Language Design for Reactive Systems

On Modal Models, Time, and Object Orientation in Lingua Franca and SCCharts

M. Sc. Alexander Schulz-Rosengarten
geb. in Kiel

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2023

Kiel Computer Science Series (KCSS) 2024/1 dated 2024-01-24

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:

▷ Alexander Schulz-Rosengarten. *Language Design for Reactive Systems — On Modal Models, Time, and Object Orientation in Lingua Franca and SCCharts* Number 2024/1 in Kiel Computer Science Series. Department of Computer Science, 2024. Dissertation, Faculty of Engineering, Kiel University.

```
@book{SchulzRosengarten24,
  author    = {Alexander Schulz-Rosengarten},
  title     = {Language Design for Reactive Systems --- On Modal Models, Time, and
               Object Orientation in Lingua Franca and SCCharts},
  publisher = {Department of Computer Science},
  year      = {2024},
  number    = {2024/1},
  doi       = {10.21941/kcss/2024/1},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering, Kiel University.}
}
```

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 Christian-Albrechts-Universität zu Kiel Kiel, Deutschland
2. Gutachter:	Prof. Dr. Edward A. Lee University of California, Berkeley Berkeley, USA
3. Gutachter:	Prof. Dr. Michael Mendler Otto-Friedrich-Universität Bamberg Bamberg, Deutschland

Datum der mündlichen Prüfung: 24.11.2023

Zusammenfassung

Reaktive Systeme spielen eine wichtige Rolle im Bereich der eingebetteten Systeme. Sie interagieren kontinuierlich mit ihrer Umgebung, verarbeiten nebenläufige Vorgänge und sollten im Allgemeinen deterministisches Verhalten aufweisen, um auch sicherheitskritische Anwendungen zu ermöglichen. In solch einem Kontext ist das Sprachdesign ein wichtiger Aspekt, denn sorgfältig gestaltete Sprachkonstrukte können dabei helfen komplexe Herausforderungen dieser Domäne anzugehen. Dies zeigen beispielsweise die verschieden Nebenläufigkeitsmodelle, die es erlauben den klassischen Fallstricken bei der Nutzung von Threads zu entgehen.

Heutzutage gibt es viele verschiedene Sprachen in diesem Kontext. Häufig zeichnen sie sich dadurch aus, dass sie einzigartige Charakteristika aufweisen, die sie für spezifische Anwendungsfälle besonders geeignet machen. Diese Arbeit beschäftigt sich mit zwei solcher Sprachen, der Aktor-orientierten polyglotten Koordinationssprache *Lingua Franca* und dem synchronen Statecharts-Dialekt *SCCharts*. Während die beiden Sprachen verschiedene Ansätze verfolgen, um eine reaktive Modellierung zu ermöglichen, weisen doch beide Gemeinsamkeiten in ihrer Semantik auf und ergänzen sich in ihren Designprinzipien.

Diese Arbeit betrachtet Schlüsselaspekte des Sprachdesigns für reaktive Systeme im Kontext dieser beiden Sprachen. Für drei relevante Konzepte werden dabei schlanke und minimalinvasive Spracherweiterungen entworfen und evaluiert. Besonderes Augenmerk liegt darauf diese neuen Konzepte an die fundamentalen Prinzipien der zugrundeliegenden Sprachen anzupassen. Konkret wird Lingua Franca um die Möglichkeit erweitert *modusabhängiges Verhalten* zu modellieren, während SCCharts eine *Timed Automata* Notation mit *dynamischen Ticks* zur effizienten Ausführung erhält und um Konstrukte zur *objektorientierten Modellierung* erweitert wird.

Abstract

Reactive systems play a crucial role in the embedded domain. They continuously interact with their environment, handle concurrent operations, and are commonly expected to provide deterministic behavior to enable application in safety-critical systems. In this context, language design is a key aspect, since carefully tailored language constructs can aid in addressing the challenges faced in this domain, as illustrated by the various concurrency models that prevent the known pitfalls of regular threads.

Today, many languages exist in this domain and often provide unique characteristics that make them specifically fit for certain use cases. This thesis evolves around two distinctive languages: the actor-oriented polyglot coordination language *Lingua Franca* and the synchronous statechart dialect *SCCharts*. While they take different approaches in providing reactive modeling capabilities, they share clear similarities in their semantics and complement each other in design principles.

This thesis analyzes and compares key design aspects in the context of these two languages. For three particularly relevant concepts, it provides and evaluates lean and seamless language extensions that are carefully aligned with the fundamental principles of the underlying language. Specifically, Lingua Franca is extended toward coordinating *modal behavior*, while SCCharts receives a *timed automaton* notation with an efficient execution model using *dynamic ticks* and an extension toward the *object-oriented modeling* paradigm.

Contents

1 Introduction **1**
1.1 The Furuta Pendulum Example . . . 3
1.1.1 Lingua Franca . . . 4
1.1.2 SCCharts . . . 5
1.2 Design Opportunities and Considerations . . . 8
1.3 Contributions and Publications . . . 9
1.3.1 Related Publications and Advised Theses . . . 12
1.4 Outline . . . 13

2 Design Principles of Lingua Franca and SCCharts **15**
2.1 Synchronous Languages . . . 17
2.1.1 Different Language Variants . . . 18
2.2 Lingua Franca . . . 19
2.2.1 Runtime Environment . . . 24
2.2.2 Target Language Integration . . . 25
2.3 SCCharts . . . 25
2.3.1 Dataflow SCCharts . . . 32
2.3.2 Runtime Environment . . . 34
2.3.3 Target Language Integration . . . 36
2.4 The Furuta Pendulum Example in Detail . . . 38
2.4.1 Main Program . . . 38
2.4.2 Pendulum Sound . . . 40
2.4.3 Pendulum Controller . . . 43
2.5 Modeling Pragmatics . . . 46

I Lingua Franca 51

3 Modal Models 53
- 3.1 Related Work . . . 56
  - 3.1.1 Mode Extensions . . . 57
  - 3.1.2 Mode Extraction . . . 60
  - 3.1.3 Modal Augmentation . . . 61
- 3.2 The Modal Pendulum Controller . . . 62
- 3.3 Modal Reactors . . . 65
  - 3.3.1 Modes and Transitions . . . 66
  - 3.3.2 Local Time . . . 70
  - 3.3.3 Startup and Shutdown . . . 75
  - 3.3.4 Implementation . . . 77
- 3.4 Evaluation . . . 82
  - 3.4.1 New Modeling Opportunities . . . 83
  - 3.4.2 Feature Comparison with Statecharts . . . 86
  - 3.4.3 Modes as Mutations . . . 90
  - 3.4.4 Embedded SCCharts . . . 91

II SCCharts 97

4 Time 99
- 4.1 Related Work . . . 102
  - 4.1.1 Modeling with Time . . . 102
  - 4.1.2 Synchronous Languages . . . 105
- 4.2 Timed Automata in SCCharts . . . 107
  - 4.2.1 The Eager Semantics . . . 110
  - 4.2.2 Timed SCCharts . . . 112
  - 4.2.3 The Furuta Pendulum in Timed SCCharts . . . 119
  - 4.2.4 Multiclock SCCharts . . . 122
- 4.3 When to React? . . . 124
- 4.4 Dynamic Ticks in SCCharts . . . 131
  - 4.4.1 A Dynamic Tick Environment . . . 131
  - 4.4.2 Sleep Time Inference from Timed Automata . . . 135

4.4.3 Dealing with Physical Time . . . 138
4.5 Evaluation . . . 143
4.5.1 Sparse Pendulum Execution . . . 145
4.5.2 A Hard Real-Time Demonstrator for Dynamic Ticks . 151
4.5.3 Event-Based Designs . . . 158

5 Object Orientation **165**
5.1 Related Work . . . 169
5.1.1 Embedded Systems . . . 169
5.1.2 Synchronous Languages . . . 171
5.1.3 Statecharts . . . 174
5.2 General Discussion and Assessment . . . 177
5.3 Object Orientation in SCCharts . . . 185
5.3.1 Class Modeling . . . 186
5.3.2 Inheritance . . . 193
5.3.3 Type Parametrization and Subtyping . . . 198
5.3.4 Modelling Object-Oriented SCCharts . . . 200
5.4 Deterministic Objects . . . 202
5.4.1 Black-Box Scheduling . . . 204
5.4.2 Scheduling Directives . . . 208
5.4.3 Scheduling Policies . . . 210
5.5 Evaluation . . . 213
5.5.1 Assessment of the High-Level Transformations . . . 216
5.5.2 Modelling Signals as Classes . . . 219
5.5.3 On Methods and Signals . . . 223
5.5.4 System Design Aspects of an Object-Oriented Steam Boiler Controller in SCCharts . . . 226

6 Conclusions **235**
6.1 Summary of Results . . . 235
6.2 Future Work on Lingua Franca . . . 237
6.2.1 Extending the Implementation of Modes . . . 238
6.2.2 Formal Analysis of Modes . . . 239
6.2.3 Behavior Trees . . . 240
6.3 Future Work on SCCharts . . . 241

6.3.1 Distributed SCCharts using Lingua Franca 241
6.3.2 Multiclocked SCCharts 242
6.3.3 Sleeping Programs 242
6.3.4 Refining Object Orientation 243
6.3.5 Object-Oriented State-Based Code Generation 245
6.4 Closing Remarks 248

Acknowledgments **251**

Bibliography **253**

Glossary **285**

List of Figures **289**

List of Tables **295**

List of Code Listings **297**

List of Algorithms **299**

Chapter 1

Introduction

Reactive systems are characterized by their ongoing and time-sensitive interaction with their environment [HP85]. This is particularly present in the *embedded* and *cyber-physical* domain [Lee08]. Software development for reactive systems is challenging due to intrinsically parallel processes, the physical environment, and timing requirements.

In order to meet these demands, languages in this field provide *concurrency*. It enables the description of logically concurrent process in the program and, orthogonally, utilizes parallel and distributed execution [Hal93]. While reactive programming is also often utilized for creating scalable and loosely-coupled software for interactive systems [BCC+13; KHA17], these often rely on unconstrained concurrency, which is prone to race conditions and thus non-determinism [Lee06; LÍG+19]. For embedded reactive systems and especially in a *safety-critical* context, any uncertainty in the program's behavior is inadmissible, or at least undesirable.

Synchronous languages are specifically designed to create deterministic software for reactive systems [BCE+03; Hal93]. The fundamental idea is to discretize time into instants, at which a program conceptually executes in zero time. Then, inputs can be considered stable during logical program execution and time progresses only in the environment. This abstraction enables a sound semantics with deterministic concurrency. Section 2.1 will provide a more detailed introduction. The synchronous paradigm and execution model has proven itself well-suited for specifying, validating, and implementing software for embedded and real-time systems [BCE+03]. While there are other languages and concepts that also achieve determinism, the synchronous principle is the one guiding this thesis.

Another important factor in developing reactive systems is the language design itself. Here, *Model-Driven Engineering (MDE)* is a prime principle that facilitates the design and implementation of complex software systems. It finds wide-ranging use [WHR14] and provides various advantages, such as domain-specific abstraction, model checking, graphical notations, and automatic code generation. In this thesis, MDE is central to the choice and design of languages for reactive systems. However, this does not categorically exclude classical code-driven techniques. *Pragmatics-aware* modeling techniques [HLF+22; FH10] and frameworks for model-based grammars [EB10] enable combining many aspects of classical programming into a unified model-driven approach. For example, *transient views* [HLF+22; SSH13; FH10] enable graphical representations of textual models and *polyglot* designs integrate support for different target languages into a single modeling language, see Chapter 2.

Modeling languages all differ in syntax, semantics, and specialization. Yet, they all offer means to abstract and shape the program based on a mental image for particular aspects in the system. This has resulted in a variety of abstractions or "views," each corresponding to a different mental model. The wide range of available modeling tools and languages illustrates this development. The Unified Modeling Language (UML) [Obj11] standard with its many manifestations is only one example for this heterogeneity. For reactive systems, two major views have emerged, one that focuses on the flow of data and one for the flow of control [Hal93; STP05].

The *dataflow view* breaks down the program into smaller blocks with interconnections representing the streams of data flowing between them. These blocks process inputs, produce outputs, and can be considered mostly independent of each other, thereby presenting opportunities for parallelization or distribution.

In the complementary *control-flow view*, the model expresses the activation of certain execution units. For example, a Control-flow Graph (CFG) [All70] models potential execution paths in a program. This is very close to classical imperative programming or even machine instructions. While this reflects the steps and states of the computation itself, it is often more relevant for a developer to think in terms of the states of the system. For this view, finite state machines (FSMs) are commonly used. When

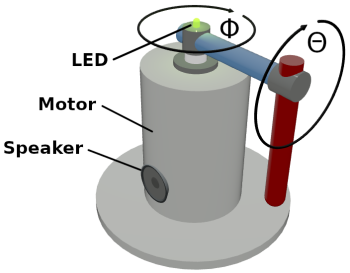


Figure 1.1. Schematic of a Furuta pendulum with an LED and speaker.

Harel and Pnueli initially introduced the notion of reactive systems [HP85], they also proposed statecharts to model such systems. States, transitions, hierarchical composition, and concurrency were introduced to address the challenges in reactive system design. In terms of language design for reactive systems, both views complement each other by emphasizing different aspects and are equally relevant to this thesis.

This thesis will evolve around two languages: *Lingua Franca (LF)* and *Sequentially Constructive Statecharts (SCCharts)*. Both are prime representatives of all the aspects discussed so far. They are modern model-based reactive languages, target embedded systems, and provide determinism using the synchronous principles. While LF features a dataflow view, SCCharts primarily use a statecharts notation providing a more control-flow-oriented view.

1.1 The Furuta Pendulum Example

To provide a first impression of both languages and to act as a motivating example, we will consider a controller implementation for an inverted pendulum designed by Furuta et al. [FYK92]. It is a classic control system problem, often used to teach feedback control. Figure 1.1 illustrates the basic setup. It consists of vertical shaft driven by motor, a fixed arm (blue) extending out at 90 degrees from the top of the shaft, and a pendulum (red) at the end of the arm. The goal is to balance the pendulum upright above

the arm. To achieve this, a controller usually has three phases; (1) it rotates the shaft to impart enough energy to the pendulum that it swings up, (2) it imposes a countermovement to catch the pendulum, and (3) it stabilizes the pendulum above the arm using minimal adjustments. Each of these steps requires a different control behavior, referred to as *SwingUp*, *Catch*, and *Stabilize* mode.

To illustrate additional timing aspects, the scenario is extended by an LED light and a speaker. The LED displays the phase in which the controller operates. *Off* during swinging up, *blinking* with a 30 msec period while catching, and constantly *on* when stabilizing. The speaker indicates how close the pendulum is to its upright position. From the lowest to the highest position it should play an ascending chromatic scale, specifically C_2–B_2 (approx. 65–123 Hz), by producing a square wave signal.

1.1.1 Lingua Franca

Lingua Franca is an actor-oriented polyglot coordination language [LMB+21] based on the deterministic *reactor* model [LÍG+19]. Reactors are inherently reactive, timed, concurrent, event-based, and represent encapsulated objects. LF focuses on efficient implementations and features a wide-ranging applicability, from the embedded to the distributed domain.

Figure 1.2 presents the LF program that controls the Furuta pendulum in an abstracted hardware environment. The program consists of *reactors*, represented by the rounded boxes with input and output ports. Reactors are concurrent and communicate via connections between their ports. In LF, computations are embedded in *reactions*, written in a target language, in this case C. In the graphical notation, reactions are represented by gray wedges, but the actual code is omitted to reflect only the coordination level.

The PendulumSound reactor handles the angle-dependent sound. Its reaction is triggered by an input at the angle port or the occurrence of an *action* (white triangle). Actions enable the scheduling of future events internal to the reactor. This action in particular is an effect of the reaction itself and is used to schedule the next occurrence of a duty cycle switch. The reaction calculates the current note based on the angle value and schedules actions to control the frequency of the square signal.

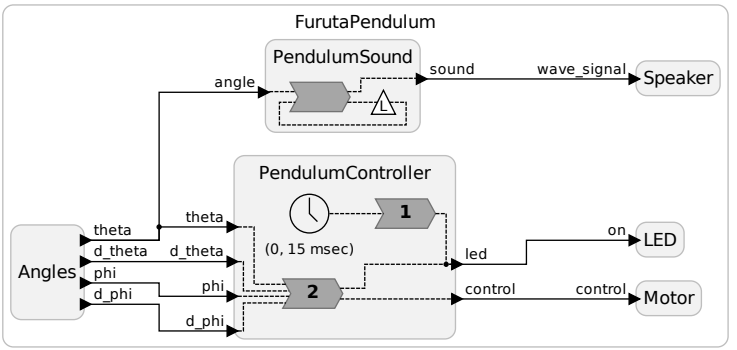


Figure 1.2. The pendulum control program in LF.

The PendulumController uses a *timer* (clock figure), with an initial offset of 0 and a period of 15 msec, to trigger reaction 1. This reaction sets the led output in an alternating pattern if operating in Catch mode. Reaction 2 computes the control response for the motor based on the angles at the two hinges and their velocity. The number in the reaction labels indicates the ordering, which is used to ensure output determinism. Here, reaction 2 is able to override the value on the led output produced by the first reaction to enforce a constant LED state for modes other than Catch.

The remaining reactors handle the interaction with the hardware. Their contents are hidden as it is of no further relevance to this example. Section 2.4 will present the given model in more detail, including the textual source code with the reaction bodies.

1.1.2 SCCharts

SCCharts are a synchronous statecharts dialect with a Sequentially Constructive (SC) semantics [HDM+14]. They augment Harel's classical statecharts [Har87] with various synchronous language constructs, while SC Model of Computation (MoC) combines sequential memory access with deterministic concurrency enabling a more intuitive imperative programming style than in classical synchronous languages, such as Esterel or Lustre.

