the resulting time values have to be checked for feasiblity.

Chapter 10

Conclusion

This thesis proposes an approach to interactive timing analysis that is based on a general formal interface, defined in Section 6.10, for the connection of a modeling tool to a timing analysis tool. The interface enables the exchangeability of the modeling tool as well as the timing analysis tool, so that arbitrary tool chains can be assembled from tools that implement the interface from one of the two sides. The approach is intended as a step towards an increased comparability between tools and also will hopefully facilitate benchmark sharing, as a modeling tool, especially one that is equipped with a corresponding benchmark suite, can be connected to different timing analysis tools for a direct comparison of their results. Also, the generated code of different modeling tools can be analyzed by the same modeling tool.

A distinct characteristic of the proposed approach is that it fits both state-based and dataflow-based modeling systems, which has been problematic so far, as traditional timing analysis granularity is on function level and thus better suited to represent dataflow-based systems, where elements are more easily mapped to functions in the generated code than for state-based systems, whose elements may be represented by arbitrary code parts. We have developed a new approach to formulate requests for timing values referring to any desired code part, which is specified by the placement of TPPs as markers in the code, described in Section 6.2.1.

This enables the retrieval of detailed timing values, whether state-based or otherwise. Any modeling tool that can trace model element representations during the compilation process and is able to map generated code parts to the model elements can connect to the interface and automatically poll timing values for a chosen granularity of model elements, which is described in Section 6.2.2.

Special aspects of the tool communication and the processing of detailed time values are discussed in this thesis, like the semantics of TPPs in parallel structures and loops, Section 6.4, the aggregation of time values in hierarchical models, Section 6.5, and a separation of concerns for the analysis of the central tick function and the analysis of called external functions. The latter unburdens the performance-critical request-response cycle of the interactive timing analysis from analysis parts that can be precomputed, detailed in Section 6.7.

Additionally the introduced approach allows the modeling tool to provide information on variable ranges for global variables and also on variables that represent system state, so that the modeling tool is able to rule out infeasible system states which is reflected in detecting special kinds of infeasible paths in the program as has been detailed in Section 6.9. This strenghtens its feasiblity for the analysis of state-based systems, but does not limit its application to them, as this additional information is optional.

Furthermore, this thesis introduces the time value categories of *flat* and *deep* timing values referring to hierarchical models and also of *fractional* and *local* time value based on whether time values represent a fraction of the critical path or a local maximum, as explained in Section 6.3.

The feasiblity of the introduced concepts is investigated with an example implementation with the KIELER SCCharts modeling tool and an experimental timing analysis tool discussed in Chapter 7. This implementation is also built to the main usecase of interactive timing analysis, the dynamic feedback of timing information to the tool user in order to help with timing related model revisions. Thus, besides detailed interactive timing value feedback for the exemplary model element granularity of regions, this thesis proposes hotspot highlighting, Section 7.2, as well as focus-and-context expansion of timing hotspots and the display of function timing values on mouse hover, Section 7.8.

The proposed interactive timing analysis approach is evaluated with the help of this example implementation. The thesis introduces an especially designed suite of test models as well as an evaluation on test models from the general KIELER benchmarks collection, detailed in Section 8.1. Also the implementation was tested with randomly generated models. Furthermore the thesis presents an aproach to the informal validation to the compilation

of extended language features of SCCharts in Section 8.3, in which expected semantical behaviour is mapped to an expected timing behaviour of the model. Finally, the presented approach was evaluated with a user study, which indicates that the interactive timing analysis can enhance modeling productivity in timing related model revisions, elaborated in Section 8.4.

The described investigations showed the usability of the interactive timing interface and suggest that this approach to interactive timing analysis can support modeling efficiency. They also exposed the prerequisites for a successful implementation of the interface from modeling tool and the analysis tool side. The technical challenges for the modeling tool lie in the tracing of model elements during compilation and automatic TPP placement, bookkeeping and time value aggregation as well as aspects of timing display and user interaction. A key enabler in this is the availability of tracing facilities. As long as a modeling tool can link the parts of the generated code to their origin model elements, a connection to the interface is possible. Challenges on analysis tool side are to provide time values for arbitrary code parts instead of functions, the detection of infeasible paths originating from infeasible state constellations for state-based systems and the preservation of code part specification during low level compilation and its consolidation with compiler optimization. Of these challenges, the handling of TPP information in the relation from host code to binary is one of the most interesting aspects for future work. An approach with assembly optimization barrieres as suggested in [FBH+16] is unproblematic on PRET architectures, and this thesis introduces methods to soften its effect on compiler optimization and consequently on the tightness of timing estimation for other processors. However, approaches that do not infringe optimization techniques as suggested in Section 9.2 can further enhance the timing analysis quality on general architectures.

Appendix A

Overview Test Models

In Table A.1, test models are listed with different characteristica. The table comprises hand designed models as well as models from the KIELER benchmark suite and models created with a random model generator. Note that the number of TPP is given excluding the two implicit program points *entry* and *exit*. Also, the number of nodes includes all superstates and the root state. The number of state variables is given with all outputs categorized as state variables, which can be optimized, when outputs have no read access, see Section 7.6.

Model Name	**Regions**	**Nodes**	**State Variables**	**TPP**
CircleWithCalls	1	5	6	1
CircleWithCalls2	1	5	6	1
Controller	7	28	22	11
FunPark2	40	151	76	63
MedicalAid	13	50	11	33
Feeder	7	22	3	12
MiniMultiWait	3	7	5	5
MultiWait	99	199	101	102
PlantController	11	33	11	25
Robot	3	9	6	6
Robot (improved)	3	8	6	6
Weaver	4	15	11	15
StrongAbort	2	5	4	3
WeakAbort	2	5	6	7
AbortMinimal	2	5	5	5
ResetAbortMinimal	2	3	3	4
Abort	2	6	3	5

Abort1	2	6	3	5
DeepHistory	3	8	14	24
ShallowHistory	3	8	12	20
ComplexFinal	2	5	8	6
ComplexFinal-ConcurrentRegions	3	8	11	10
CountDelay	2	8	16	6
Signal	1	4	10	8
Signal2	1	4	7	4
Suspend	3	8	11	14
Deferred	1	4	5	1
Entry	3	7	4	5
nested_broadcast	5	8	9	7
DependencyTest	2	7	8	7
SCU_Monitor	12	58	57	41
Elevator	4	10	25	12
ABRO	4	8	7	16
Cabin	2	10	14	10
ABO	3	7	7	9
Hierarchy	4	12	11	4
DVDPlayer	4	11	39	17
Broadcast	3	6	9	6
GoodCycle	2	6	10	7
model0	8	24	17	12
model1	4	13	10	8
model2	3	18	18	5
model3	7	27	21	15
model4	4	19	20	20
model5	2	10	16	10
model6	1	5	11	1
model7	1	4	7	1
model8	4	16	16	17
model9	1	9	15	1
model10	2	12	12	4
model11	4	18	23	14

model12	3	11	19	14
model13	1	5	6	1
model14	4	13	10	6
model15	1	5	12	1
model16	4	15	13	8
model17	1	9	12	1
model18	1	7	7	1
model19	3	9	5	
model20	6	25	19	10
model21	24	85	68	41
model22	1	5	10	1
model23	2	9	9	3
model24	5	20	16	14
model25	4	18	15	6
model26	4	13	10	6
model27	6	25	20	19
model28	6	25	19	17
model29	4	16	15	7
model30	5	20	16	11
model31	10	35	23	23
model32	8	35	25	18
model33	2	12	14	5
model34	20	83	53	37
model35	16	59	41	48
model36	24	98	87	72
model37	2	10	13	3
model38	2	9	11	4
model39	5	20	16	10
model40	8	39	32	25
model41	19	71	53	50
model42	15	67	45	32
model43	10	41	43	25
model44	54	228	190	143
model45	10	36	39	34
model46	59	238	204	144

model47	48	208	184	140
model48	10	46	38	22
model49	49	205	127	109
model50	13	63	49	23

Table A.1. Characteristics and values for the test models. The given number of TPP includes only explicit TPP. This means that the two implicit TPP at the beginning and at the end of the tick function are not included. The number of nodes includes superstates, including the SCChart root state.

Appendix B

Example Timing Focus

This chapter shows a maximal effect of the timing related focus-and-context implementation introduced in Section 7.8.1. The model called *model44* is a model that was generated automatically as test model for the interactive timing analysis. Though it is a large model, it has a concentrated timing hotspot in its outermost region, as can be perceived from Figure B.1, though details are not readable. Due to the single timing hotspot, the modeler can profit significantly from the focus-and-context view that collapses all regions whose timing contribution is insignificant. The result is shown in Figure B.2.