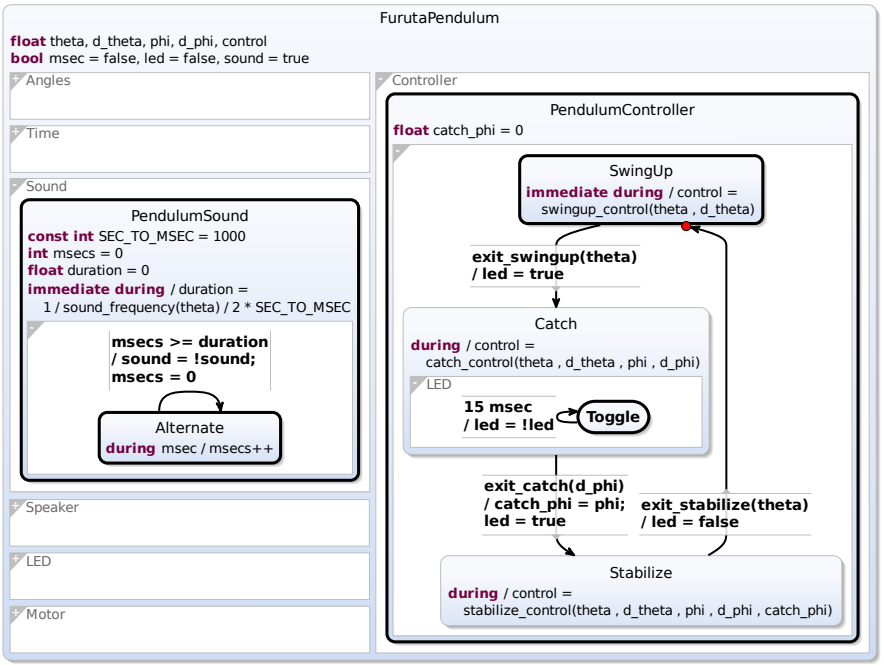


Figure 1.3. The pendulum control program in SCCharts.

SCCharts support multiple code generation strategies tailored to different use cases and target languages combined in a powerful and transparent model-based compiler [Smy21].

Figure 1.3 illustrates the SCChart that models the same behavior for the pendulum as in LF. The SCChart has seven concurrent *regions* (white inner boxes). These regions communicate via shared variables, declared at the top of the root state. The additional msec variable indicates the passage of one millisecond. It is not present in LF because it can rely on a built-in time model. In this example, msec is computed in the Time region by

communicating with the hardware.[1] This and the other regions for hardware access are again hidden (collapsed).

The PendulumSound state uses the milliseconds to control the frequency of the square signal. This state is the initial state of the enclosing region, indicated by the thicker border. It has a constant, two local variables, a *during action*, and an inner region. During actions executed the given effect as long as their state is active. If marked *immediate*, they already execute at the tick their state is entered; otherwise, they would only start at the subsequent tick. In this case, the external function sound_frequency is invoked to compute the frequency based on the current angle of the arm. The basic computations for the sound and motor control are factored out into external C functions to reduce the model size and reuse the same logic in SCCharts and LF. The frequency is converted into a half cycle duration in milliseconds and stored in duration. The inner region has a single state Alternate that increments the msecs counter every time a millisecond passed by using a during action. The self transition is activated if the msecs counter reaches the duration threshold and then toggles the sound signal and resets the counter. The transition *weakly preempts* the state, hence, the counting happens before the trigger test. Since the during action is not immediate, the millisecond cannot be counted twice.

In the PendulumController state, three inner states represent the different control modes. Each state uses different external functions to compute the control response and whether to exit this state. The LED is set as an effect of the transitions, and in the Catch state an additional region handles the toggling every 15 milliseconds. The 15 msec notation is a *count delay*; syntactic sugar for a similar counting infrastructure as in PendulumSound, but only for constant values. The red dot at the transition to SwingUp indicates a *deferred* entry into this state, suppressing the immediate execution of the during action. Hence, the control logic in each state will only activate in the tick after entering its state.[2]

[1] For an SCChart in general, it would be more common to have the hardware communication and time as dedicated inputs and outputs, see Section 2.3.2. However, the presented design focuses on highlighting the equivalence to the LF program.

[2] Again, there are alternative designs, but this one was chosen for equivalence with LF, especially in regard to modes presented in Chapter 3. This does not impair the modeling quality of the solution.

1.2 Design Opportunities and Considerations

Without further details on the implementation or semantics of LF and SCCharts, a comparison between the two pendulum models already reveals valuable insights on the different design principles and enables identifying opportunities for the improvement of both languages and for such language design in general.

Most apparent are the two notations that are used; the dataflow approach in LF, and SCCharts with its statecharts notation, as well as the different degrees of detail in the graphical notation. As discussed before, there is no clear preference for one or the other, and the granularity of the view is mostly a question of tool configuration. Yet, from a modeling perspective, LF cannot directly express the *modal behavior* of the pendulum controller, or the fact that the timer is only relevant to the Catch mode. Both aspects are explicit in the SCChart.

Another important aspect is the *handling of time*. In LF, time is a first-class citizen. This enables the use of a timer in the Controller reactor and the scheduling of actions in Sound with dynamic delays. On the other hand, the SCChart requires time as an explicit input; in this case single discrete milliseconds read from the hardware. This is a classical approach in synchronous languages that follows a multiform notion of time [Ber99]. In the face of the dynamic and real-valued nature of the angle-dependent delay and a per-tick time input, the counting of milliseconds is only a compromise between precision and workload. The counting requires a periodic execution of the SCChart at least every millisecond. In contrast to that, LF features a more resource-friendly sparse execution that facilitates a nanosecond precision (if supported by the hardware).

There are also notable differences in the way both languages express concurrency, provide internal communication channels, and integrate their target languages to provide basic instructions. These and other design aspects are discussed in more detail in Chapter 2. However, a more subtle aspect that is in the focus of this thesis is the fact that LF supports an *Object-Oriented (OO) design*. To a certain degree, this comes naturally to an actor-based language [Cap03; LLN09], but there is also great potential for a statecharts language in adopting OO concepts to improve its capabilities

in abstraction, genericity, reusability, and modularity. This is particularly relevant in the face of increasingly complex software systems, also in the safety-critical and embedded domain [Dvo09].

A straightforward solution to all these issues could be an attempt to create a new language that is a union of both. However, this is not the goal of this thesis. Both language have their fundamental design principles and semantics that makes them valuable and successful in their area. Instead, the goal is to address and investigate the discussed issues by creating simple and minimally invasive extensions that stay true to the core principles of the language. This especially includes a seamless integration into the textual and graphical syntax, a lean implementation, and conservative and robust semantics.

The previous comparison might give the impression that both languages are played against each other. However, the opposite is the case since they act as mutual motivation and inspiration due to the collaboration of both projects. Chronologically, some concepts were initially developed in parallel (cf. Section 1.3). Recent developments on efficient sparse execution [EH20; SIL+17] or time and object-like structures in imperative synchronous languages [SIL+17; GG18], also show the high relevance of the discussed topics to this domain. Yet, the consideration of modern MDE, deterministic concurrency, embedded targets, and high-level coordination, in combination with synchronous statecharts, or respectively reactor-oriented programming, represents a relevant contribution to the language design of reactive systems.

1.3 Contributions and Publications

This thesis is the first work analyzing the design of LF and SCCharts in direct comparison (Chapter 2). The main contributions, however, concern the improvement of both languages and are divided into three topics. Modal models in LF, Time in SCCharts, and Object Orientation (OO) for SCCharts. In each contribution the respective language is used as an environment to carefully design an integration of this aspect that extends the modeling capabilities for reactive systems.

Modal Models

Chapter 3 investigates the design of a modal coordination layer in a reactor-oriented environment. This chapter draws from and extends publications on modal reactors [SHL+23b; SHL+23c; SHL+23a]. Specifically, Chapter 3 presents

- a minimally invasive language extension of LF to express modal structures that embraces the reactor-oriented nature and black-box approach towards reaction code;
- an adaptation of a lean set of hierarchical composition capabilities and common transition types, reset and history, implementing modal behavior with reactors; and
- a semantics for modal behavior that introduces mode-local time and leverages the superdense time model to achieve deterministic behavior.

The adaption of the LF tooling to the modeling pragmatics of the KIELER project (see Section 2.5) is also part of my work, including the synthesis of LF diagrams. Yet, this is only covered implicitly and is not part of a dedicated publication or chapter in this thesis.

Time

Chapter 4 evolves around research questions on real-time modeling in a synchronous context with a focus on a flexible notation of time, efficient executions strategies, and practical arrangements for timer imperfections when dealing with physical time. The presented topics draw from and expand on publications titled “Time in SCCharts” [SHM+18; SHM+20]. The chapter covers

- a lean timed automaton notation for SCCharts that models time with real-valued clocks and can be expressed in a synchronous setting and only minimal requirements on the execution environment or the languages itself;
- a new language feature for periodically timed regions that enables modeling multiclocked systems;

- a detailed investigation of the suitability of different execution strategies in a timed setting; and
- a sparse execution environment that implements dynamic ticks in a synchronous real-time setting and requires only a minimal runtime infrastructure.

Object Orientation

Chapter 5 investigates the integration of the OO programming and design methodology into a pragmatics-aware synchronous statecharts modeling language. My work on OO is published with the title "Toward Object-oriented Modeling in SCCharts" [SSM19; SSM21]. This chapter draws from these results and extends them, consisting of

- a conservative extension of SCCharts that permits OO modeling under the principles of synchronous languages, providing modeling class-based data structures, capabilities for programming using methods, inheritance to improve reusability, and a proposal for type parameterization and subtyping to further facilitate abstraction;
- an integration of the new OO design features into the modeling and compilation tooling of KIELER; and
- mechanisms to ensure the determinism of host language objects under the shared memory concurrency of synchronous languages while retaining their encapsulation under a contract-based black-box scheduling approach.

All three concepts have been implemented and tested and are publicly available in the respective open-source projects of LF[3] and SCCharts[4].

[3] https://github.com/lf-lang/lingua-franca

[4] https://github.com/kieler/semantics

1.3.1 Related Publications and Advised Theses

There are various co-authored publications and student theses advised by me that relate to the topics discussed in this thesis and inspire and support some contributions.

- On the topic of modal models and hybrid modeling, there is work under the lead of Nis Wechselberg on "Augmenting State Models with Data Flow" [WSS+18]; a Master's thesis [Gri19] and subsequent publication [GSS+20; GSS+22] by Lena Grimm on the adaption of Lustre/SCADE dataflow into SCCharts (using the dataflow extension developed by Steven Smyth [Smy21]); and a Master's thesis [Luc20] and publication [LSH+21] by Daniel Lucas on "Extracting Mode Diagrams from Blech Code".

- For time in SCCharts, Andreas Boysen developed in his Master's thesis "An FPGA-based Demonstrator for Dynamic Ticks" that was also published [BSH20b; BSH20a; BSH20c], see also Section 4.5.2. The collaboration with the LF team also led to my participation in a publication on time in LF [LMS+20; LBM+23].

- In relation to OO stands the Bachelor's thesis of Gavin Lüdemann on "Modular Code Generation for SCCharts" [Lüd21] and a supportive role in the introduction of scheduling directives [SSH19; SSH18b] (see Section 5.4.2 and Section 4.2.2) under the lead of Steven Smyth (covered in [Smy21]).

- On MDE and modeling pragmatics, there are the Master's theses by Philip Eumann on "Model-Based Debugging" [Eum20], by Andreas Stange titled "Model Checking for SCCharts" [Sta19], and by Sören Domrös on moving MDE tooling into web technologies [Dom18]. Likewise, a supportive role in "Guidance in Model-based Compilations" [SSH18a] led by Steven Smyth (covered in [Smy21]) relates to this topic. Finally, I participated in the creation of "Pragmatics Twelve Years Later: A Report on Lingua Franca" [HLF+22].

1.4 Outline

After this introduction follows Chapter 2 providing a direct comparison of LF and SCCharts, including a brief introduction of their notations and conceptional foundations. At the heart of this thesis are two parts. Part I focuses on LF and the integration of modal models in Chapter 3, and Part II bundles the two contributions based on SCCharts. Chapter 4 presents the contribution on time and dynamic ticks, and Chapter 5 covers OO. Each of the three main chapters briefly covers related work in the given context and presents and evaluates its contribution. Finally, Chapter 6 concludes these topics and gives an outlook on future work.

Chapter 2

Design Principles of Lingua Franca and SCCharts

Both LF and SCCharts are languages that enable modeling embedded reactive systems and feature deterministic concurrency. They also put an emphasis on practicality when it comes to the design and implementation of the language itself, the generated code, and also the surrounding tooling. While there are many commonalities between the two, there are also differences, such as the statecharts and dataflow modeling principle, support for distributed execution, and integration of target languages.

Outline This chapter starts with a short description of the synchronous approach in Section 2.1, which is an important link between the semantics of LF and SCCharts. Afterwards, Section 2.2 and Section 2.3 discuss LF and SCCharts. They both start with a brief introduction of the syntax, structure, and semantics, highlighting differences and commonalities. Both sections also feature the respective perspective on the runtime environment and their approach to integrate target languages, as these become especially relevant in subsequent chapters. In anticipation, Table 2.1 gives a brief overview and comparison of some key aspects in LF and SCCharts that will be presented. Section 2.4 presents a detailed look into the source code and behavior of the pendulum implementations introduced in Section 1.1. This provides a more practical impression of the different modeling features and principles. Finally, Section 2.5 presents the modeling pragmatics and state-of-the-art pragmatic-aware tooling that influence the design and usability of LF and SCCharts.

Table 2.1. A brief overview and comparison of key aspects in LF and SCCharts. It covers the current state of the implementation, plus references to extensions made by this thesis.

Aspect	Lingua Franca	SCCharts
Notation	dataflow actors + *modes (Chapter 3*)	statecharts and dataflow
Modularity	OO	macro modules + *OO (Chapter 5)*
Reactions	time and event-driven	clock-driven + *dynamic ticks (Chapter 4)*
Synchrony	multiclocked synchronous	globally synchronous
Causality	static acyclic	static SC
Time	first-class citizen, logical and physical time	"just" input + *real time (Chapter 4)*
Target language integration	embedded black-box code	generic expression language and black-box host functions
Internal communication	events and state variables with locally restricted scopes	shared variables
External communication	direct or via physical actions	direct or via inputs & outputs
Execution	single-threaded, multi-threaded, and distributed	single-threaded
Runtime restructuring	limited mutations[1]	none
Compilation structure	monolithic	model-to-model
Artifact	standalone program	tick function
Supported languages	C, C++, Python, Typescript, and Rust	C, Java, and VHDL

[1]Generally supported by reactors but only partially implemented in the Typescript target.

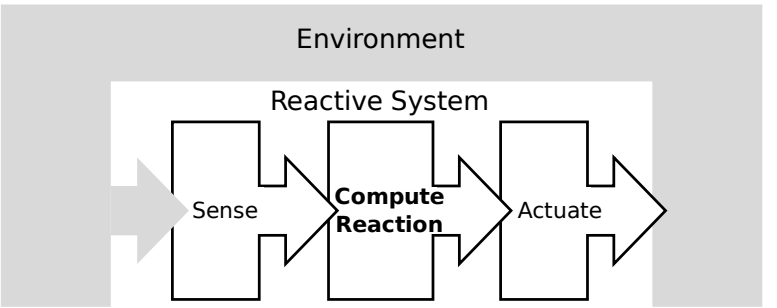


Figure 2.1. Abstract view on a reactive system embedded in its environment (based on [MHH13]).

2.1 Synchronous Languages

Concurrency is both a powerful programming principle and intrinsic to the execution model of embedded reactive systems. Synchronous languages are specifically designed to address the challenge of deterministic[2] concurrency [BCE+03; Hal93; STP05].

Figure 2.1 illustrates the conceptual setup that drives the synchronous principle. By definition, the reactive system is embedded in an environment [HP85]. From the synchronous design perspective the abstracted process inside the program is (1) reading inputs from sensors, (2) computing a reaction, and (3) writing outputs to actuators. To ensure that this lifecycle always has a deterministic outcome and is not exposed to race conditions in the presence of concurrency, synchronous languages traditionally apply two techniques; *synchrony* and *causality*.

The fundamental idea of the former is to discretize time into *instants* that conceptually run in zero time [BCE+03; Hal93; STP05]. Each reaction represents such an instant. Since the pace of reactions in synchronous lan-

[2]There is a distinction by Milner [Mil89] that separates determinacy from determinism. According to his definition, a computation is determinate if the same input sequence produces the same output sequence. In contrast to a deterministic one that additionally requires an identical *internal* behavior and scheduling. This thesis does not follow this distinction and uses both terms synonymously to refer to determinacy in Milner's sense. It considers observable determinacy w.r.t. the model level.

guages is often driven by a global clock (inspired by hardware design), the execution instants are also called *ticks*. As a consequence of this abstraction, outputs are generated at the same (logical) time as inputs are read. Thus, inputs can be considered stable during reaction computation and time progresses only in the environment. Synchronous programs classically consist only of finite reaction workload, implicitly or explicitly separated by some form of *pauses* that end the reaction and give the environment (conceptually) some time to adjust before continuing with the next reaction. For any concurrency expressed in the program, these pauses result in a barrier synchronization for all threads that keeps them in lockstep.

The second technique introduces causal reasoning for data accesses. The basic idea is that any read value should be definitely and deterministically defined before it is accessed, especially in the presence of multiple concurrent writers [Hal93; STP05; Ber00]. Hence, synchronous languages require any communication at each instant between concurrent threads to be regulated by synchronous *signals* or *channels*. These special-purpose shared memory structures are protected by an *(intra-instant) synchronization protocol*, which ensures a unique value per instant. As a consequence, the observable behavior of a program is that of a synchronous Mealy machine. This provides synchronous languages with a sound mathematical semantics. In practice, the compiler performs a static *causality analysis* that checks if the synchronization protocol can be satisfied for each memory reference. If a scheduling order can be found that adheres to these rules for any relevant state and input of the program, it is considered *constructive* [Ber00]. Non-constructive programs are rejected.

2.1.1 Different Language Variants

Since their emergence in the 80s, various variants of synchronous languages have been developed. They come in different programming styles and with varying semantics. Yet, the fundamental principles are always present, even if the specific MoC differs, for example in terms of synchronization protocols.

One of the most prominent and influential synchronous language is *Esterel* [Ber00]. Its constructive semantics are motivated by hardware circuit behavior and use multi-writer/multi-reader signals that follow a *write-before-*

read synchronization protocol. This results in a globally consistent state for each signal, similar to wires in a netlist. Esterel features an imperative coding style and allows interacting with its host language (here C) to facilitate practical application [PEB07]. The language supports various constructs, most of which are considered syntactical sugar that can be represented by a set of kernel language features to ease compilation.

The more recently developed synchronous languages Blech [GG18] and Céu [SIL+17] are heavily inspired by Esterel and improve the handling of time, working with data structures, and modularity. Quartz [Sch10] is another imperative synchronous language close to Esterel that focuses on hardware software co-design. SyncCharts [And03] is a synchronous statecharts dialect that draws its semantics from the equivalence to Esterel.