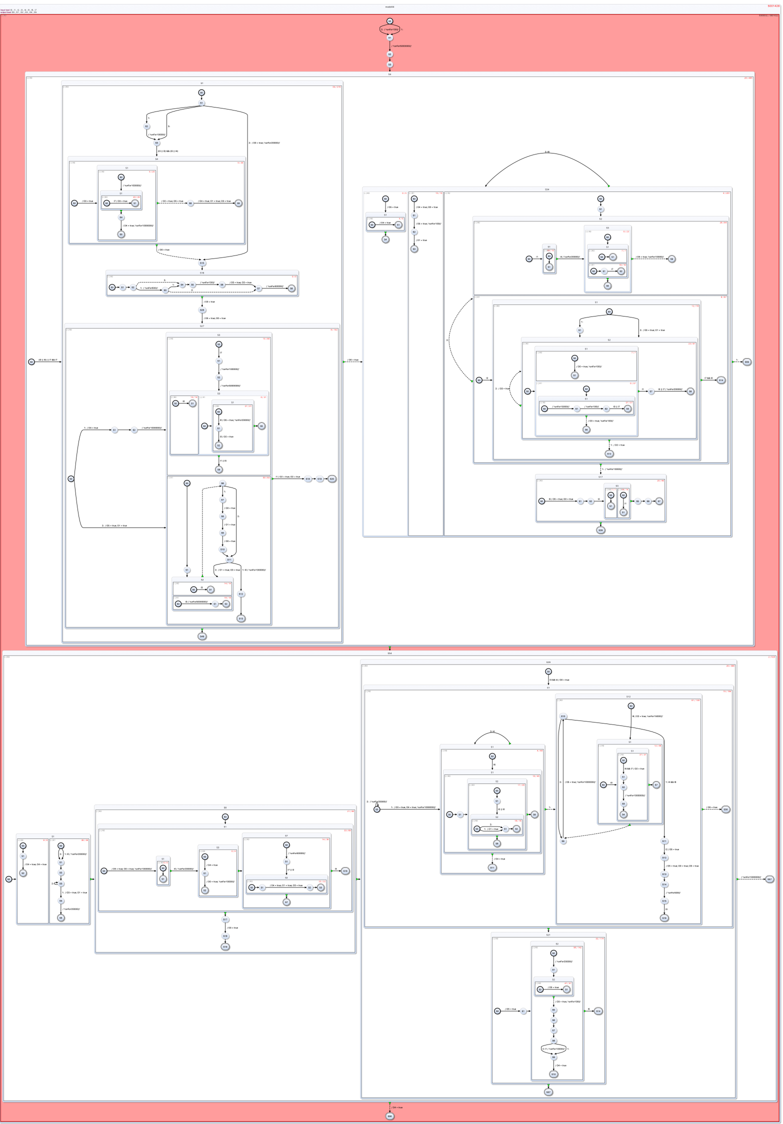


Figure B.1. The model *model44* is a randomly generated model. It has a concentrated timing hotspot in the outermost region, as can be seen with the help of the hotspot highlighting.

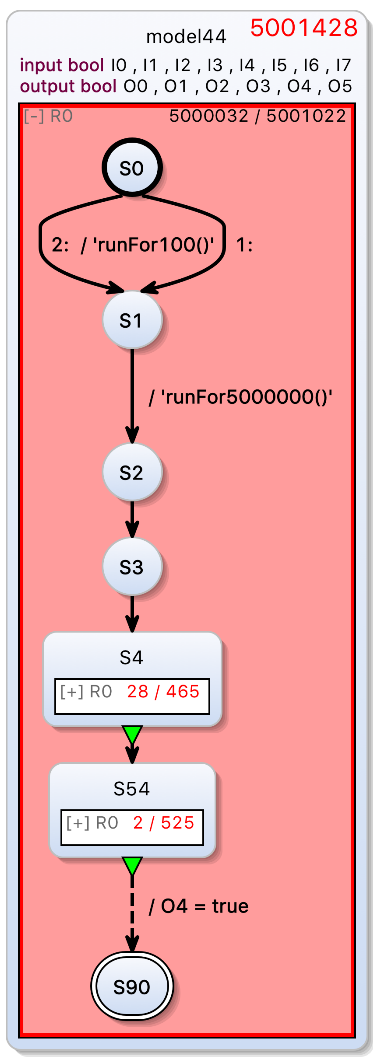


Figure B.2. The model *model44* in the timing hotspot focus view.

Bibliography

[ABS02] Oren Avissar, Rajeev Barua, and Dave Stewart. "An optimal memory allocation scheme for scratch-pad-based embedded systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 1.1 (2002), pp. 6–26.

[AMH+14a] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. "Grounding synchronous deterministic concurrency in sequential programming". In: *Proceedings of the 23rd European Symposium on Programming (ESOP '14), LNCS 8410*. Grenoble, France: Springer, Apr. 2014, pp. 229–248.

[AMH+14b] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. *Grounding synchronous deterministic concurrency in sequential programming*. Technical Report 94. ISSN 0937-3349. University of Bamberg, Faculty of Information Systems and Applied Computer Sciences, Aug. 2014.

[AMH+15a] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. *Denotational fixed-point semantics for constructive scheduling of synchronous concurrency*. Technical Report 1504. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2015.