In the category of dataflow or declarative synchronous languages, *Lustre* [HCR+91] is the most notable one. Lustre programs consist of equation systems that use nodes as reusable operators and subsystems. Communication is handled via clocked channels that implement a synchronization protocol similar to Esterel. The main difference is that each channel must always provide a value (be written) in accordance with its clock, while Esterel also supports reaction to absence. A clock calculus ensures that channels in a Lustre program are well-formed.

The Safety-Critical Application Development Environment (SCADE) is a commercial variant of Lustre that supports graphical modeling [CPP17]. In SIGNAL [GGB+91] a multiclocked approach facilitates targeting distributed systems [GG10]. Zélus [BP13] is a hybrid synchronous language that extends discrete dataflow with ordinary differential equations. The heterogeneous modeling environment Ptolemy II features a synchronous reactive domain for modeling dataflow and coordinating state machines [EJL+03; Pto14].

2.2 Lingua Franca

Lingua Franca implements the concept and MoC of reactors [LÍG+19]. It is a dataflow-oriented polyglot coordination language that abstracts a target language into black-box containers (reactions) and coordinates their execution in a reactive, deterministic, timed, and concurrent manner. The paradigm

of reactor-oriented programming is inspired by many well-established principles, such as OO [Str87], actor-oriented design [Hew77], or event-driven programming [DZK+02].

This short overview can only provide a limited introduction into LF. A more comprehensive description is available in the respective publications [LÍG+19; Loh20; LMB+21] and the official documentation[3].

Composition LF programs consist of reactors. The entry point is a single *main* reactor. Each reactor can instantiate other reactors and create connections between input and output ports to establish communication channels. In addition to ports and reactor instances, reactors can contain reactions, state variables, timers, and actions, as Figure 1.2 already illustrated.

Object Orientation Reactors can be considered instantiable classes in an OO sense. They also offer *inheritance* to facilitate reusable designs. A reactor can extend another one, where all declaration of reactions, ports, timer, etc. in the super class are placed syntactically before locally defined ones. The natural ordering of LF reactions then automatically results in an overriding behavior. This form of inheritance is a bit more restrictive than in classical mainstream OO languages.

Like objects, reactors encapsulate state and behavior in the form of state variables and reactions in combination with timers and actions. All contents of a reactor, except ports, have a local scope; they are only visible inside the reactor or its subclasses. While objects classically interact via method calls, reactors receive events from ports, timers, and actions and handle them in reactions. Unlike methods, reactions are not invoked directly, but are executed once if triggered by the presence of an event. There is no recursion or multiple invocation per tick. In addition to reactions, LF also has classical methods. However, these cannot react to input events but instead are invoked from within reactions.

Causality Each reaction has a signature in the following form:

reaction(<*trigger**>) <*source**> -> <*effect**> {= <*body*> =}

[3] https://www.lf-lang.org/docs/

It defines optional lists of triggers (ports, actions, or timers that can trigger it), sources (ports that the reaction can read from when triggered), and effects (ports it may set or actions it may schedule). Scoping rules for the reaction body, enforced by the compiler, ensure that the data dependencies expressed in this signature are conservative. Hence, reaction signatures represent a causality interface for reactions [ZL08].

Reaction signatures and connections between ports or instantiated reactor can be turned into a dependency graph. If acyclic, it yields a partial order for all reactions that expresses all scheduling constraints that will result in a deterministic execution for each tick. This corresponds to the write-before-read protocol in synchronous languages. A causality problem is present if the graph is cyclic, e. g., a reaction's trigger or source depends on its own immediate effect. Figure 2.6b will illustrate an example of a model with a causality error.

The fact that the dependencies graph is valid independent of the actual code in the body enables to treat reactions as black-boxes. This property is the key factor in the polyglot nature of LF.

State variables are implicitly accessible in all reactions, and multiple reactions may set the same output port. Therefore, reactions within the same reactor always have a fixed scheduling order assigned to them, which is derived from their textual position in the code. The dependency graph also reflects these constraints to ensure determinism. In this regard, LF features sequentiality similar to the SC MoC.

Synchronicity The runtime mechanism of LF is event-driven, as only events trigger reactions. The event processing in reactors is synchronous and similar to synchronous languages. Events are tagged with their time of occurrence on a *logical timeline*. Execution progresses in ticks where all events of the current time are processed at once and time does not progress during execution. Consequently, outputs carry the same logical time tag as their inputs and may instantly trigger downstream reactions. The previously mentioned causality analysis ensures that no reaction executes before all the data sources that it depends on are known. While ticks can act as a barrier synchronization for concurrent reactors, primarily in a non-distributed

context, LF also facilitates scheduling regimes that enable reactions to span multiple ticks, as in logical execution time [KS12; HHK03].

Events can be checked for their presence at the current time and carry a value. In this respect they correspond to valued signals in synchronous languages.

Time While ports are terminals for the flow of events between reactors, actions and timers enable the creation of timed events inside reactors. Timers produce events with a predefined period and offset, whereas actions provide an interface to manually schedule future events as an effect of reactions.

Considering the previously discussed causality, a reaction that dependents on itself via an action, as in PendulumSound in Figure 1.2, could be considered a problematic cycle. Yet, this is not the case due to a mandatory delay upon scheduling. While the delay clearly must not be negative, it may be zero.

Tags in LF are pairs (t, m), where t is a time value and m a microstep index. Mircosteps separate subsequent ticks at the same logical time t. This implements the concept of *superdense time* [MP93].

Connections pass events instantaneously (within the same tick/microstep), but they can also be configured to impose a delay. Again, this can be zero, introducing a microstep delay when passing events between reactors.

Another important aspect of time in LF is the relation to *physical time* (wall clock time). This is also reflected in the difference between physical and logical actions. Logical actions create events from within the execution of reaction, hence, in sync with logical time, while physical actions can be scheduled from an asynchronous context, e. g., a spawned thread or interrupt service routine. Events from physical actions receive a tag based on current physical time. The execution engine ensures that logical time will not run ahead of physical time, such that there is no risk of out-of-order events. In practice, logical time will lag behind physical time, because it will not progress during execution. LF provides deadlines to detect and handle timing violations [LMS+20; LBM+23].

Concurrency Reactions are atomic execution units that are subject to a dependency graph. This transparently enables the parallelization of logically simultaneous reactions that are independent of each other, without risking

data races or deadlocks. LF's runtime implementation features different options for single- or multithreaded execution.[4]

Furthermore, LF programs can also be *federated*. If the main reactor is marked federated, this turns its inner reactors instances into federates that can be distributed onto different machines. Connections will be automatically set up as network channels. The semantics of reactors, especially the notion of logical time, will ensure determinism even in this distributed setting. Yet, the nature of this environment results in multiple timelines and a tradeoff between consistency and availability [LBL+23; LBL+21; LMS+20; LBM+23]. LF offers the choice between a centralized coordination that handles global synchronization of logical time or a decentralized control that relies on the local physical clocks and techniques from Programming Temporally Integrated Distributed Embedded Systems (PTIDES) [ZLL07]. The second mechanism requires explicit bounds on network latencies and clock synchronization errors, which may be violated in a practical deployment. However, LF can detect these situations and enables reactions to such faults, similar to deadlines.

The decentralized distributed aspect shows that LF is not a classical synchronous language, in the sense that it is not driven by a single global clock, but rather corresponds to multiclocked variants, such as SIGNAL [GG10] or multiclocked Esterel [BS01]. It also relates to the idea of a Globally Asynchronous Locally Synchronous System (GALS) [Cha84].

Mutations While static instantiation is the normal and most common way of composing reactors, the concept of reactors also supports dynamic runtime creation and destruction using *mutations* [LÍG+19]. This includes changing connections between reactors or reactions. By requiring the same static signatures for mutations, it is possible to decide at compile time on the soundness of runtime modifications to the reactor topology. In LF, mutations are currently only implemented for the Typescript target as an experimental feature.

[4]Currently, only the C, C++, and Rust targets provide support for multithreaded execution.

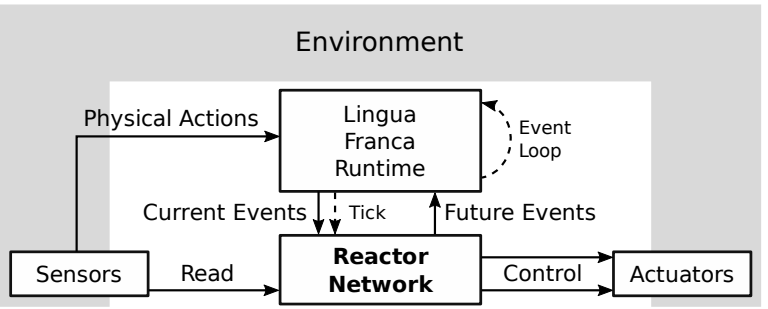


Figure 2.2. LF-specific reactive program components and their interaction with each other and the environment.

2.2.1 Runtime Environment

LF programs compile into standalone executables. In case of a federated program, multiple executables are generated that can be automatically deployed onto the different machines. An LF program consist of program-specific code, generated from the reactor network and its reactions, plus a generic runtime engine handling the event loop, scheduling of reactions, and progression of time. This event-driven approach and the fact that main reactors (as well as federated ones) cannot have input or outputs ports, defines the way LF programs interact with their environment.

Figure 2.2 presents a concretized version of the reactive system schematics in Figure 2.1. It shows the LF-specific components and their interaction with each other and the environment. The arrangement represents the LF perspective on a reactive system. Hence, the Sensors and Actuators are positioned mostly in the environment, as they represent hardware components and are only accessed via their respective application programming interface (API). The Sensors provide input data asynchronously by scheduling events via physical actions or read directly from within reactions. The latter can be used to implement a polling mechanism by triggering a reading reaction by a periodic timer. The LF Runtime manages events and time. Dashed lines in the schematic represent triggering/execution of components, while solid lines indicate passing data. The LF Runtime runs an internal Event

Loop that advances time until processable events are present. If present, it will start the execution—a Tick of the Reactor Network—passing the events with that tag. The execution of the reactions will control the Actuators and produce new internal events that need to be processed in the future.

2.2.2 Target Language Integration

LF aims for a polyglot language design by embedding code of any target language directly into its coordination language. Key enabler for this approach is the black-box treatment of reactions. Target code blocks ({= ... =}), such as reaction bodies, are neither parsed nor otherwise analyzed. Only the reaction signatures are used as conservative dependency interfaces, while it remains unknown to the LF compiler if certain data source are actually accessed or effects are produced. The code generated for reactions contains a verbatim copy of the reaction body. Additionally, the LF compiler generates an individualized preface that makes only the triggers, sources, and effects of the reaction and state variables of the reactor available to its body.

Section 2.4 will illustrate some functions that LF makes available to the embedded code. This API is target language specific. Furthermore, the LF compiler must support code generation for the desired target languages and requires a compatible implementation of the LF runtime engine. With these prerequisites the LF compiler can synthesize the main reactor into code and create an executable using a target language compiler. LF currently supports C, C++, Python, Typescript, and Rust as targets.

LF's coordination layer establishes separation between reactors, which in turn facilitates a polyglot design where different reactors are written in different target languages. While the LF compiler does not yet support such programs, at least federated LF models will soon provide the option to have polyglot federates.

2.3 SCCharts

SCCharts are a synchronous language that follows that statecharts notation developed by Harel [Har87]. They are inspired by SyncCharts [And03]

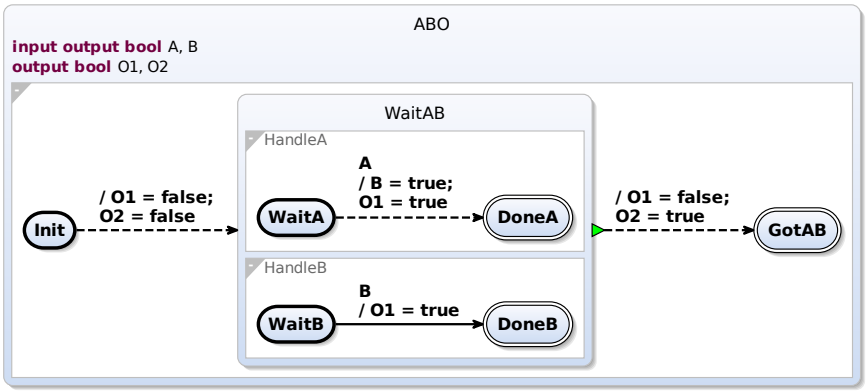


Figure 2.3. The ABO SCChart [HDM+14], consisting only of Core SCCharts elements.

and draw concepts from various other synchronous languages, providing a powerful and versatile set of modeling elements. SCCharts are built around the SC MoC but also follow a more classical synchronous approach compared to LF.

This short overview can only provide a limited introduction into SCCharts. A more comprehensive description is available in the respective publications [HDM+14; Mot17; Smy21; HDM+13] and the official documentation[5].

Core SCCharts Similar to Esterel, SCCharts are defined by a kernel language, called *Core SCCharts*, that facilitates the definition of semantics and code generation. This minimal syntactic core is relatively close to the basic building blocks of statecharts. All other modeling elements in SCCharts, called *extended features*, are reducible to this core language.

Figure 2.3 shows the ABO SCChart that only consists of Core SCCharts elements. Its main purpose is to demonstrate core characteristics of the SCCharts semantics, e. g., shared variables with multiple different values during a tick. The program will produce O1 and O2 depending on the inputs A and B. Both A and B will set O1 to true but with different timing.

[5] https://github.com/kieler/semantics/wiki

Additionally, A will imply a true value on B once, overriding the value set by the environment. After both A and B are correctly consumed by HandleA and HandleB, O1 is reset to false and O2 is set to true. A complete presentation of the behavior of the model can be found in its original publication [HDM+14].

This model illustrates that Core SCCharts consist of states, variables, regions, and transitions that can carry an optional trigger and effects, separated by a slash. The top level state of an SCChart is called *root state*, in this example named ABO. It declares the input output interface of the program. Here, the interface consists of A, B, O1, and O2. A and B are both inputs and outputs, which means they are consumed from the environment but can be altered and provided as product of the program.

Composition Regions are used to compose SCCharts hierarchically and additionally express concurrency if there are multiple regions in the same state. If a state with one or more inner regions, called *superstate*, is entered, all regions immediately start executing. Each region must have an *initial* state, drawn with thicker border (e. g., Init), and can have one or more *final* states, drawn with a double border (e. g., GotAB). When a region reaches a final state, it terminates.[6] Only when all inner regions have terminated, it can be left via a termination transition (indicated by a green triangle), as in the case of WaitAB.

Reactivity & Synchronicity While LF follows an event-driven dataflow approach with connected reactors and reactions as event handlers, SCCharts with its statecharts notation expresses behavior in a control-flow manner. Yet, both languages react in synchrony and in discrete ticks, following the synchronous idea. Where LF processes events by running reactions, SCCharts transitions from one state to another. With concurrent regions there can be multiple active states. Each active state checks its available outgoing transitions (ordered by priority) whether its trigger expression holds, takes the first match, executes its effect sequence, and passes activity on to the target state. This continues until no further transition is enabled, which marks the end of the tick. Hence, ticks act as a barrier synchronization

[6]Transitions leaving final states and final states with inner behavior are considered an extended feature and are not part of Core SCCharts.

for concurrent regions. In contrast to reactions, SCCharts can reenter the same state and thus run the same behavior multiple times, w.r.t. limitations by the SC MoC and code generation approach.

Time SCCharts do not have a built-in notation for expressing time or tagged events as LF has. It follows the idea that time is "just" another input to the system that comes in synchrony with other input values. This results in a multiform notion of time as in Esterel, where for example one signal indicates the passage of a second, while another represent a minute [Ber99]. The pendulum example in Section 1.1.2 illustrates this approach. However, as already mentioned, this approach has limitations. Chapter 4 will present a more advanced way of handling time, which is closer to LF and synchronous languages such as Céu.

Yet, transitions have an important timing property concerning the discrete synchronous ticks. They can be either *delayed* or *immediate*. Delayed transitions require that at least one tick has passed after their source state was entered before they can be taken, while immediate ones are always available. Hence, a delayed transition contains a pause in the imperative sense of synchronous languages. In the graphical syntax, delayed transitions are solid lines, while immediate ones are dashed. For `ABO` this means that the SCChart cannot reach the state `GotAB` during the first reaction (assuming the appropriate inputs) because region `HandleB` has a delayed transition and always consumes at least one tick before state `WaitB` can be left.

Sequential Constructiveness The semantics of SCCharts adhere to the *Sequentially Constructive (SC)* MoC [HDM+14] that establishes causality and determinism. A key factor is sequentiality, especially compared to classical synchronous languages, such as Esterel. In these languages, signals are the first-class citizens and their rather strict write-before-read protocol is enforced globally. This rules out sequential check-and-override patterns common to imperative programming, e. g., `if (x < 42) x++`, even in the absence of concurrency because it conflicts with the assumption of a globally consistent state. Destructive updates can be encoded in thread-local variables in Esterel, but cannot be shared concurrently. Even the `ABO` example in Figure 2.3 would be rejected in Esterel because it may write different

values to O1 in the same tick. The SC MoC relaxes this limitation and unifies signals, channels, and local variables into a single notion, the *SC-variable*.

SC-variables can have different values during a tick and are guided by a dependency graph that considers natural sequentiality in the code and only applies a synchronization protocol to concurrent contexts. This is similar to LF where connections between reactors are subject to a write-before-read protocol, but reactions in a reactor are ordered sequentially. The latter is imposed by the fact that the state variables are shared between the reactions of a reactor, and they may perform destructive updates on these variables, as well as on output ports. In SCCharts the communication solely relies on SC-variables, and they are shared between all concurrent regions of their declaring state and into deeper levels of hierarchy.

In a concurrent context, SC-variables are synchronized under the *Initialize-Update-Read Protocol (IURP)*. The IURP allows variables to be initialized by an *absolute write* first and then permits multiple *relative updates* that are required to be commuting[7] (i. e., in case of multiple writers, a combination function must deterministically unify the value) before the value can be read.

In ABO, the transition to WaitAB occurs before any behavior inside that state and, hence, is ordered sequentially. The IURP only applies to variables shared in the scope of the two regions inside WaitAB. This ensures that writing to B in HandleA will happen before reading it in HandleB. The two concurrent initializations of O1 are sound since they are commuting and any ordering will yield the same result. The IURP applies per SC-variable, instant, and concurrent context. Hence, the end of tick or leaving a superstate resets the IURP for this variable.