[AMH+15b] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. *Denotational fixed-point semantics for constructive scheduling of synchronous concurrency*. Technical Report 96. ISSN 0937-3349. University of Bamberg, Faculty of Information Systems and Applied Computer Sciences, Apr. 2015.

[AMH+15c] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. "Denotational fixed-point semantics for

constructive scheduling of synchronous concurrency". In: *Acta Informatica, Special Issue on Combining Compositionality and Concurrency* 52.4 (2015), pp. 393–442.

[And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.

[And04] Charles André. "Computing SyncCharts reactions". In: *Electr. Notes Theor. Comput. Sci.* 88 (2004), pp. 3–19.

[And96a] Charles André. "Representation and analysis of reactive behaviors: A synchronous approach". In: *Computational Engineering in Systems Applications (CESA)*. Lille, France: IEEE-SMC, July 1996, pp. 19–29.

[And96b] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.

[App98] Andrew W. Appel. "SSA is functional programming". In: *SIGPLAN Not.* 33.4 (Apr. 1998), pp. 17–20. ISSN: 0362-1340.

[ARG10] Sidharta Andalam, Partha S. Roop, and Alain Girault. "Deterministic, predictable and light-weight multithreading using PRET-C". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*. Dresden, Germany, 2010, pp. 1653–1656.

[ARG11] Sidharta Andalam, Partha S. Roop, and Alain Girault. "Pruning infeasible paths for tight WCRT analysis of synchronous programs". In: *Design, Automation and Test in Europe (DATE'11)*. Grenoble, France, Mar. 2011, pp. 204–209.

[Ban12] Subarno Banerjee. *Timing analysis for the Precision Timed ARM processor*. Technical Report 1212. ISSN 2192-6247. Kiel University, Department of Computer Science, June 2012.

[Bay09] Özgün Bayramoglu. "The KIELER textual editing framework". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.

[BC84] Gérard Berry and Laurent Cosserat. “The ESTEREL Synchronous Programming Language and its Mathematical Semantics”. In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. “The Synchronous Languages Twelve Years Later”. In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[BCO+92] F. L. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronisation and linearity*. John Wiley & Sons, 1992.

[BCP+01] Valerie Bertin, Eric Closse, Marc Poize, Jacques Pulou, Joseph Sifakis, Paul Venier, Daniel Weil, and Sergio Yovine. “Taxys= esterel+ kronos. a tool for verifying real-time properties of embedded systems”. In: *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*. Vol. 3. IEEE. 2001, pp. 2875–2880.

[BCR+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. “Otawa: an open toolbox for adaptive wcet analysis”. In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2010, pp. 35–46.

[BDM+07] Guillem Bernat, Robert Davis, Nick Merriam, John Tuffen, Andrew Gardner, Michael Bennett, and Dean Armstrong. “Identifying opportunities for worst-case execution time reduction in an avionics system”. In: *Ada User Journal* 28.3 (2007), pp. 189–195.

[Ber00] Gérard Berry. “The foundations of Esterel”. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.

[Ber02] Gérard Berry. *The constructive semantics of pure Esterel*. Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France: Draft Book, Version 3.0, Dec. 2002.

[Ber07] Gerard Berry. "Circuit design and verication with esterel v7 and esterel studio". In: *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*. IEEE. 2007, pp. 133–136.

[Ber93] Gérard Berry. "Preemption in concurrent systems". In: *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1993, pp. 72–93. ISBN: 3-540-57529-4.

[BL90] Albert Benveniste and Paul Le Guernic. "Hybrid dynamical systems theory and the signal language". In: *IEEE transactions on Automatic Control* 35.5 (1990), pp. 535–546.

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. "Synchronous programming with events and relations: the signal language and its semantics". In: *Sci. Comput. Program.* 16.2 (Sept. 1991), pp. 103–149. ISSN: 0167-6423.

[Bou98] Frédéric Boussinot. *SugarCubes implementation of causality*. Research Report RR-3487. INRIA, Sept. 1998.

[BS01] Gérard Berry and Ellen Sentovich. "Multiclock Esterel". In: *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. London, UK: Springer-Verlag, 2001, pp. 110–125.

[BS91] Frédéric Boussinot and Robert de Simone. "The ESTEREL language. another look at real time programming". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1293–1304.

[BTH08] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. "Worst case reaction time analysis of concurrent reactive programs". In: *Electronic Notes in Theoretical Computer Science* 203.4 (June 2008). Proceedings of the International Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P '07), March 2007, Braga, Portugal, pp. 65–79. DOI: 10.1016/j.entcs.2008.05.011.

[Bus16] Jonas Busse. "SCCharts modeling for embedded systems with limited resources". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbus-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CC92] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks". In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[CP01] Antoine Colin and Isabelle Puaut. "A modular and retargetable framework for tree-based wcet analysis". In: *Real-Time Systems, 13th Euromicro Conference on, 2001.* IEEE. 2001, pp. 37–44.