Another important difference to LF is that SCCharts feature their own expression language for triggers and effects. They are analyzed in a white-box approach to extract variables accesses and in turn infer dependencies. In contrast to that, LF uses the reaction signatures and connections for explicit dependencies. For integrating host code, SCCharts rely on a similar principle as LF, see Section 2.3.3, and with more modular SCCharts or

[7] In [HDM+14] this is called "confluent", but "commuting" seems more precise in this context. The execution of all writers is "confluent" because they are pairwise "commuting."

dataflow SCCharts (Section 2.3.1) the lines further blur. Yet, this remains a major distinguishing factor as SCCharts is not only a coordination language but also a programming language by itself.

Concurrency In SCCharts, concurrency is expressed by regions. Yet, the actual runtime scheduling of region and potential interleaving is controlled by SC-implied dependencies. During compilation, SCCharts are usually represented by an Sequentially Constructive Graph (SCG) [HDM+14], a CFG notation extended by synchronous constructs for concurrency and delays. It is a one-to-one result of a further normalized Core SCChart. The SCG then is augmented with the data-dependencies imposed by sequentiality and the IURP. In practice, the compiler performs a static structural causality analysis to determine SC-admissible schedules (i. e., adhering to the IURP) and rule out nondeterminism. For such statically SC-admissible programs, multiple low-level code synthesis strategies are available: (1) a hardware-oriented netlist approach [HDM+14], (2) a more dynamic priority-based compilation [HDM+14] with a limited support for cyclic execution, and (3) a state-based approach [SMH18; Smy21] that synthesizes SCCharts into more readable code and preserves the state machine structure. The state-based code synthesis also features more lean variants dropping support for some extended features and interleaving between regions in favor of more modular and simple code [Smy21].

However, these approaches focus on statically establishing a single SC schedule by removing concurrency. While it is a set goal for future development, SCCharts currently do not provide multithreaded, parallel, or distributed execution that utilizes concurrency modeled by regions. LF's multithreaded and distributed capabilities could act as a bridge technology in this regard, see Section 6.3.1.

Extended Features Beyond Core SCCharts, there is a wide range of modeling elements adopted from other synchronous languages. All these extended features are reduced to the core by model-to-model transformations in the compiler. The following descriptions only give a brief overview of major extended features. A complete presentation including semantics and transformations is provided by Motika [Mot17].

Transitions can be configured to cause any combination of the following behavior.

weak/strong abort preempts inner behavior of the source state and forces all inner regions to terminate, when leaving the state. While the strong variant prevents any execution of inner behavior, a weak abort grants a "last wish." That means all behavior that can execute during the current tick is executed before leaving the state.

deferred suppresses any immediate behavior of the target state.

shallow/deep history causes the target state to continue its behavior from the last active state when left, instead of the initial state. The shallow variant only affects regions directly inside the target state, while deep also includes nested regions recursively.

count delay only activates the transition after the trigger was met at least n-times after the entry of the source state.

States can carry a list of actions of the following kind.

entry executes only when entering a state.

exit executes only when leaving a state.

(immediate) during executes as long as a state is active. The immediate variant will start in the same tick the state is entered, otherwise it is delayed by one tick.

(immediate) (weak) suspend prevents all execution in the state as long as its condition holds. All behavior is frozen in its current state but may continue normally if the condition is no longer met. The immediate variant can take effect in the same instant the state is entered, otherwise it is delayed by one tick. The weak variant grants a "last wish," similar to weak abort.

Furthermore, SCCharts provide an extended data-type for Esterel's signals that expands naturally into SC-variables with their IURP [SMR+17; RSM+15]. There are some more features, but these are beyond the scope of this thesis.

Modularity While in LF reactors act as instantiable classes, SCCharts represent modules that can be *referenced* by states in other SCCharts [SMS+15]. A

state that references another SCChart represents an instance of that module. However, there is no notion of OO or the flexibility of inheritance as in LF; Section 5.3 will introduce these capabilities. Additionally, a referencing state has to provide a *binding* for all input and output variables of that module that assigns locally available variables to them. This connects the abstract module to the concrete context, similar to connections in LF. Section 2.4.1 will present an example for this.

Referenced SCCharts are handled by a macro expansion mechanism inspired by Esterel [Ber99]. References are statically expanded at compile time, similar to function inlining or macros of the C preprocessor. The body of the referencing state is replaced by the module and all input output variables are substituted according to the binding. This mechanism fits well with synchronous languages, as it eases a global static analysis of programs and is semantically solid.

LF uses a similar mechanism. An LF program is fully instantiated/expanded at compile time in order to perform the global dependency analysis. For some target languages, namely C++ and Typescript, an additional analysis is performed at runtime. However, a difference is that LF does not synthesize a fully expanded model but composes the modular reactor network again at runtime, whereas SCCharts by default produces fully expanded code. Referenced SCCharts are treated as an extended feature. Yet, there is also an alternative approach for modular SCCharts compilation [Lüd21; Smy21].

LF's concept of mutations or any dynamic restructuring is not supported by SCCharts or any other common synchronous language.

2.3.1 Dataflow SCCharts

In addition to classical statechart design, SCCharts also enable hybrid designs by providing a dataflow notation [GSS+20; GSS+22; Smy21]. This feature is inspired by SCADE but also has parallels to LF. In special dataflow regions, SCCharts can be instantiated as actors and control logic is expressed as equation systems.

Figure 2.4 illustrates the FurutaPendulum program from Figure 1.3 modeled as a hybrid dataflow SCChart. In the root state, the seven regions are

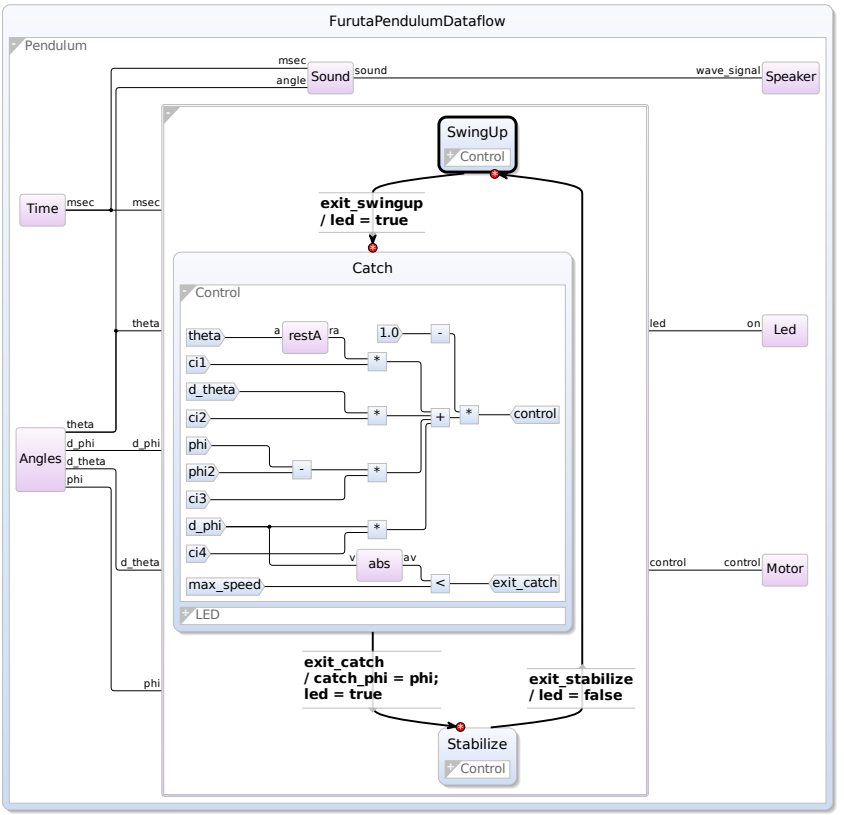


Figure 2.4. The FurutaPendulum program modeled as a hybrid dataflow SCChart.

now replaced by a single dataflow region. Each former region is now an actor and their communication channels are explicitly visible, like in the LF equivalent. Each actor is still an SCChart, as in the case of the expanded PendulumController in the middle, which still has the same states for its modes. However, in this example the calls to the external functions that compute the control behavior are replaced by dataflow regions. The Control region shows a dataflow representation of the actual computation for the control

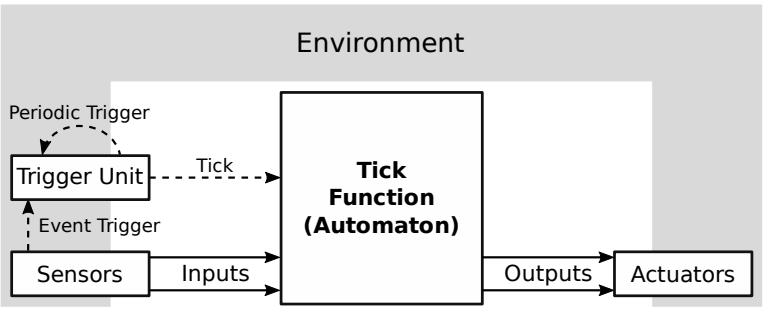


Figure 2.5. SCCharts-specific reactive program components and their interaction with each other and the environment.

result and exit_catch condition. In this case, restA and abs are not SCCharts but external C function for restricting the angle value and computing the modulus. The additional inputs are constants that configure the behavior, but all declarations are omitted for brevity.

In the SCCharts compilation dataflow is treated as an extended feature and transformed into referenced SCCharts and during actions. Yet, dataflow SCCharts bear opportunities for a more modular compilation [Lüd21], provide a more OO perspective in SCCharts (Chapter 5), and could act as bridge to LF (Section 6.3.1).

2.3.2 Runtime Environment

SCCharts follow the classical approach of synchronous languages and produce a *tick function* (or reaction function) as the result of a compilation [PEB07]. A tick function is a procedure that, if invoked with the current inputs, advances the state of the synchronous program by one reaction (tick) and produces outputs. Compared to LF, this approach results in a different perspective on the structure and environment of such a reactive program.

Figure 2.5 presents the reactive system schematics from Figure 2.1 concretized for SCCharts. The reactive program is represented by the Tick Function in the center. All other components are considered predominantly part of the environment, from the perspective of the automaton itself (cf.

Section 2.2.1). On the top left is the Trigger Unit. This is some application-specific procedure that will invoke a Tick of the Tick Function, indicated by the dashed arrow. The Trigger Unit could be periodically triggered or event-driven via Sensors. However, the actual implementation is up to the code embedding the tick function (execution shell [PEB07]). Section 4.3 will compare the implications of different triggering strategies. When invoked, the Tick Function receives inputs from potential Sensors and produces outputs to control the Actuators.

In Figure 2.2, the data exchange of the LF program with the Sensors and Actuators is labeled Read and Control. This does not mean that LF programs do not have inputs or outputs, which they indeed receive from sensors, respectively send to actuators. Instead, this should emphasize that SCCharts, and synchronous languages in general, typically communicate with the environment by a dedicated input output interface, usually using synchronous signals, channels, or variables. Main reactors in LF, however, do not permit input or output ports. Again, this does not mean that a tick function cannot read its own data from sensors or cannot directly invoke an actuator's API to control it, but such a design can be considered uncommon. In Figure 1.3 such a design is present in order to correspond to the LF implementation, but to be more in line with synchronous languages, the FurutaPendulum would rather declare the local variables (theta, control, etc.) as inputs and outputs and handle hardware communication externally instead of in regions.

The tick function design relies on an external application-specific implementation for invocation and handling of inputs and outputs. Listing 2.1 illustrates a main function example in C that invokes the tick function generated for the PendulumSound SCChart in Listing 2.5.[8] The program in Listing 2.1a first includes the header with the tick function for the SCChart (Listing 2.1b) and other required libraries. Then it defines the main function, which creates a variable for the state of the PendulumSound program and initializes it by passing it to reset. For the tick loop, it creates an infinite loop with no delay (Section 4.3). In every cycle, first, the angle is read from the hardware and stored in the input variable in the state. The millisecond

[8]This example is chosen for brevity, since it only represents a subset of a potential FurutaPendulum interface.

```c
#include "PendulumSound.h"
#include "HardwareMockup.h"
int main(int argc, const char* argv[]) {
  TickData model;
  reset(&model);
  while (1) {
    model.angle = read_from("theta");
    model.msec = msec_passed();
    tick(&model);
    write_to("speaker", model.sound);
  }
}
```

(a) Main function invoking the tick function and handling inputs and outputs.

```c
typedef struct {
  double angle;
  char msec;
  char sound;
  ...
} TickData;

void reset(TickData* d);
void tick(TickData* d);
```

(b) PendulumSound.h generated by the SCCharts compiler.

Listing 2.1. Abstract implementation of a tick function loop for the PendulumSound SCChart in C.

input is likewise determined by a utility function. Then, the tick function is called, and finally the output is written to the hardware.

This setup illustrates that the tick function approach is more lightweight in terms of code generation but requires an additional environment implementation, compared to a standalone approach in LF. Yet, the SCCharts is also able to generate such a tick function wrapper and then compile and deploy an executable, if the context is known, e. g., for simulation or based on templates [Smy21]. It would also be possible to embed a tick function in LF and use it as an environment. In such a setup the LF runtime engine would act as the Trigger Unit. Section 3.4.4 will discuss such a design.

2.3.3 Target Language Integration

SCCharts feature a design primarily independent of any target language, by providing a small generic expression language for specifying conditions and computations directly in SCCharts. If relying on this core language, a model can be specified without a pre-defined target. Furthermore, all SCCharts structures and the expression language itself can easily be translated into different target languages [HDM+14]. For example in the netlist compilation

approach, the final sequentialized code only relies on conditional statements and assignments, which is available in nearly every programming language and easy to synthesize. SCCharts currently support C, Java, and VHDL.

The inputs and outputs produce a simple communication interface to the environment, which could also be used for coordinating external behavior. Additionally, SCCharts support direct interaction with their *host language*, e. g., the `sound_frequency` function in Figure 1.3. Section 2.4 will present the involved `extern` declarations. The term "host language" is synonymous to target language but was coined by the fact that the tick function design implies a host system embedding the synchronous program.[9] External host functions in SCCharts are inspired by Esterel's host language integration [PEB07; Ber99]. They are treated as black-boxes, same as reaction code in LF. In order to correctly include them in the causality analysis and to provide deterministic handling, an interface for potential data accesses is required. SCCharts infer this from the use of passed arguments. Call-by-value parameters are considered read accesses and call-by-reference non-confluent writers. In most cases and under the assumption that the function does not have side effects, this is sufficient to establish a causal relation to the surrounding synchronous program. Section 5.4 further discusses this aspect and introduces a concept for deterministically interacting with objects and their methods.

An `extern` declaration introduces an external function, provided by a verbatim string, to the SCCharts expression language under an alias. This enables a degree of multilanguage support. For example the line

```
extern @C "rand", @Java "Math.random" random
```

declares a `random` function that works in C and Java using annotations to synthesize the correct host code depending on the compilation target. Additional annotations in the SCCharts source file would handle the correct host-specific import of the required libraries. Like LF and Esterel, SCCharts can use the host's type system to define variables. For example

```
host "uint64_t" mask
```

[9] Even if synonymous, in this thesis the term "target language" or "target code" will be used primarily in the context of LF's polyglot reaction approach, while "host language" or "host code" will describe the way synchronous languages and SCCharts handle the interfacing with external languages.

will create a 64 bit unsigned integer variable by using the type from the C library stdint. SCCharts also has a verbatim code expressions. Any string in the form ` ... ` will be directly passed on to the generated code, similar to {= ... =} in LF.

This host language integration enables SCCharts to function as a deterministic coordination layer around an existing software systems without requiring communication via an input output interface. However, a polyglot programming concept that effectively mixes multiple host languages in the same SCChart, similar to the vision in LF, is not supported nor planned.

2.4 The Furuta Pendulum Example in Detail

Section 1.1.2 presented the basic setup and objectives in the augmented Furuta pendulum scenario, as well as a first look at the two models in LF and SCCharts. In this section the implementation and the practical application of design elements in LF and SCCharts are in focus. This involves the modular composition starting with the main program and the two primary components PendulumSound and PendulumController.

Again, the inner workings of modules for platform-specific hardware communication are excluded. Furthermore, the code factors out some of the control logic into an external file, used in both the SCCharts and LF model. The motor control behavior for the pendulum arm replicates a solution by Eker et al. [LEJ+02]. All source files are available online.[10]

2.4.1 Main Program

The FurutaPendulum diagrams in Figure 1.2 and Figure 1.3 show the fully instantiated main reactor, respectively the expanded root state, of the main program. In the actual source, the program composes the two separate components for sound and motor control with the hardware handlers and sets up communication channels.

[10] https://github.com/a-sr/furuta-pendulum

```
target C {
  cmake-include: ["behavior.cmake"]
}
preamble {=
  #include "behavior.h"
=}
import PendulumController from
        "PendulumController.lf";
import PendulumSound from
        "PendulumSound.lf";
import Angles, Motor, LED, Speaker from
        "PendulumHardware.lf";

main reactor {
  controller = new PendulumController();
  sound = new PendulumSound();
  angles = new Angles();
  led = new LED();
  speaker = new Speaker();
  motor = new Motor();

  angles.theta -> controller.theta;
  angles.d_theta -> controller.d_theta;
  angles.phi -> controller.phi;
  angles.d_phi -> controller.d_phi;
  controller.control -> motor.control;
  controller.led -> led.on;
  angles.theta -> sound.angle;
  sound.sound -> speaker.wave_signal;
}
```

Listing 2.2. Source code of the FurutaPendulum main program in LF.

Lingua Franca Listing 2.2 shows the textual source of the main reactor. The first line specifies the target language that the reactors will use, in this case C. It also includes a file that will help the build system find the source files that provide the low-level behavior implementation. The subsequent preamble includes the corresponding header. The next lines import the relevant reactors from accompanied files. The main reactor then instantiates these reactors and sets up connections between their input and output ports.

SCChart Listing 2.3 shows the textual source of the main SCChart. Similar to the LF program, the first lines make the external files known to the build system, include the header file in the generated code, and import the SCCharts modules. The following implementation of the SCChart starts with the declaration of shared variables for the communication between the modules. Afterwards, seven regions are defined. Each region has a single initial state that references one of the modules. The parameters in these module macros represent the binding of the local variables to the inputs and outputs of the respective module. Hence, they correspond (in their use) to the connections between the LF reactors.

```
#resource "behavior.h", "behavior.c"
#hostcode-c "#include \"behavior.h\""
import "PendulumController.sctx"
import "PendulumSound.sctx"
import "PendulumHardware.sctx"

scchart FurutaPendulum {
  float theta, d_theta, phi, d_phi, control
  bool msec, led, sound

  region Sound {
    initial state PendulumSound is
        PendulumSound(theta, msec, sound)
  }
  region Controller {
    initial state PendulumController is
        PendulumController(theta, d_theta, phi,
        d_phi, msec, control, led)
  }
  region Time {
    initial state Time is Time(msec)
  }
  region Angles {
    initial state Angles is Angles(theta,
        d_theta, phi, d_phi)
  }
  region Speaker {
    initial state Speaker is Speaker(sound)
  }
  region LED {
    initial state LED is LED(led)
  }
  region Motor {
    initial state Motor is Motor(control)
  }
}
```

Listing 2.3. Source code of the FurutaPendulum main program in SCCharts.

2.4.2 Pendulum Sound

The PendulumSound module produces a square wave signal for the speaker. Both implementations use the external sound_frequency function to compute the correct frequency for the note to play based on an angle value.

Lingua Franca The reactor in Listing 2.4 first declares its input port for the angle and output port for the speaker signal. It has three state variables: wave_state, cycle_duration, and last_switch. The first indicates whether the signal is currently in its duty cycle or not. The next represents the current length of a half cycle. The last one is the time of the last signal edge. Additionally, the reactor has a logical action that carries a time value as payload.

The behavior is defined in a single reaction that is triggered by input events on the angle port or the alternate action (both triggers can also be present simultaneously). The effects list the output port and the action.

```
1  target C;
2  reactor PendulumSound {
3    input angle: double;
4    output sound: bool;
5    state wave_state: bool = false;
6    state cycle_duration: time = 0;
7    state last_switch: time = 0;
8    logical action alternate: time;
9    reaction(angle, alternate) -> sound, alternate {=
10     if (alternate->is_present && alternate->value == self->cycle_duration) {
11       self->wave_state = !self->wave_state;
12       lf_set(sound, self->wave_state);
13       lf_schedule_copy(alternate, self->cycle_duration, &self->cycle_duration, 1);
14       self->last_switch = lf_time_logical();
15     }
16     if (angle->is_present) {
17       interval_t new_duration = SEC((1 / sound_frequency(angle->value)) / 2);
18       if (new_duration != self->cycle_duration) {
19         interval_t remaining_time = MAX(new_duration - (lf_time_logical() - self->last_switch), 0);
20         lf_schedule_copy(alternate, remaining_time, &new_duration, 1);
21         self->cycle_duration = new_duration;
22       }
23     }
24   =}
25 }
```

Listing 2.4. Source code of the PendulumSound component in LF.

The basic idea is to determine the targeted frequency based on the angle input and then schedule the action at the time of the next planned signal edge. The reaction code is separated into two if statements. In the fist one (lines 10 to 15), the periodic square signal is produced by emitting alternating outputs based on the occurrence of the action. In the second (lines 16 to 23), the cycle duration is adjusted based on the angle-induced frequency.

To illustrate how these parts work together, we start with a first angle event at tag $(0, 0)$. The reaction will be triggered and the first if block is skipped because only angle is present. In the second block, the new duration for a cycle is computed using the sound_frequency function. The SEC function handles the conversion into LF's time representation. If the new duration differs from the current one, the timing behavior must be adjusted, which is always the case at program start. The remaining time to the next signal switch is computed in line 19. It takes into account the last switch and

```
1  scchart PendulumSound {
2   input float angle
3   input bool msec
4   output bool sound = true
5
6   const int SEC_TO_MSEC = 1000
7   extern "sound_frequency"
          sound_frequency
8   int msecs = 0
9   float duration = 0
10  immediate during do duration =
11   ((1 / sound_frequency(angle)) / 2) * SEC_TO_MSEC
12
13  region {
14   initial state Alternate {
15    during if msec do msecs++
16   }
17   if msecs >= duration do sound = !sound; msecs = 0
18    go to Alternate
19  }
20 }
```

Listing 2.5. Source code of the PendulumSound component in SCCharts.

prevents negative values by a maximum function. This time is then used as a delay for scheduling the action. Finally, it saves the new cycle duration. At time 0, the logic will always determine an immediate signal switch and hence schedule the action with a delay of 0. However, actions impose a microstep delay in this case. Hence, the reaction will not be executed again at tag $(0, 0)$ but at $(0, 1)$. In this turn, only the first part will be active, since angle is absent. This block first toggles the state, sets the output to this new value, schedules the action again with the current cycle duration to ensure a periodic signal, and saves the current logical time in last_switch.

In the further execution, the duration will change as the angle changes, which will result in additional actions scheduled for a time before or after the action that was scheduled based on the old frequency in line 13. This is where the payload becomes relevant. It associates each action with the duration it represents. The if statement in line 10 checks this value and ignores actions that do not represent the currently targeted sound frequency.

This design is a consequence of the current limitation in LF that prohibits removing or adjusting scheduled events.[11] The fact that this implementation introduces a microstep delay for producing a cycle switch "immediately" is mainly for demonstration purposes, as the output could also be set directly before line 20 and then only the next switch would be scheduled.

[11] A limitation that LF may lift in the future by returning a handle upon schedule that enables the user to unschedule an action.

SCChart Listing 2.5 shows the code of the PendulumSound SCChart, initially illustrated in the Sound region of Figure 1.3. Comparing the code with the diagram, one can notice that there is very little difference in the amount of information. This is a clear contrast to LF, which hides the entire implementation in reactions and only illustrates the coordination layer in the diagram. While this is subject to the configuration of details in the diagram, it presents different views on the model that are provided by default. While LF focuses on the coordination aspect, SCCharts' focus is more on behavior modeling, which includes details on concrete conditions and effects of individual transitions. Section 2.5 will discuss the concept of views in more detail.

The SCChart code in Listing 2.5 first declares its module-specific inputs, angle and msec, and the sound output variable. In line 6 it declares a constant for converting seconds into millisecond as it cannot resort to a built-in time model, as in LF. Afterwards, the external hostcode function sound_frequency is defined and two local variables are declared.

Since SCCharts are not explicitly event-driven as LF, the during action updates the cycle duration in each tick based on the angle. The Alternate state in the region advances the msecs counter every time a milliseconds passes. The self transition toggles the sound signal output and resets the counter.

Hence, the signal edge in the square wave signal is modeled as a dynamic threshold between the targeted half cycle length and the passed time, see line 17. However, in order to work with the msec input, the SCChart needs to be executed each millisecond, i. e., every time any operand changes. A sparse execution, more similar to LF, requires a more sophisticated infrastructure, as proposed in Chapter 4.

2.4.3 Pendulum Controller

The PendulumController module controls the motor to balance the pendulum and sets the LED state to indicate its mode of operation. Again, both implementations use external functions for the control logic in each mode and to check for mode switches.

```
1  target C;
2  reactor PendulumController {
3    input theta: double;
4    input d_theta: double;
5    input phi: double;
6    input d_phi: double;
7    output control: double;
8    output led: bool;
9
10   preamble {= typedef enum {SwingUp, Catch,
           Stabilize} ControlModes; =}
11   state control_mode: ControlModes = {=SwingUp=};
12   state led_state: bool = false;
13   state catch_phi: double = 0.0;
14   timer toggle_led(0, 15 msec);
15
16   reaction(toggle_led) -> led {=
17     if (self->control_mode == Catch) {
18       self->led_state = !self->led_state;
19       lf_set(led, self->led_state);
20     }
21   =}
22   reaction(theta, d_theta, phi, d_phi) -> control, led {=
23     switch (self->control_mode) {
24       case SwingUp:
25         lf_set(control, swingup_control(theta->value,
             d_theta->value));
26         if (exit_swingup(theta->value)) {
27           self->control_mode = Catch;
28           self->led_state = true;
29           lf_set(led, self->led_state);
30         }
31         break;
32       case Catch:
33         lf_set(control, catch_control(
             theta->value, d_theta->value,
             phi->value, d_phi->value));
34         if (exit_catch(d_phi->value)) {
35           self->catch_phi = phi->value;
36           self->control_mode = Stabilize;
37           self->led_state = true;
38           lf_set(led, self->led_state);
39         }
40         break;
41       case Stabilize:
42         lf_set(control, stabilize_control(
             theta->value, d_theta->value,
             phi->value, d_phi->value,
             self->catch_phi));
43         if (exit_stabilize(theta->value)) {
44           self->control_mode = SwingUp;
45           self->led_state = false;
46           lf_set(led, self->led_state);
47         }
48         break;
49     }
50   =}
51 }
```

Listing 2.6. Source code of the PendulumController component in LF.

Lingua Franca The reactor code in Listing 2.6[12] is similarly structured as the SoundController. The PendulumController first declares its input and output ports and then local state variables. The type for the control_mode is defined as an enum in C by using a preamable that will be added to the generated code for this reactor. The control_mode is initialized to the SwingUp mode and the led_state starts in its Off state. The value catch_phi must be passed

[12]This code is based on an implementation by Edward A. Lee that adapts the original Ptolemy II model by Eker et al. [LEJ+02] to C and LF. My variant extends this solution by additionally controlling the LED.

between the Catch and Stabilize phase. Additionally, the reactor defines a timer with an initial offset of 0 and a period of 15 msecs.

The first reaction is triggered by this timer. It toggles the led_state and sets the output with this value but only when operating in Catch mode. The second reaction processes all angle related inputs. The implementation assumes that all four events will always be present simultaneously and omits individual checks for presence. The different modes of operation are implemented in a switch statement. In each case, the control output is set with the result of the respective control logic function, which receives the relevant angle data. Afterwards, an if statement checks whether this mode should be left. If that is the case, the new mode is set, the led_state is updated accordingly, and the led output is set.

Setting the LED in the second reaction, ordered after the one for toggling, is important to override the effect of the previous reaction if they happen to be triggered simultaneously. However, while the design with two reactions is quite reasonable, it results in a timing subtlety. Even if the first reaction toggles the LED only in the Catch mode, the timer is not aligned with actual mode change. Hence, the start of the Catch mode may not be the start of the periodic blinking. This issue will be discussed in more detail in Chapter 3.

SCChart As it was the case for the PendulumSound, the source code in Listing 2.7 only reveals minor additional details on the behavior of the PendulumController compared to Figure 1.3. Again, this SCChart declares the inputs and outputs relevant to this module and defines the external functions.

Compared to the LF implementation, the state machine notation enables SCCharts to express the modes more naturally and explicitly. This supersedes the need for the corresponding state variables and the synchronization of the alternating LED to the Catch mode.

The SCChart models the same behavior as its LF counterpart, which includes the way modes are changed. Specifically, at any time, the control output is defined by the current mode, and if it decided to exit this mode and switch to the next, the next one will only determine the value in the subsequent instant. This is modeled by the fact that the during actions for the control output are not immediate and the transitions only perform

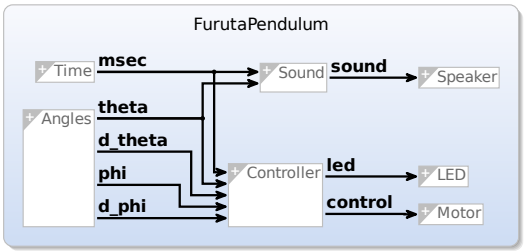
```
1  scchart PendulumController {
2   input float theta, d_theta, phi, d_phi
3   input bool msec
4   output float control = 0
5   output bool led = false
6
7   extern "swingup_control"
          swingup_control
8   extern "exit_swingup" exit_swingup
9   extern "catch_control" catch_control
10  extern "exit_catch" exit_catch
11  extern "stabilize_control"
          stabilize_control
12  extern "exit_stabilize" exit_stabilize
13  float catch_phi = 0
14
15  region {
16   initial state SwingUp {
17    immediate during do control =
          swingup_control(theta, d_theta)
18   }
19   if exit_swingup(theta) do led = true
19   if exit_swingup(theta) do led = true
20    go to Catch
21
22   state Catch {
23    during do control = catch_control(theta, d_theta,
          phi, d_phi)
24    region LED {
25     initial state Toggle
26      if 15 msec do led = !led go to Toggle
27     }
28   }
29   if exit_catch(d_phi) do catch_phi = phi; led = true
30    go to Stabilize
31
32   state Stabilize {
33    during do control = stabilize_control(theta,
          d_theta, phi, d_phi, catch_phi)
34   }
35   if exit_stabilize(theta) do led = false
36    go to SwingUp deferred
37  }
38 }
```

Listing 2.7. Source code of the PendulumController component in SCCharts.

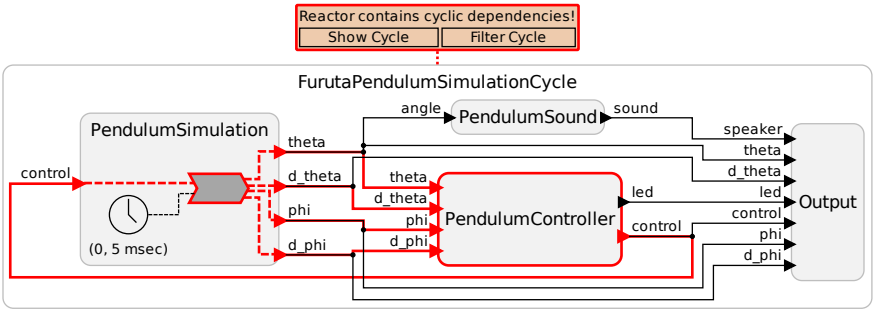
a weak abort (**go to** instead of **abort to**). Hence, transitions change the state but the control value is not immediately overridden. One exception is the initial state SwingUp. Here, the during action is immediate to provide a control output in the inital tick. Therefore, the transition from Stabilize enters this state deferred to suppress this immediate behavior and prevent overriding.

2.5 Modeling Pragmatics

Both LF and SCCharts are guided in their design by modeling pragmatics [HLF+22; FH10] and use state-of-the-art pragmatics-aware tooling to enhance developers' grasp of their code and increase their productivity. Following the principles of MDE, LF and SCCharts programs are *models*. In contrast to many other modeling tools, for example SCADE, their source



(a) FurutaPendulum SCChart with induced dataflow, exposing implicit communication between regions.



(b) Highlighted causality cycle in an LF program that simulates the main components of the FurutaPendulum example but creates an immediate feedback loop via control and theta.

Figure 2.6. Two examples for specialized views in SCCharts and LF.

representation is textual instead of graphical, as Section 2.4 already illustrated. This provides many benefits to the editing process and facilitates version control. Yet, the real strengths of MDE play out with graphical representations [Gur99], namely diagrams. At this point *transient views* [HLF+22; SSH13; FH10] come in and provide customized diagrammatic representations. This idea corresponds well with the Model-View-Controller (MVC) paradigm [Ree79]. However, it is unlikely that a user who just specified the model textually, is eager to manually create, arrange, and continuously synchronize an additional graphical view. Hence, automatic diagram synthesis and layout algorithms are a key enabler in this field. All illustrations

of LF and SCCharts models in this thesis are such automatically generated diagrams.

Furthermore, an automated process enables on-the-fly creation of different views for the same model that focus on or reveal different aspects. Examples for such specialized views can be found in Figure 2.6. Figure 2.6a illustrates the FurutaPendulum SCChart from Figure 1.3 but with induced dataflow [WSS+18]. This view reveals the implicit data-dependencies between concurrent regions, which are induced by their internal variable accesses and visualized as dataflow. In Figure 2.6b, an LF program that simulates the PendulumControl and PendulumSound reactors is implemented incorrectly, such that an immediate feedback is present between the simulation and the control reactor. The compiler detects this causality cycle and this view highlights the involved components and communicates the problem.

Further capabilities in terms of interactivity, such as filtering, focus and context methods, or an adjustable level of detail, enhance the browsing and exploration workflow of diagrams. This can lead to a better understanding of the model. For example, the buttons in the message box in Figure 2.6b indicate that this view can be filtered to only show components that are part of the cycle. The result is a modeling experience that cannot be adequately described textually and should be experienced personally to reveal its true advantages.

The KIELER Project The open-source *Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)* project[13] has been the birthplace and testbed for many pragmatics-aware modeling technologies and languages. It provides a comprehensive modeling experience that combines textual editing and interactive diagrams with automatic layout.

In the past, KIELER was built around the Eclipse Integrated Development Environment (IDE), a versatile and extensible Java-based development tool. Recent development also enables simultaneous support for Visual Studio Code [Dom18; Ren18]. A major factor in all of this is the Xtext framework [EB10]. It follows a model-driven approach for creating textual

[13] https://rtsys.informatik.uni-kiel.de/kieler/

languages. Based on a grammar specification and a metamodel, Xtext automatically creates a parser and serializer, as well as multiplatform editor support with syntax highlighting, content-assist, jump-to-declaration, etc. The KIELER Lightweight Diagrams (KLighD) framework [SSH13] then facilitates the implementation of syntheses that turn the underlying models into custom diagrams. It provides for automatic layout based on the Eclipse Layout Kernel (ELK) [SSH14] framework, handles rendering, and supports interactivity.

While the SCCharts implementation is part of KIELER, LF is a standalone project but uses the same frameworks as SCCharts for its language implementation and diagram support.

KIELER also comes with an interactive model-based compiler [SSH18c; Smy21] that is closely integrated into the diagram tooling. The model-to-model transformations that form the SCCharts compile chain are implemented in this compiler. This enables a visual inspection of all intermediate results and facilitates checking or understanding the effect of individual extended features.

Part I

Lingua Franca

Chapter 3

Modal Models

The direct comparison of the pendulum implementation in LF with its SCCharts counterpart (in Section 1.2 and Section 2.4) reveals two major opportunities for improvement in LF;

1. extending modeling capabilities to express modal behavior[1], such as SwingUp, Catch, and Stabilize; and
2. binding of timed elements, such as the timer in PendulumController, to specific modes of operation.

Modeling (1.) Complex software systems often feature distinct modes of operation that provide a particular behavior for a specific context. The Furuta pendulum example represents this characteristic well, even if it is relatively small. Yet, the implementation of modes in a state machine using a switch pattern is relatively extensive, as Listing 2.6 illustrates. More importantly, such handwritten code contradicts the fundamental idea of model-driven engineering and, moreover, is easily prone to errors, complex to extend, and hinders formal verification.