[CPH+87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. "Lustre: a declarative language for programming synchronous systems". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*. Munich, Germany: ACM, 1987, pp. 178–188.

[CPP+02] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. "SAXO-RT: Interpreting Esterel semantic on a sequential execution structure". In: *Electronic Notes in Theoretical Computer Science*. Ed. by Florence Maraninchi, Alain

Girault, and Eric Rutten. http://www.elsevier.com/gej-ng/31/29/23/117/53/34/65.5.010.pdf. Elsevier, July 2002.

[Dud12] Björn Duderstadt. "A statechart dialect with sequential constructiveness". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/bdu-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2012.

[Dum59] Michael Dummett. "A propositional calculus with denumerable matrix". In: *The Journal of Symbolic Logic* 24.2 (1959), pp. 97–106.

[EL07] S. A. Edwards and E. A. Lee. "The case for the Precision Timed (PRET) machine". In: *Proceedings of the 44th Design Automation Conference*. San Diego, CA, USA, June 2007, pp. 264–265.

[FBH+16] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. "Time for reactive system modeling: interactive timing analysis with hotspot highlighting". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: ACM, 2016, pp. 289–298. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997467.

[FBH15] Insa Fuhrmann, David Broman, and Reinhard von Hanxleden. *Interactive timing analysis for designing reactive systems*. Presentation at the 22nd International Open Workshop on Synchronous Programming (SYNCHRON '15), Kiel, Germany. Dec. 2015.

[FBS+14a] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. "Towards interactive timing analysis for designing reactive systems". In: *Reconciling Performance and Predictability (RePP '14), satellite event of ETAPS '14*. Apr. 2014.

[FBS+14b] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. *Towards interactive timing analysis for designing reactive systems*. Tech. rep. UCB/EECS-2014-26. EECS Depart-

ment, University of California, Berkeley, Apr. 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-26.html.

[FH09a] Hauke Fuhrmann and Reinhard von Hanxleden. *Enhancing graphical model-based system design—an avionics case study*. Technical Report 0901. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Jan. 2009.

[FH09b] Hauke Fuhrmann and Reinhard von Hanxleden. *Exploring modeling pragmatics with ptolemy and kieler*. Presentation at the Eighth Biennial Ptolemy Miniconference, Berkeley, CA, USA. 16 10 2009.

[FH09c] Hauke Fuhrmann and Reinhard von Hanxleden. *On the pragmatics of model-based design*. Technical Report 0913. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming graphical modeling". In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*. Vol. 6394. LNCS. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[FKR+05] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. "The aerospace demonstrator of DECOS". In: *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems (ITSC '05)*. Vienna, Austria, Sept. 2005, pp. 19–24. ISBN: 0-7803-9215-9.

[FKR+06] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. "Model-based system design of time-triggered architectures—an avionics case study". In: *25th Digital Avionics Systems Conference (DASC '06)*. Portland, OR, USA, Oct. 2006.

[FL10] Heiko Falk and Paul Lokuciejewski. "A compiler framework for the reduction of worst-case execution times". In: *Real-Time Systems* 46.2 (2010), pp. 251–300.

[Fli16] Niclas Flieger. “Comparison of compilation approaches in KIELER”. Master thesis. Kiel University, Department of Computer Science, Apr. 2016.

[Fuh11] Hauke Fuhrmann. “On the pragmatics of graphical modeling”. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[FW99] Christian Ferdinand and Reinhard Wilhelm. “Efficient and precise cache behavior prediction for real-time systems”. In: *Real-Time Systems* 17.2 (Nov. 1999), pp. 131–181. ISSN: 1573-1383. DOI: 10.1023/A:1008186323068.

[GES+06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. “Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution”. In: *Proceedings of the IEEE International Real-Time Systems Symposium*. IEEE, 2006.

[GGB+91] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. “Programming real time applications with SIGNAL”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1321–1336.

[Gri16] Lena Grimm. “Debugging SCCharts”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.

[HAM+99] Christopher A Healy, Robert D Arnold, Frank Mueller, David B Whalley, and Marion G Harmon. “Bounding pipeline and instruction cache performance”. In: *IEEE Transactions on Computers* 48.1 (1999), pp. 53–70.

[Han09] Reinhard von Hanxleden. “SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency”. In: *Proc. Int’l Conference on Embedded Software (EMSOFT ’09)*. Grenoble, France: ACM, Oct. 2009, pp. 225–234.

[Han10] Sören Hansen. “Configuration and automated execution in the kieler execution manager”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/soh-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2010.

[Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data-flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for safety-critical applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[Hed00] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HLT+03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. “The influence of processor architecture on the design and the results of wcet tools”. In: *Proceedings of the IEEE* 91.7 (2003), pp. 1038–1054.

[HMA+13a] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation". In: *Proc. Design, Automation and Test in Europe Conference (DATE '13)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.

[HMA+13b] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. Technical Report 1308. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013.

[HMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation". In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.

[HSK+12] Trevor Harmon, Martin Schoeberl, Raimund Kirner, Raymond Klefstad, KHK Kim, and Michael R Lowry. "Fast, interactive worst-case execution time analysis with back-annotation". In: *Industrial Informatics, IEEE Transactions on* 8.2 (2012), pp. 366–377.

[IE06] Nicholas Jun Hao Ip and Stephen A Edwards. "A processor extension for cycle-accurate real-time software". In: *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*. 2006, pp. 125–134.

[JKA+08] Lei Ju, Bach Khoa, Huynh Abhik, and Roychoudhury Samarjit Chakraborty. "Performance debugging of Esterel specifications". In: *In International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*. 2008.

[Joh13] Gunnar Johannsen. "Hardwaresynthese aus SCCharts". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Oct. 2013.

[KBC+14] Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava. "WCET-aware dynamic code management on scratchpads for software-managed multicores". In: *Proc. of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. 2014.

[KFM13] Jan C Kleinsorge, Heiko Falk, and Peter Marwedel. "Simple analysis of partial worst-case execution paths on general control flow graphs". In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press. 2013, p. 16.

[KHR+96] Lo Ko, Christopher Healy, Emily Ratliff, Robert Arnold, David Whalley, and Marion Harmon. "Supporting the specification and analysis of timing constraints". In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE. 1996, pp. 170–178.

[KLF+02] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. "Fully automatic worst-case execution time analysis for Matlab/Simulink models". In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. IEEE. 2002, pp. 31–40.

[KM02] Oliver Köth and Mark Minas. "Structure, abstraction, and direct manipulation in diagram editors". In: *Proceedings of the Second International Conference on Diagrammatic Representation and Inference (DIAGRAMS'02)*. Springer, 2002, pp. 290–304.

[KSR11] Matthew Kuo, Roopak Sinha, and Partha S. Roop. "Efficient WCRT analysis of synchronous programs using reachability". In: *Proceedings of the 48th Design Automation Conference, DAC'11, San Diego, California, USA, June 5-10, 2011*. 2011, pp. 480–485. DOI: 10.1145/2024724.2024837.

[LA94] Ying K. Leung and Mark D. Apperley. "A review and taxonomy of distortion-oriented presentation techniques". In: *ACM Transactions on Computer-Human Interaction* 1.2 (June 1994), pp. 126–160.

[Lee08] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363–369.

[Liu12] Isaac Liu. "Precision timed machines". PhD thesis. EECS Department, University of California, Berkeley, May 2012.

[LLB+05] Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. "An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis". In: *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '05)*. San Francisco, CA, USA: ACM Press, Sept. 2005, pp. 225–236. ISBN: 1-59593-149-X. DOI: 10.1145/1086297.1086327.

[LLK+08] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. "Predictable programming on a precision timed architecture". In: *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*. Atlanta, GA, USA, Oct. 2008.

[LM87] Edward A. Lee and David G. Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE*. Vol. 75. IEEE Computer Society Press, Sept. 1987, pp. 1235–1245.

[LM97] Y-TS Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.12 (1997), pp. 1477–1487.

[LRB+12] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance". In:

Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012). IEEE. 2012, pp. 87–93.

[LS01] George Logothetis and Klaus Schneider. "A new approach to the specification and verification of real-time systems". In: *Real-Time Systems, 13th Euromicro Conference on, 2001.* IEEE. 2001, pp. 171–180.

[LSM03a] G. Logothetis, K. Schneider, and C. Metzler. "Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems". In: *Forum on Design Languages (FDL)*. Frankfurt, Germany: Kluwer, 2003.

[LSM03b] George Logothetis, Klaus Schneider, and Christian Metzler. "Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs". In: *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE. 2003, pp. 256–264.

[Mar89] Florence Maraninchi. "Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra". In: *International Conference on Computer Aided Verification*. Springer. 1989, pp. 38–53.

[Mar91] Florence Maraninchi. "The Argos language: Graphical representation of automata and description of reactive systems". In: *IEEE Workshop on Visual Languages*. Oct. 1991.

[MBB+16] Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chak ra borty, and R Venkatesh. "Tic: a scalable model checking based approach to wcet estimation". In: *ACM SIGPLAN Notices*. Vol. 51. 5. ACM. 2016, pp. 72–81.

[MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. "Layout adjustment and the mental map". In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.