Furthermore, the entire modal structure is "hidden" in a single reaction, see Figure 1.3. From a modeling perspective, the explicit use of separate states for the modes in the SCChart variant provides in comparison an additional value by enabling meaningful diagrams. While there are techniques

[1]The term *mode* describes the concept of combining program behavior into modes of operation. While the most natural notation for modes is a state machine, modes can subsume multiple states of a system into a single mode [MR98]. Hence, modes and states can be used synonymously, if states do not refer to a single memory state but a more abstract modal state of the program, as it is usually the case in statecharts and SCCharts.

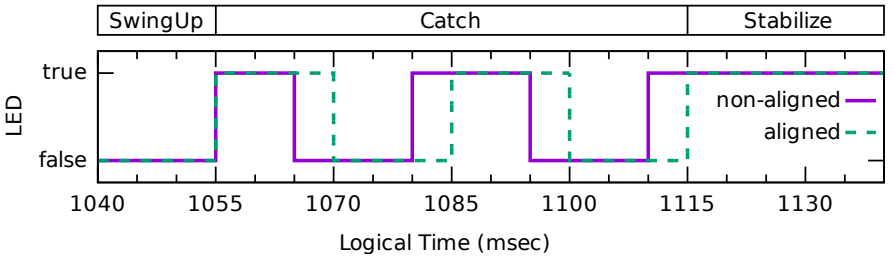


Figure 3.1. An excerpt of the led output by the PendulumController reactor in a simulation. It illustrates a timing behavior of a non-aligned and an aligned timer in relation to the catch mode.

to extract such mode diagrams from handwritten code, as Section 3.1.2 will present, this contradicts the black-box approach that governs LF's reactions.

Hence, there is merit to expressing modal structures directly on the coordination layer of LF. It breaks down the behavior into smaller units that promote code readability and can be supported by more meaningful diagrams. Furthermore, it enables robust code generation for controlling these modal units and exposes mutually executive behavior that can be used to enhance modeling capabilities or facilitate verification.

Timing (2.) Figure 3.1 shows an excerpt of the LED output by the PendulumController reactor in a simulation. The simulation emulates the pendulum behavior with a sample period of 5 msec. The presented interval reflects the time in which the controller switches from mode SwingUp to Catch and later to Stabilize, as indicated at the top of the plot. The solid non-aligned line represents the output of the PendulumController reactor presented in Listing 2.6. According to its implementation, the LED is set directly by the second reaction when the mode changes, at time 1055 and 1115 msec. During Catch mode the signal is toggled by the first reaction when the timer expires. Since the timer runs all the time, it happens to trigger the reaction at 1065 msec in this simulation. This reduces the on cycle to only 10 msec. The dashed aligned line is the output of the corresponding SCChart that models the alternating LED outputs inside the Catch and thus aligns their activity.

Of course, this behavior has no critical consequences in this particular scenario. Even if the start of the cycle would be aligned, the last cycle could be cut off depending on the start of the Stabilize mode.[2] Alternatively, a logical action could be used to create a periodic triggering that starts with the Catch mode, similar to the one in the PendulumSound reactor. Yet, a timer is more simple and robust than manually scheduled actions, but it cannot be controlled correctly if modes are only expressed on the target code level and not in the model.

This example shows that timing is an important issue when working with modes and that modal models are able to express time-related associations more naturally if their semantics are carefully designed.

Goals The main goal is to bring the advantages of modal models to LF and the reactor-oriented programming paradigm, and indirectly also closer to mainstream programming languages, which can be embedded into LF. From FSMs to statecharts up to SCCharts, there are many concepts and languages that are already able to express modal models. While the idea of modes is not new, a seamless integration into LF that is guided by its fundamental principles is. The goal is to create modal models that offer the following characteristics.

Lean design Modes should constitute a minimal coordination layer that provides the most essential functionality but still offers maximal versatility and user adjustability.

Polyglotism The modal notation should act as flexible multilanguage wrapper that focuses on the user's language and requires only minor adaptation effort.

Time sensitivity The model should offer a reliable and precise way to specify time sensitive modal behavior, even in parallel and distributed environments.

Concurrency The design should enable the composition of multiple separate modal units acting independently.

[2] It is only a coincidence that the Catch phase in this simulation is exactly 60 msec long.

Determinism A model must yield unambiguous and reproducible output behavior for the same sequence of input events.

The concept of *modal reactors* presented in this chapter embodies these very principles and embraces the crucial black-box approach of reactions.

Outline This chapter starts with a brief presentation of related work, covering different languages and approaches to express modal behavior. Next, Section 3.2 illustrates the basic principles of modal reactors by presenting a variant of the PendulumController reactor that uses modes. Subsequently, Section 3.3 describes the detailed concept and implementation of modal reactors in LF. Finally, Section 3.4 discusses the proposed design and briefly illustrates potential alternatives.

3.1 Related Work

The idea of expressing modal behavior is not new and there are many languages and concepts that enable the use of mode automata or state machines in this regard. While the design of modal reactors builds on existing concepts, there is no previous work that truly matches the unique principles of LF, especially in terms of polyglot design.

Statecharts A natural notation for modal behavior can be found in the many variants of FSMs. Beyond that, there are statecharts that offer more feature-rich language constructs. With their hierarchical composition, they facilitate the encapsulation of individual behavior and fine-grained states into broader modes of operation.

Synchronous dialects, such as SyncCharts, adapt statecharts into a semantic domain that is relatively close to LF. However, in contrast to plain statecharts or FSMs, modes for LF require a more hybrid approach, to accommodate for the dataflow nature of reactors.

SCCharts SCCharts surpass most other statecharts languages in terms of features variety and versatility. This also includes modal modeling capabilities. Yet, the focus of the modal model concept in this thesis is on LF,

since it provides a unique opportunity to design modes in an actor-oriented, polyglot, coordination context.

While SCCharts also provide a dataflow notation (Section 2.3.1), their own form of multilanguage support (Section 2.3.3), and can be used to deterministically orchestrate processes, these aspects are more pronounced in LF.

3.1.1 Mode Extensions

Since LF is a reactor-oriented language, work that extends dataflow notations with state machines is particularly relevant. Yet, a commonality in all existing work is that they feature an additional extensive notation for states, transitions, triggers, and effects, representing an obstacle for a lean and polyglot integration in the context of LF.

SCADE The work that is perhaps the closest in spirit to modal reactors is the extension of Lustre and SCADE with state machines. Maraninchi and Rémond propose a concept of mode-automata [MR98] that combines statecharts-like automata with a minimal Lustre language.

Colaço et al. pick up this approach and adapt a more lean variant into Lustre and SCADE [CPP05; CHP06]. At its core, SCADE is a synchronous dataflow modeling language, where concurrent nodes communicate via clocked streams, see also Section 2.1.1. With the extension for modes, nodes received a state machine notation that features reset and history transitions, as well as preemption. Most importantly, states encapsulate equations, which enables a hybrid design that mixes modes and dataflow.

The implementation uses a clock-directed approach based on a source-to-source transformation. It extends the original clocks of Lustre/SCADE into a richer type system that encodes and controls modes. In order to simplify analysis, states and their equations are in mutual exclusion. Therefore, preemption is restricted, such that a state can only be entered weakly (non-deferred) if the previous is strongly preempted upon leaving or vice versa.

The SCADE approach represents a very similar design to the one later found in modal reactors. However, it is based on the classical synchronous principles, which LF exceeds in terms of distributed execution and notion of time. In this regard, LF embodies a more event-driven concept than a clock-driven one.

Ptolemy The Ptolemy II tool provides an environment for heterogeneous modeling [EJL+03]. It is based on an actor-oriented design that supports nesting and interconnecting components. A unique characteristic is that each level of the hierarchy can have a separate MoC directing its semantics. Ptolemy II provides various MoCs, such as continuous time (CT) or synchronous dataflow (SDF), as well as a state machine layer that enables modal models [LT10; Pto14]. A key to combining arbitrary MoCs in a modal model is a notion of mode-local time to preserve a sound compositional semantics.

As mentioned in Section 2.4, some parts of the low-level implementation of the Furuta pendulum are based on a previous implementation in Ptolemy [LEJ+02]. Figure 3.2 presents an excerpt of this model. It illustrates the different layers of the model governed by different MoCs. A modal model inside the controller manages the different computations of the control output (u in this variant).[3]

In many ways, LF continues the research of the Ptolemy project. Yet, the focus shifted from an experimentation platform for different MoCs to a more application-oriented polyglot coordinating language.

Simulink MathWorks' Simulink tool is an environment for modeling and simulating control logic with block diagrams. With Stateflow [HR04] it provides a statecharts-like notation able to express modes by nesting blocks in different states.

In contrast to the discretized time in LF and synchronous languages, Simulink primarily uses a continuous-time concept with semantics that depend on the configuration of the simulation [CCM+03].

[3] Most of the pendulum controllers in this thesis factor out the control logic into external functions, see Section 2.4. The example of dataflow SCCharts in Figure 2.4 illustrates a variant that includes the modeling of these computations and, hence, corresponds more to the approach in the Ptolemy model.

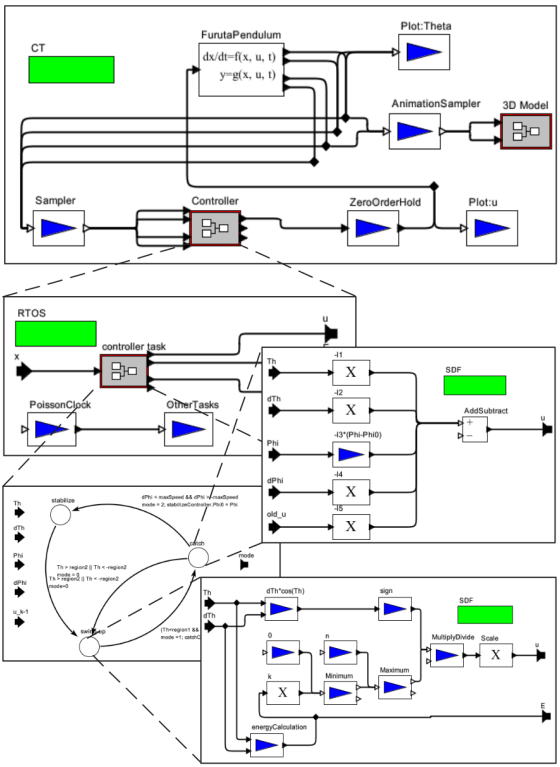


Figure 3.2. A Furuta pendulum implementation in Ptolemy by Liu et al. [LEJ+02]. (©2002 International Federation of Automatic Control. Reproduced with the permission of IFAC from J. Liu, J. Eker, J. W. Janneck, E. A. Lee, "Realistic Simulations of Embedded Control Systems". IFAC Proceedings Volumes, 35/1, pp. 391-396)

LabVIEW National Instruments' LabVIEW is a graphical dataflow language for implementing control systems [Kod20]. It enables modeling modal behavior by using a pattern with a switch structure in a while loop. A dedicated state diagram view supports the user in this design.

While LabVIEW does provide a notion of modes, the pattern-based state notation and strong reliance on a graphical syntax only loosely relates to the modal extension pursued in this thesis.

ROOM The Real-time Object-Oriented Modeling (ROOM) language by Selic et al. [SGW94] uses an actor notation for the high-level specification of distributed real-time systems. The behavior of these actors is modelled in a statecharts dialect, called *ROOMcharts* [Sel93]. In terms of characteristic statecharts features, Selic et al. exclude concurrent composition with the arguments that this aspect is better expressed on the actor level. Hierarchical nesting is kept but is restricted only to the ROOMcharts elements.

Hence, while ROOM actors can be composed hierarchically, ROOMcharts are specified independently and do not enclose inner actors into modal units. Instead, it represents an additional specification for message processing. Furthermore, communication in ROOM is governed by synchronous (blocking) and asynchronous (non-blocking) message delivery rather than a global model of timestamped events as in reactors.

Akka The Java-based actor programming framework Akka [RWB16] provides means to implement concurrent actors that interchange and process event messages. LF uses Akka as a guideline for its performance benchmarks [Loh20; MLB+23]. Akka also supports implementing stateful behavior in actors by extending an FSM actor class [RWB16]. It enables defining states, their event processing, transitions, and internal timers.

However, Akka does not ensure deterministic behavior. Furthermore, the FSM notation does not support hierarchy, resulting in limited hybrid modeling capabilities.

3.1.2 Mode Extraction

Instead of extending a language towards modes, there is also work that extracts the implicit states from the code into a modal view. However, this approach confines the modal modeling capabilities to the target language and does not account for a modal coordination layer.

C/C++ Said et al. [SQK18] propose a technique that explores the state space of a program by mining for state variables and simplifies these into a state machine. The approach of Somé and Lethbridge [SL02] detects special patterns, such as switch statements (as in Listing 2.6), and visualizes

them. The same technique is used by Andersen to convert C/C++ code into SCCharts [And19].

Blech Synchronous languages facilitate detecting states, as they explicitly contain pauses. The translation from imperative Esterel code to SyncCharts [PTH06] illustrates such a procedure but with strong focus of semantic equivalence. The more recent work for the similarly structured Blech language [GG18] applies an approach that emphasizes abstraction and reveals a mode-oriented view on the underlying behavior of the program [LSH+21].

SCCharts An inverse procedure of mode extraction is illustrated by SCCharts' induced dataflow [WSS+18]. It extracts the implicit dataflow relations between concurrent regions, as illustrated in Figure 2.6a.

3.1.3 Modal Augmentation

As an alternative to a separate coordination language with modes, there are various proposals that aim at directly augmenting mainstream programming languages with a notion of states or modes.

Statecharts in C/C++ Wagner et al. describe a design process for flat FSMs directly in C [WSW+06]. An advanced approach by Samek describes the implementation of UML statecharts in C/C++ [Sam08]. While this includes concurrency, there are no provisions for a deterministic behavior.

FairThreads FairThreads [Bou06] are an extension of C that enables cooperative threading. They are implemented based on macros and use native threads. FairThreads also include macros to model automata for auxiliary tasks, as their restricted structure facilitates a more efficient execution w.r.t. threads.

SyncCharts in C There are also some synchronous extensions to C, with SyncCharts in C [Han09] as probably the most relevant for modes. Again, this concepts relies on C macros to provide a light-weight low-level language extension. It enables expressing states and provides deterministic concurrency based on a priority-based dispatching of threads.

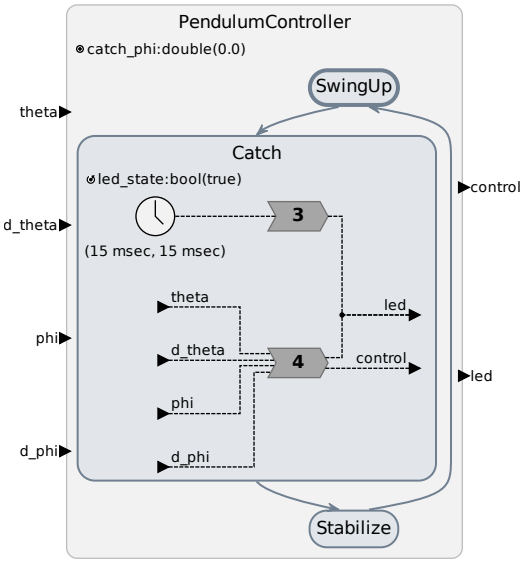


Figure 3.3. The PendulumController reactor using modes.

3.2 The Modal Pendulum Controller

The basic idea of modes in LF is to use the existing reactor model as is and to provide a minimally invasive notation to associate reactions and other contents with modes. Transitions between modes are then triggered as an effect of reactions.

Figure 3.3 illustrates a variant of the PendulumController reactor that uses modes. The reactor now features three separate modes for the different behaviors, similar to the SCCharts variant. Their graphical appearance is inspired by SCCharts, for example by highlighting the initial mode SwingUp with a thicker border. Each mode contains regular reactor elements. In this diagram, only the Catch mode is expanded and reveals its contents. The timer and the reaction are now local to this mode. As a result, the temporal behavior is bound to the activation of the mode and yields the aligned output presented in Figure 3.1, discussed in more detail in Section 3.3.2.

```
1  target C;
2  reactor PendulumController {
3    input theta: double;
4    input d_theta: double;
5    input phi: double;
6    input d_phi: double;
7    output control: double;
8    output led: bool;
9
10   state catch_phi: double = 0.0;
11
12   initial mode SwingUp {
13     reaction(theta, d_theta) ->
           control, led, reset(Catch)  {=
14       lf_set(control,
           swingup_control(theta->value,
           d_theta->value));
15       if (exit_swingup(theta->value)) {
16         lf_set_mode(Catch);
17         lf_set(led, true);
18       }
19     =}
20   }
21   mode Catch {
22     reset state led_state: bool = true;
23     timer toggle(15msec, 15msec);
24     reaction(toggle) -> led {=
25       self->led_state = !self->led_state;
26       lf_set(led, self->led_state);
27     =}
28     reaction(theta, d_theta, phi, d_phi) -> control,
           led, reset(Stabilize) {=
29       lf_set(control, catch_control(theta->value,
           d_theta->value, phi->value, d_phi->value));
30       if (exit_catch(d_phi->value)) {
31         lf_set_mode(Stabilize);
32         lf_set(led, true);
33         self->catch_phi = phi->value;
34       }
35     =}
36   }
37   mode Stabilize {
38     reaction(theta, d_theta, phi, d_phi) -> control,
           led, reset(SwingUp) {=
39       lf_set(control, stabilize_control(theta->value,
           d_theta->value, phi->value, d_phi->value,
           self->catch_phi));
40       if (exit_stabilize(theta->value)) {
41         lf_set_mode(SwingUp);
42         lf_set(led, false);
43       }
44     =}
45   }
46 }
```

Listing 3.1. Source code of the PendulumController reactor with modes.

While the diagram presents modes in a classical statecharts appearance with states and transitions, this is primarily a result of the graphical view. In the textual source there is a more seamless integration into the dataflow-oriented syntax of LF.

Listing 3.1 shows the source code of the modal PendulumController in Figure 3.3. It reveals that the monolithic reaction with the switch pattern for the modes (cf. Listing 2.6) is now split up into three reactions (starting at lines 13, 28, and 38), one in each mode. The modes simply enclose their associated contents. The execution semantics of modes will ensure that only one mode is active at a time. Transitions between modes are implemented as

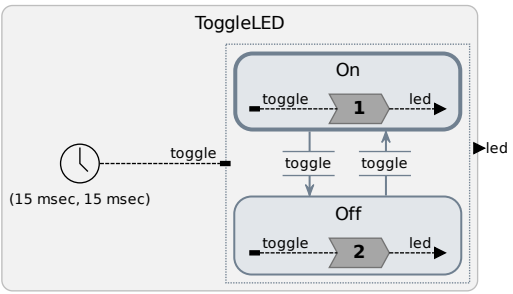


Figure 3.4. The ToggleLED reactor that models the alternating LED state with modes.

```
1  target C;
2  reactor ToggleLED {
3    output led: bool;
4    timer toggle(15 msec, 15 msec);
5
6    initial mode On {
7      reaction(toggle) -> Off, led {=
8        lf_set(led, false);
9        lf_set_mode(Off);
10     =}
11   }
12   mode Off {
13     reaction(toggle) -> On, led {=
14       lf_set(led, true);
15       lf_set_mode(On);
16     =}
17   }
18 }
```

Listing 3.2. Source code of the ToggleLED reactor.

effects of reactions. For example, the reaction of the SwingUp mode in line 13 declares the Catch mode as an effect. This enables the lf_set_mode(Catch) command in line 16 to issue a transition to that mode. The remaining transitions are modeled analogously.

Another notable difference to the implementation presented in Section 2.4 is that the led_state state variable is now located in the Catch mode. Therefore, it has the default value true because the led output is already set to true while transitioning to this mode. Consequently, the timer also has an initial delay of 15 msec to take this into account. Furthermore, the previous control_mode state variable, see Listing 2.6, is superseded by the use of modes on the coordination level of LF.

Replacing State Variables by Modes In this spirit one could further apply this strategy for the led_state state variable. The mode extension of LF supports nesting another modal model in the Catch mode by instantiating a modal reactor. Figure 3.4 illustrates the reactor that is able to replace the led_state state variable, the timer, and toggle reaction in the modal Pendu-

lumController. Listing 3.2 presents the corresponding source code. Instead of inverting the led_state value, the led output is set explicitly by a reaction in each mode (lines 8 and 14). The alternation is caused by the switching between the On and Off mode, triggered by the timer.

This solution is evidently a less compact implementation, but may be preferable in terms of graphical representation. Furthermore, this example illustrates that not all contents of a reactor must be associated with modes. The timer is independent of the two modes and a shared trigger for both reactions. In order to visually separate mode-independent reactor elements from those in modes, the modal model is confined to a dedicated area. As in Figure 3.3, the relevant input and output ports are duplicated into each mode to prevent dependency edges from crossing and visually cluttering the modal model layer. Since the timer is not a named port and cannot be duplicated, as this would imply a different timing semantics (see Section 3.3.2), it is connected with the mode area and has its own named port that is then referenced inside the modes. Furthermore, this diagram is configured to show the triggers of reactions potentially invoking a transition as a label on the respective edge.

3.3 Modal Reactors

The previous section already illustrated some core principles of the proposed modal reactor concept. To sum it up, it enables a partitioning of a reactor's contents into disjoint subsets that are associated with mutually exclusive modes. In a modal reactor, only a single mode can be active at a particular logical time instant, while activity in other modes is automatically suspended. Transitioning between modes switches the reactor's behavior. While the previous example only shows the default behavior of transitions, there are actually two different options to control the starting point of an entered mode. A mode can either be *reset*, or it may continue with the mode's *history*. These are two common and powerful abstractions that are particularly helpful in the automatic management of timed behaviors, which can be extremely error-prone when carried out manually.

The research question investigated here is: How can we enhance reactors with a lean concept that enables the coordination of reactive tasks based on modes of operation? The resulting design should align with core principles of LF, such as the black-box abstraction of reactions that enables a polyglot design. Adapting such an approach involves a trade-off because it comes at a cost of decreased analyzability if the actual triggering of transitions is hidden. Generally, the concept aims for a potential "best of both worlds" situation that enables modeling state-oriented behavior seamlessly in an actor-oriented language. The multitude of existing languages for expressing modal- or state-oriented behavior illustrates that this entails a number of language design questions and trade-offs. For example, this includes selecting a suitable set of transitions, evaluating potential preemption mechanisms, integrating transition timing into the model of super-dense time, and technical considerations for implementing resets and harmonizing them with the existing way of memory management for state variables at startup and shutdown.

3.3.1 Modes and Transitions

Modes associate elements inside a reactor with a specific mode of operation, while transitions control the change of activity and the effect on the entered modes.

Mode Syntax Modes can be defined in any regular reactor.[4] Each mode requires a unique (per reactor) name and can declare contents that are local to this mode. There must be exactly one mode marked as `initial`, see line 12 in Listing 3.1. A mode can contain state variables, timers, actions, reactions, reactor instantiations, and connections. This excludes the declaration of ports in modes. Instead, modes have access to the scope of their parent reactor, which enables references to ports, state variables, and parameters declared on the reactor level. Inner declarations of other modes are not accessible. While modes cannot be nested in other modes directly, hierarchical

[4]The only exception is a federated reactor. The current concept does not account for modes with a state that needs to be synchronized across a federation. Yet, federates themself can be modal reactors. Investigating a modal coordination of a federation is considered future work, see Section 6.2.1.

and concurrent composition is possible through the instantiation of modal reactors.

Mode Activity In the presence of modes, only parts that are contained in the currently active mode, or outside any mode, are executed at any point in time. This also holds for parts that are nested in multiple ancestor modes due to hierarchy. Consequently, all those ancestors must be active in order to execute. Upon reactor startup, the initial mode of each modal reactor is active, others are inactive. Reactions in inactive modes are simply not executed. All components that model timing behavior, namely timers, scheduled actions, and delayed connections, are subject to a concept of *local time*. That means while a mode is inactive, the progress of time is suspended locally. Section 3.3.2 will provide a more detailed explanation. How the timing components behave when a mode becomes active depends on the transition type.

Transition Syntax Transitions are declared within reactions. If a reactor has modes, reactions can list them as effects if they intend to invoke transitions. This enables the use of the target language API to set the next mode, e. g., `lf_set_mode` in C, see line 16 in Listing 3.1. If the target code references a mode that is not declared as an effect, the compiler will issue an error. The user also needs to specify the type of the transition by adding the modifier `reset` or `history` to the effect. History transitions are indicated by an "H" at the arrowhead in the diagram, see Figure 3.5. In case a mode has no actual state, i. e., only consists of reactions, the modifier can be omitted because there is no effective difference between the two transitions types into this specific mode.

Transition Timing A transition is triggered if a new mode is set in a reaction body. This raises two questions: When will the transition take effect, and what if multiple reactions set different modes? SCCharts illustrate that there can be immediate and delayed transitions, see Section 2.3.

Immediate transitions would allow mode changes to occur directly after executing the initiating reaction. The target mode would be instantly activated and its reactions executed at the same tag as the reaction that issued the transition. This implies dependencies between reactions with transitions

and all reactions associated with their target modes. As a consequence, any cyclic modes structure would impose a causality loop and would have to be rejected. Alternatively, transitions could wait until all contents of a mode finish executing but then immediately switch to the next mode and execute that one, still at the same tag. However, this would raise the question of how to handle the reactivation of the same mode multiple times at the same tag. Moreover, this would permit an arbitrary number of mode changes during the same execution instant. In the end, some notion of a sequential separation between mode activations would be necessary. Furthermore, the reactor MoC is simply not intended to execute reaction multiple times at the same tag. The evident solution is to implement delayed transitions that introduce at least a microstep delay.

Hence, in modal reactors, reactions can set a new mode, but this has no immediate effect. Only when the reactor has finished executing all its contents, the transition will take effect and the new mode becomes active in the next microstep. Hence, no two modes in the same reactor can be active at the same tag. Neither can a transition interfere with ongoing reactions. The same principle for mutual exclusion of modes can be found in SCADE [CPP05; CHP06]. Yet, this approach requires resolving potentially "conflicting" transition effects from different reactions. To resolves such situations, the same mechanism applies as in setting ports. The fixed ordering of reactions determines the effective target mode that will be used. In terms of deterministic outcome and overriding behavior, setting new modes can be considered analogous to assigning output ports. However, in terms of timing, transition effects correspond to scheduling actions with a zero delay, which also enforces a microstep delay to prevent causality cycles.

To be precise, if at a tag (t, m), a new mode was set by a reaction in a modal reactor, the execution of that tick will finish unchanged. Only at the end of that instant, the target mode will be determined by the last reaction that set a mode, the current mode will be deactivated, and the new one will be activated for future execution. This means no reaction of the newly active mode will execute at tag (t, m). Instead, the earliest possible reaction in the new mode occurs one microstep later, at $(t, m + 1)$. If the newly active mode has for example a timer that will elapse with an offset of zero, it will trigger at $(t, m + 1)$. In case the mode itself does not require an immediate

execution in the next microstep, the next executed tag depends on future events $(t + e, 0)$, with e as the time offset to the next event, just as in the normal behavior of LF. Thus, modes in the same reactor are always mutually exclusive w.r.t. superdense time.

Reset Transitions A mode can be reset upon entry, returning it to its initial state. Specifically, this has the following effects:

- all contained modal reactors are reset to their initial mode (recursively);
- all contained timers are reset and start again awaiting their initial offset;
- all events (actions, timers, delayed connections) that were previously scheduled from within this mode are discarded;
- all contained state variables that are marked for automatic reset are reset to their initial value; and
- all contained reactions with the new reset trigger are executed.

Note that "contained" refers to all contents defined locally in the mode and in local reactor instances (recursively) that are not otherwise enclosed in modes of lower levels.

These effects ensure that whenever a mode is entered with a reset transition, the subsequent timing of behavior is as if the mode was never executed before. Furthermore, state variables are not reset automatically by default. Instead, they need to be marked for reset explicitly because it is idiomatic for reactors to store manually managed resources, such as allocated memory, in state variables. Hence, an automatic reset could easily lead to memory leaks or runtime exceptions. Section 3.3.3 will provide a more detailed example of manual resource management. To provide manual control over resetting state variables, a new built-in reset trigger for reactions is provided that enables reacting to a reset entry of a mode. The reset modifier on state variables is a convenience feature that automatically resets state variables to their initial value, e. g., used for led_state in line 22 of Listing 3.1.

History Transitions In contrast to a reset transition, a history transition will "simply" skip resetting a mode's contents. This enables the mode to continue its behavior from the point it was last left, or its initial configuration

if it was not active yet. In regard to the temporal behavior, the time is frozen during mode inactivity. This requires adjusting all events that originate from timers, scheduled actions, and delayed connections. During mode inactivity these events are suspended and will not be present at the intended tag. Upon continuing the mode, their remaining delay is adjusted such that it reflects the remaining delay recorded at the instant the mode was previously left. This results in a notion of local time that elapses only when the mode is active.

Diagrams As Section 3.2 illustrates, the extension to modal reactors also includes a graphical notation to provide an intuitive perception of the modal structures.

3.3.2 Local Time

The notion of mode-local time suspends all timing behavior within inactive modes. This is an established and well-formed principle also found in modal models in Ptolemy II [LT10; Pto14] and synchronous languages such as Esterel or SCCharts. The considerations by Lee et al. that favor local time over alternative approaches also apply to LF. The suspension of time gives a clear and consistent meaning to the inactivity of modes and provides comprehensible state of the mode's contents upon entry. This especially favors modularity, as reactors that may be instantiated in modes do not have to anticipate the fact that their time (driven by timers or scheduled actions) will advance while their reactions are suppressed. Furthermore, modes allow defining reactor elements outside of modes, which gives the developer control over which elements should be subject to local time. The timer in the `ToggleLED` reactor in Figure 3.4 is one example for this.

Example Figure 3.5 shows an LF model that illustrates the different characteristics of local time affecting timers and actions in the presence of the two transition types. It consists of two modes `One` (the initial one) and `Two`, both in the `Modal` reactor. The `next` input toggles between these modes. The input port is controlled by a reaction on the top level that is triggered by the timer `T`. After one second, the mode switch is triggered periodically. Both modes' contents are structured identically. Each has a timer `T1`/`T2` that triggers a

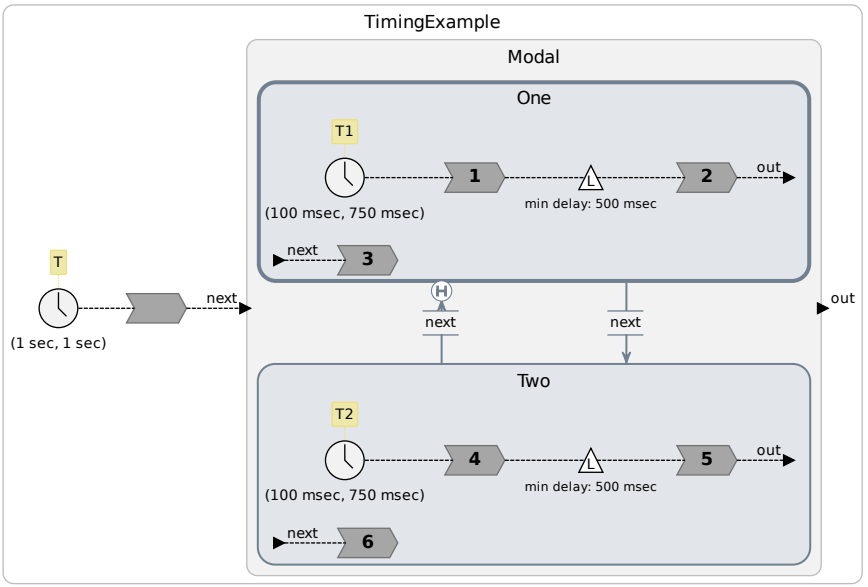


Figure 3.5. The TimingExample model for illustrating the different effects of reset and history transitions on timers and actions in modes. (Publ. in [SHL+23c])

reaction after an initial offset of 100 msec and then periodically after 750 msec. This reaction then schedules a logical action with a delay of 500 msec (the actual target code does not add an additional delay upon the minimum specified in the model). This action triggers the second reaction that writes the output out. The last reaction is triggered by the input next and invokes the transition to the other state. The main difference between the modes is that One is entered via a history transition, continuing its behavior, while Two is reset.

Execution Trace Figure 3.6 illustrates the execution trace of the first 4 seconds of this program. Above the timeline are the model elements that are executed at certain points in time, together with arrows indicating triggering relations and dashed lines for distribution through time. On top of that is the currently active mode.

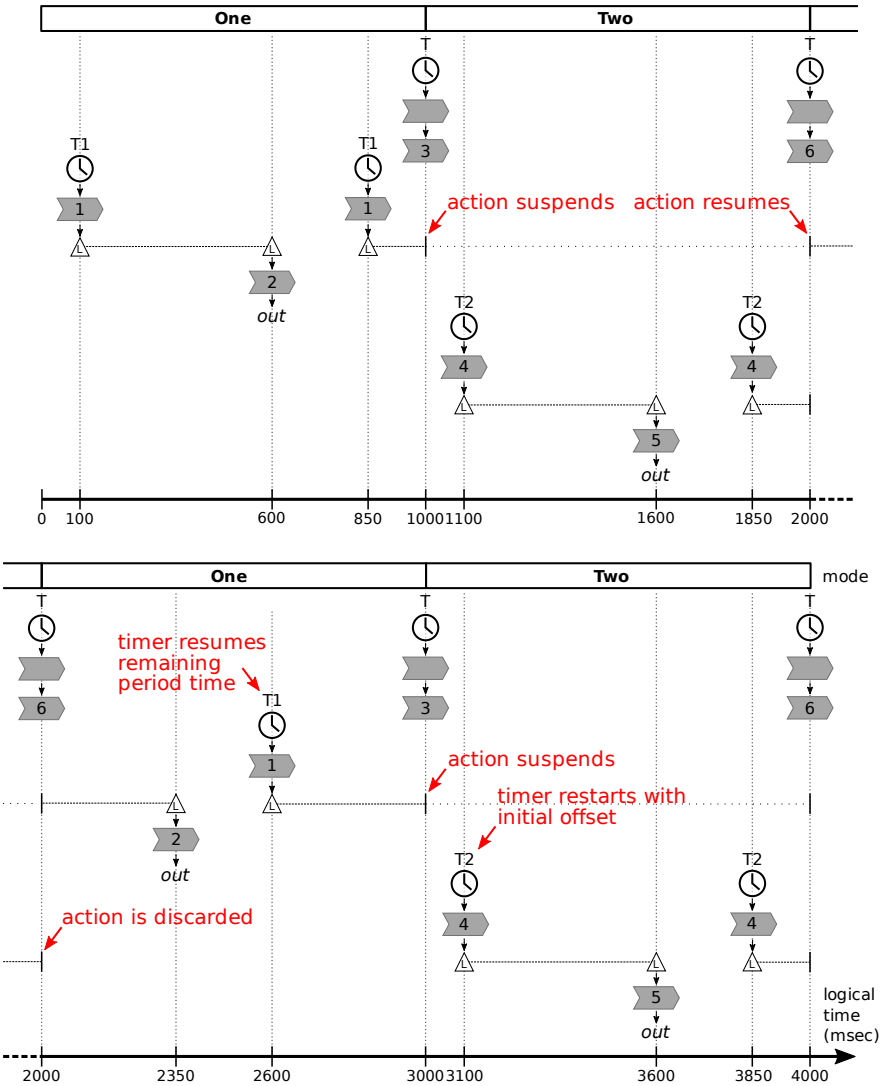


Figure 3.6. The execution trace with reaction illustration for the TimingExample model in Figure 3.5. (Publ. in [SHL+23c])

At 100 msec the initial offset of timer T1 elapses, which leads to the scheduling of the logical action in this mode. The action triggers the reaction 500 msec later at 600 msec and thus causes an output. At 850 msec the first 750 msec period of T1 elapses and again invokes its chain of effects. However, this time at 1000 msec the timer T causes a mode switch and mode One is rendered inactive. For the scheduled action, this means that its event (scheduled for 1350 msec) are suspended and will no longer trigger. The same holds for timer T1.

The reset transition has no real effect on Two as the mode was never active before. Hence, mode Two starts with the initial offset of T2 triggering at 1100 msec. The following sequence of events is identical to the one that mode One produced.