[MH14] Christian Motika and Reinhard von Hanxleden. “Light-weight Synchronous Java (SJL) — an approach for programming deterministic reactive systems with java”. In: *Journal of Computing, Special Issue on Software Technologies for Embedded and Ubiquitous Systems* 97.3 (2014), pp. 281–307. DOI: 10.1007/s00607-014-0416-7.

[MHT09] Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. “WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing”. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE ’09)*. Nice, France, Apr. 2009.

[MJ03] Benjamin Musial and Timothy Jacobs. “Application of focus + context to UML”. In: *APVis ’03: Proceedings of the Asia-Pacific symposium on Information visualisation*. Adelaide, Australia: Australian Computer Society, Inc., 2003, pp. 75–80. ISBN: 1-920682-03-1.

[Mot09] Christian Motika. “Semantics and execution of domain specific models—KlePto and an execution framework”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.

[Mot10] Christian Motika. *Executing synccharts with ptolemy*. Presentation at the 17th International Open Workshop on Synchronous Programming (SYNCHRON ’10), Frejus, France. Dec. 2010.

[Mot11] Christian Motika. *Interactive esterel to synccharts transformation for executing esterel with ptolemy*. Presentation at the 18th International Open Workshop on Synchronous Programming (SYNCHRON ’11), Dammarie-les-Lys, France. Dec. 2011.

[Mot17] Christian Motika. *Sccharts—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017.

[MR01] F. Maraninchi and Y. Rémond. "Argos: An automaton-based synchronous language". In: *Computer Languages* 27.27 (2001), pp. 61–92.

[MRB16] Michael Mendler, Partha S. Roop, and Bruno Bodin. "A novel WCET semantics of synchronous programs". In: *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*. 2016, pp. 195–210.

[MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. "Compiling SCCharts—A case-study on interactive model-based compilation". In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[MSQ17] Fanqi Meng, Xiaohong Su, and Zhaoyang Qu. "Interactive wcet prediction with warning for timeout risk". In: *International Journal of Pattern Recognition and Artificial Intelligence* 31.05 (2017), p. 1750012.

[PH00] Patrik Persson and Görel Hedin. "An interactive environment for real-time software development". In: *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*. IEEE. 2000, pp. 57–68.

[PH99] Patrik Persson and Görel Hedin. "Interactive execution time predictions using reference attributed grammars". In: *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*. 1999, pp. 173–184.

[PK89] Peter Puschner and Ch Koza. "Calculating the maximum execution time of real-time programs". In: *Real-time systems* 1.2 (1989), pp. 159–176.

[PS91] Amir Pnueli and M. Shalev. "What is in a step: on the semantics of Statecharts". In: *Proc. Int. Conf. on Theoretical Aspects of Computer Software (TACS'91)*. London, UK: Springer, 1991, pp. 244–264.

[RAV+09] Partha S Roop, Sidharta Andalam, Reinhard Von Hanxleden, Simon Yuan, and Claus Traulsen. “Tight wcrt analysis of synchronous c programs”. In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2009, pp. 205–214.

[Ray08] Pascal Raymond. “Synchronous program verification with lustre/lesar”. In: *Modeling and Verification of Real-Time Systems* (2008), p. 7.

[RLP+11] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. “Pret dram controller: bank privatization for predictability and temporal isolation”. In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*. IEEE. 2011, pp. 99–108.

[RMP+13] P. Raymond, C. Maiza, C. Parent-Vigouroux, and F. Carrier. “Timing analysis enhancement for synchronous programs”. In: *Real-time Networks and Systems (RTNS'13)*. 2013, pp. 141–150.

[RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2016, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[RSS+13] Ulf Rüegg, Christian Schneider, Christoph Daniel Schulze, Miro Spönemann, Christian Motika, and Reinhard von Hanxleden. *Light-weight synthesis of Ptolemy diagrams with KIELER*. Presentation at the Tenth Biennial Ptolemy Miniconference, Berkeley, CA, USA. Nov. 2013.

[SB92] Manojit Sarkar and Marc H. Brown. "Graphical fisheye views of graphs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1992, pp. 83–91. ISBN: 0-89791-513-5. DOI: http://doi.acm.org/10.1145/142750.142763.

[SBT96] Thomas R. Shiple, Gérard Berry, and Hervé Touati. "Constructive Analysis of Cyclic Circuits". In: *Proc. European Design and Test Conference (ED&TC'96), Paris, France*. Paris, France: IEEE Computer Society Press, Mar. 1996, pp. 328–333.

[Sch01a] Klaus Schneider. "A verified hardware synthesis of esterel programs". In: *Architecture and Design of Distributed Embedded Systems*. Springer, 2001, pp. 205–214.

[Sch01b] Klaus Schneider. "Embedding imperative synchronous languages in interactive theorem provers". In: *Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on*. IEEE. 2001, pp. 143–154.

[Sch04] Martin Schoeberl. "A time predictable instruction cache for a java processor". In: *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*. Springer. 2004, pp. 371–382.

[Sch05] Martin Schoeberl. "Design and implementation of an efficient stack machine". In: *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE. 2005, 8–pp.

[Sch06] Martin Schoeberl. "A time predictable Java processor". In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE'06)*. Munich, Germany, Mar. 2006, pp. 800–805.

[Sch08] Martin Schoeberl. "A Java processor architecture for embedded real-time systems". In: *Journal of Systems Architecture (JSA)* 54.1–2 (2008), pp. 265–286.

[Sch09] Matthias Schmeling. "ThinKCharts—the thin KIELER SyncCharts editor". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf. Student Research Project. Kiel University, Department of Computer Science, Sept. 2009.

[Sch11a] Christian Schneider. “On integrating graphical and textual modeling”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf. Diploma Thesis. Kiel University, Department of Computer Science, Feb. 2011.

[Sch11b] Christoph Daniel Schulze. “Optimizing automatic layout for data flow diagrams”. Diploma Thesis. Kiel University, Department of Computer Science, July 2011.

[Sch14] Alexander Schulz-Rosengarten. “Framework zum Tracing von EMF-Modelltransformationen”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2014.

[SFH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. *Automatic layout of data flow diagrams in KIELER and Ptolemy II*. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.

[Sha89] Alan C. Shaw. “Reasoning about time in higher-level language software”. In: *IEEE Transactions on Software Engineering* 15.7 (1989), pp. 875–889.

[SMH15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “A data-flow approach for compiling the sequentially constructive language (SCL)”. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörtschach, Austria, May 2015.

[Smy13] Steven Smyth. “Code generation for sequential constructiveness”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, July 2013.

[Som16] Dirk Sommerfeld. “Laufzeitmessung für SCCharts auf Lego Mindstorms”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dso-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Apr. 2016.

[SPL09] Martin Schoeberl, Hiren D Patel, and Edward A Lee. “Fun with a deadline instruction”. In: *JOP* 22 (2009), p. 8.

[Spö09] Miro Spönemann. "On the automatic layout of data flow diagrams". Diploma Thesis. Kiel University, Department of Computer Science, Mar. 2009.

[SPP+10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. "Worst-case execution time analysis for a Java processor". In: *Software: Practice and Experience* 40.6 (2010), pp. 507–542.

[SR12] Sanjit A. Seshia and Alexander Rakhlin. "Quantitative analysis of systems using game-theoretic learning". In: *ACM Transactions on Embedded Computing Systems* 11.S2 (Aug. 2012), 55:1–55:27. ISSN: 1539-9087.

[SSH12] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Transient view generation in Eclipse". In: *Proceedings of the First Workshop on Academics Modeling with Eclipse*. Kgs. Lyngby, Denmark, July 2012.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125.

[SW01] K. Schneider and M. Wenz. “A new method for compiling schizophrenic synchronous programs”. In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*. ACM. Atlanta, Georgia, USA, Nov. 2001, pp. 49–58.

[TAH11] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. “Compiling SyncCharts to Synchronous C”. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE '11)*. Grenoble, France: IEEE, Mar. 2011, pp. 563–566.

[TS04] Olivier Tardieu and Robert de Simone. “Curing schizophrenia by program rewriting in Esterel”. In: *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)*. San Diego, CA, USA, 2004.

[Uml15] Axel Umland. “Konzept zur Erweiterung von SCCharts um Datenfluss”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015.

[Wec15] Nis B. Wechselberg. “Model railway 4.0”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbw-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Mar. 2015.

[WEE+08] Reinhard Wilhelm et al. “The worst-case execution-time problem—overview of methods and survey of tools”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008).

[Wei15] Tibor Weiß. “Von nebenläufigkeit zur parallelität in SCCharts”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/twe-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Oct. 2015.

[Wil04] Reinhard Wilhelm. “Timing analysis and timing predictability.” In: *FMCO*. 2004, pp. 317–323. DOI: 10.1007/11561163.

[Wis06] Mirko Wischer. “Textuelle Darstellung und strukturbasiertes Editieren von Statecharts”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/miwi-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Feb. 2006.

[WRA13] Jia Jie Wang, Partha S. Roop, and Sidharta Andalam. “ILPc: A novel approach for scalable timing analysis of synchronous programs”. In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES’13)*. Montreal, QC, Canada, Sept. 2013, pp. 1–10.

[ZBS+14] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. “Flexpret: a processor platform for mixed-criticality systems”. In: *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS’14)*. Apr. 2014.