At 2000 msec the Modal reactor switches back to mode One with a history transition. The event of the action in Two is discarded because this mode is only entered via a reset and no suspension of events is necessary. In general, it could be conservatively suspended and then discarded when the mode is entered via reset instead of history, but this is not necessary in this case. The event of the logical action in mode One resumes. Since it was scheduled at 850 msec with a 500 msec delay and a 1000 msec inactivity of this mode, it is reintroduced for occurring at 2350 msec. Hence, the delay is kept relative to the time passing local to the mode. The same holds for the timer T1, whose second period would have elapsed at 1600 msec without mode inactivity, now it triggers at 2600 msec.

After 3 seconds the next mode switch happens, which again suspends the event of the action in One. However, mode Two is reset upon entry, which effectively puts the mode in a state as if it has never run before. No events resume and timers restart with their initial offset. Hence, T2 triggers at 3100 msec and the sequence of events local to mode Two unravel in the same way as the first time.

Time Progression Figure 3.7 illustrates the relation between global time in the environment and the localized time for each timer in Figure 3.5. Since the top-level reactor TimingExample is not enclosed by any mode, its time always corresponds to the global time. Mode One is the initial mode and hence progresses in sync with TimingExample for the first second. During

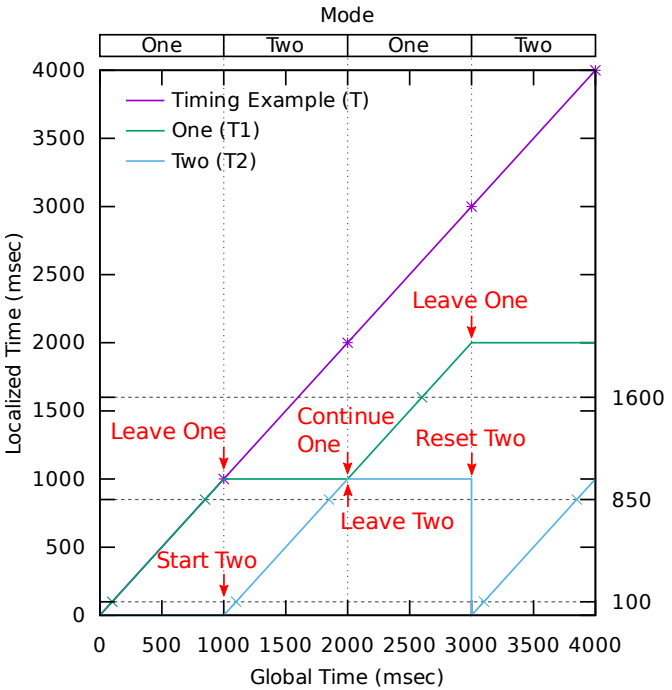


Figure 3.7. The progression of time in each mode and their respective timer of the TimingExample model in Figure 3.5. (Publ. in [SHL+23c])

inactivity of mode One the timer is suspended and does not advance in time. At 2000 msec it continues relative to this time. T2 only starts advancing when the mode becomes active at 1000 msec. The reentry via reset at 3000 msec causes the local time to be reset to zero.

This example illustrates that from the perspective of timers and actions, time does not advance during mode inactivity. This also applies to indirectly nested reactors, if instantiated inside a mode. In the same way, delayed connections are affected by local time, if their source lies within a mode. This corresponds to the fact that delayed connections can be considered syntactic sugar for connections delayed by a logical action.

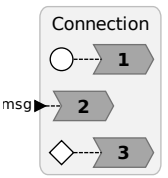


Figure 3.8. The Connection reactor.

```
1  target C;
2  reactor Connection {
3    input msg: char*;
4    state socket: int;
5
6    reaction(startup) {=
7      self->socket = socket(AF_INET, SOCK_STREAM, 0);
8      struct sockaddr_in server = {/*Connection configuation*/};
9      connect(self->socket, (struct sockaddr*)&server, sizeof(server));
10   =}
11   reaction(msg) {=
12     send(self->socket, msg->value, strlen(msg->value), 0);
13   =}
14   reaction(shutdown) {=
15     close(self->socket);
16   =}
17 }
```

Listing 3.3. Source code of the Connection reactor illustrating manual management of a socket connection.

3.3.3 Startup and Shutdown

Section 3.3.1 already mentions that state variables cannot be reset by default because they might store manually managed resources. Hence, the reset trigger was introduced. This new trigger comes in addition to the two built-in triggers, startup and shutdown, that are defined by the reactor MoC [LÍG+19]. Reactions with a startup trigger will be executed at the very first tag at which their reactor exists. Symmetrically, shutdown reactions will execute during the very last tag before the reactor ceases to exist. In most programs this corresponds to the start and end of the program. Only if mutations are used, the association with a reactor's lifetime becomes relevant.

These two triggers are commonly used for initializing and finalizing manually managed resources, such as allocating memory, sensor or actuator connections, or threads for handling asynchronous interactions. Figure 3.8 illustrates a reactor that manages a socket connection. The circle indicates a startup trigger, while the diamond represents shutdown. Listing 3.3 presents the corresponding source code. The startup reaction (lines 6 to 10) creates

a socket, stores it in a state variable, and connects it to some address. The second reaction sends a message received on the input port via the socket (line 12) and the shutdown reaction closes the socket connection (line 15). The code omits the actual address configuration for the socket, checks on return codes, and the relevant includes, to keep the example simple.

While the behavior is relatively clear with respect to the reactor, the question is: How does this reactor behave when instantiated in a mode? More specifically: When are the startup and shutdown reactions triggered? This question corresponds to defining the lifetime of mode-local elements.

Considerations on the Lifetime of Modes One solution could be binding the lifetime to the modes' activity and triggering `startup` and `shutdown` upon entering and leaving a mode. This would also render the `reset` trigger redundant. However, in case of a history transition, the mode has to continue its behavior, requiring the previous configuration and state. And in the general case with both reset and history transitions to a mode, this can only be decided upon re-entry. Moreover, there is the challenging question of finding the right time for executing shutdown reactions upon leaving because the decision for leaving a mode is only final at the end of execution, see Section 3.3.1. Consequently, shutdown reactions cannot simply be invoked when leaving a mode.

The next best alternative is to associate mode-local elements with the reactor's lifetime. However, based on the current definition, modal reactors suppress any behavior in inactive modes, which would include reactions triggered by `startup` and `shutdown`. Hence, only initial modes would execute their startup reactions (depending on the nesting of modes), while other startup reactions would be skipped. The same applies to shutdown reactions in inactive modes at the time of a shutdown. A possible solution is to exclude startup and shutdown reactions from this suppression mechanism. However, these are reactions with arbitrary effects that now would circumvent the mutual exclusion of modes at start and end of a reactor's lifetime.

Proposed Behavior While several other options and variations were considered to solve this issue, there was no fully satisfying solution. The proposed concept is a compromise that

- tries to minimize behavioral oddities in practical applications,
- enables embedding reactor instances with startup and shutdown reactions into modes without further adjustment, and
- provides intuitive and deterministic behavior.

Startup reactions are executed at first activation of a mode. When the reactor is shut down, it triggers all shutdown reactions at the same time, but only those which modes were at least once active, i. e., had a potential startup reaction. Hence, the lifetime of reactor elements in modes, especially reactor instances, starts with the first activation of the mode and ends with the shutdown of the reactor.

This design is a result of application-oriented considerations. Startup reactions are more common, also for tasks other than resource management, and are more likely to have effects than shutdown reactions. Therefore, preventing their execution from bypassing mutual exclusion, as in shutdown, hopefully reduces the noticeable effects of this design. The possible improvement of this behavior is considered future work but may require a more invasive redesign of resource management that in parts replaces startup and shutdown reactions, see Section 6.2.1.

3.3.4 Implementation

The goal of creating a lean mode extension is not only reflected in the language design itself but also in its runtime implementation. At the time of writing this thesis, modes are implemented for C and Python, but extending support for modal reactors to other targets is already planned for future development. Nonetheless, modes have been successfully tested for these targets with single-threaded and multi-threaded execution, and inside reactors of a federation. This section provides a generic and reasonably target language-independent view on the adjustments required to support modes, achieving the behavior described in Section 3.3.1.

Extending the LF Compilation and Runtime As initially presented in Section 2.2.1, generated LF programs consist of two parts, the program-specific code and the generic runtime engine. The adjustments necessary

for extending both parts towards modes are relatively small. Moreover, they are mostly additive, which means that in the absence of modes in the source model, there is virtually no difference to an implementation that does not account for modes, and thus no performance overhead.

The existing runtime implementation is adjusted in two ways. First, the triggering of reactions must check if a reaction is in an active mode and otherwise prevent its execution. A trivial approach is to recursively check a mode and all its parent modes whether they are all currently active modes. Any element associated with no mode at all is considered always active. Shutdown reactions are excluded from this activity check but their mode (if any) must have had a startup phase, which is additionally recorded when executing these reactions. Second, the execution life-cycle requires the handling of transitions. It must be invoked after the processing of reactions has finished but before logical time advances. Algorithm 3.1 presents the procedure that handles mode transitions, which includes performing resets, managing local time, and scheduling special triggers.

New Data Structures The algorithm relies on a few global data structures: (1) the existing event queue of LF (EventQueue), the manipulation of which is the sole change to the runtime that is needed to implement local time; (2) a collection to store events suspended in local time (SuspendedEvents); and (3) a set of all modal reactor instances in the model (ModalReactors). The latter is a result of program-specific generated code, which also produces new data structures and references for modes. Each modal reactor r provides access to the following information, presented in a member notation here. r_{modes} denotes the set of modes in r.

r.parentMode $\in \{\text{Nil}\} \cup \{x_{modes} : x \in \text{ModalReactors}\}$
r.initialMode $\in r_{modes}$
r.currentMode $\in r_{modes}$
r.nextMode $\in \{\text{Nil}\} \cup r_{modes}$
r.transition $\in \{\text{None, Reset, History}\}$

r.parentMode is either absent or the mode immediately containing r. Note that this models only a unidirectional relation for mode hierarchy. While one could also consider introducing a list of contained modal reactors, this notation is closer to the actual implementation. r.initialMode is the

mandatory initial mode. As the parent mode, it is constant and set up at program start. r.currentMode is the currently active mode w.r.t. to r, starting with the initial mode. Whether the current mode is actually active w.r.t. execution depends additionally on parent modes. r.nextMode and r.transition represent the presence, type, and target of a transition. These fields are filled if the target code sets a new mode, e. g., `lf_set_mode(Catch)`. The transition type is inferred from the effect definition. Furthermore, each mode m in r_{modes} carries additional mode-specific information.

m.reactor $\in$ ModalReactors
m.leaveTime $\in \mathbb{T}$
m.reset $\in \{\text{True}, \text{False}\}$
m.hadStartup $\in \{\text{True}, \text{False}\}$

m.reactor is a constant reference to the mode's reactor. m.leaveTime stores the logical time at which this mode was last left. It is initialized with the start time of the execution. m.reset indicates that this mode needs to be reset as soon as it becomes active (initially False). m.hadStartup is a boolean flag that is set from False to True as soon as the mode is active for the first time.

Finally, each reaction, timer, and action has a reference to its immediately enclosing mode, if any exists. This also enables associating events with modes via their trigger.

The Algorithm In the first line of Algorithm 3.1, every modal reactor instance is processed in a top-down order. This refers to the partial order of mode hierarchy and ensures that if a mode is entered with a reset, inner modal reactors (line 2) are recursively forced into their initial mode via a reset transition. Afterwards, transitions are processed in a separate iteration. This iteration is separated from the previous iteration because the hierarchical reset relies on the presence of transition information in parent modes to reset itself accordingly (line 2) and this information is now overwritten (line 22). First, events of the next mode that are suspended in time are processed. At a reset, all timers are restarted with the initial offset relative to the current time (line 12). Other events (e. g., scheduled actions) are dropped. For reintroducing previously suspended events into the event queue, the shift function is used to create the correct tag w.r.t. superdense time.

```
1:  for each r ∈ topdown(ModalReactors) do
2:    if r.parentMode ≠ Nil and r.parentMode.reactor.transition = Reset then
3:      r.nextMode := r.initalMode                          ▷ Hierarchical reset
4:      r.transition := Reset
5:  for each r ∈ ModalReactors do
6:    if r.transition ≠ None then
7:      for each e ∈ SuspendedEvents do                     ▷ Handle suspended events
8:        if e.mode = r.nextMode then
9:          Remove e from SuspendedEvents
10:         if r.transition = Reset then
11:           if e is Timer then                            ▷ Reset timers, discard other events
12:             t := shift(currentLogicalTime, e.timer.offset)
13:             Insert e into EventQueue with tag t
14:         else                                            ▷ Resume events adjusted to local time
15:           t := shift(currentLogicalTime, e.tag - e.mode.leaveTime)
16:           Insert e into EventQueue with tag t
17:     if r.transition = Reset then                        ▷ Perform transition
18:       r.nextMode.reset := True
19:     r.currentMode.leaveTime := currentLogicalTime
20:     r.currentMode := r.nextMode
21:     r.nextMode := Nil
22:     r.transition := None
23:   if isActive(e.currentMode) then                       ▷ Trigger special reactions
24:     if not r.currentMode.hadStartup then
25:       Trigger startup reactions in r.currentMode at the next microstep
26:     if r.currentMode.reset then
27:       r.currentMode.reset := False
28:       Trigger reset reactions in r.currentMode at the next microstep
29:       Reset state variables in r.currentMode that are marked with reset
30: for each e ∈ EventQueue do
31:   if not isActive(e.mode) then                          ▷ Suspend events in now inactive modes
32:     Remove e from EventQueue
33:     Add e to SuspendedEvents
```

Algorithm 3.1. Processing of mode transitions at the end of each execution cycle.

$$\text{shift}(\text{base} : (t, m), \text{offset} : (t, m)) = \begin{cases} \text{offset}_t > 0 : & (\text{base}_t + \text{offset}_t, \text{offset}_m) \\ \text{offset}_t = 0 : & (\text{base}_t, \text{base}_m + \text{offset}_m + 1) \end{cases}$$

This creates a tag that is the base tag shifted by a given offset into the future. It takes into account that a zero delay offset (w.r.t. the timestamp t) results in a future (incremented) microstep. In case time should continue due to a history transition, all events are reintroduced into the event queue with an adjusted target time. Here, the time that mode was left is subtracted from the original tag (time to happen) of the event, to get the remaining time at time of leaving, which is used to offset this event from the current time (line 15). Next, the actual effect of the transitions is applied to the internal data structures (lines 17 to 22). This includes marking the mode for reset if necessary, storing the time the mode was left, setting the new mode, and clearing transition information for use in future execution. Afterwards, the special reactions are triggered for the current mode. Note that this takes effect based on mode activity and not triggered by a transition (line 23). The isActive function relies on the definition presented before. If the mode was never active before, its startup reaction will be triggered at the next microstep. This includes reactions in the mode and in all inner non-modal reactors. Likewise, reset reactions are triggered when the mode is marked for a reset. Additionally, the automatic reset for the respective state variables is invoked, and the flag is cleared. At the end, all events that are associated with now inactive modes are pulled from the event queue and stored in the suspended events collection (lines 30 to 33).

In the real implementation of Algorithm 3.1, the procedure includes some additional consistency checks and optimizations, e. g., a non-recursive isActive implementation. However, as these specifics are not relevant for the overall semantics, they are omitted for ease of readability. Overall, the algorithm is considerably compact while providing the bulk of semantic functionality for modes. The remaining behavior, such as suppression of inactive reaction and triggering of shutdown, is directly integrated into the normal LF runtime and uses the same mode-specific data structures.

3.4 Evaluation

The modal pendulum controller presented in Section 3.2 already illustrates that modal reactors are capable of expressing different modes of operation in a program and elevate a low-level state machine implementation onto the coordination level. Moreover, the encapsulation of timed elements and the concept of local time enables the correct alignment and control over time in modes, see Figure 3.1.

Reflection on Goals Considering the initially set goals, the question is: Have these goals been met? (cf. page 55)

Lean design The additional syntax required for defining modes is rather minimal. It consists only of declaring modes as encapsulation units, transitions as reaction effects, and a new trigger for handling reset. The core language remains unchanged and none of the new elements break or disrupt the existing modeling paradigms in LF. Simultaneously, the synthesized diagrams provide a more coherent view in a classical statecharts notation. Furthermore, with reset and history transitions, modal reactors have two basic but powerful ways to control the effect of mode changes. Their behavior cannot easily be achieved by other means due to their effect on temporal behavior, see also Section 3.4.1. Yet, languages such as SCCharts provide a broader and more versatile set of transition types. However, as Section 3.4.2 will present, some of their functionality could be obtained in modal reactors by adjusting the reaction implementation accordingly.

Polyglotism Modal reactors fully embrace the embedded target language approach and black-box abstraction in LF. Hence, they do not obstruct polyglot designs in LF. While the declaration of transition effects in reaction signatures is sufficient to have a modal code generation, diagrams, and static mutual exclusion (see Section 3.4.1), it imposes a limitation in terms of analyzability and verification. As it turns out, this is a more general issue in LF. By design, reactions do not expose detailed causal relations between their triggers and effects, which hampers in-depth model checking. Mode transitions likewise are subject to this limitation, which should be addressed in future work, see Section 6.2.2.

Time sensitivity The concept of mode-local time offers a well-defined and consistent way to deal with time during mode inactivity and reentry. This design of "freezing" time favors composability, since reactors instantiated in modes retain their temporal behavior relative to their context [LT10]. It is also a well-established principle in Ptolemy II [LT10; Pto14] and in synchronous languages, which often provide a suspend feature that conceptually cuts off the clock signal for certain program sections [PEB07].

Concurrency Concurrent and hierarchical composition comes naturally to modal reactors, as they simply augment the existing reactor model without introducing additional burdens or dependencies in this regard. Modes are exclusively scoped to their reactor and thus facilitate composability.

Determinism LF provides determinism and the modal extension retains this property in all aspects. Temporal and causal behavior of transition is designed to prevent potentially problematic inter-reaction dependencies. Instead, the explicit modeling of mutual exclusion with modes enables accepting programs otherwise conservatively rejected as potentially non-deterministic, see Section 3.4.1.

3.4.1 New Modeling Opportunities

The introduction of modal reactors clearly expands the modeling capabilities within LF. The modal pendulum controller in Section 3.2 is only very limited example for this. It illustrates the separation of a reaction into modes and the improvement of temporal behavior by association with modes. However, the opportunities that modes offer in terms of modeling go beyond this example.

Local Time While Section 3.3.2 illustrates the different effects of local time, it does not yet fully explore the significant effect that this concept can have on the temporal modeling capabilities of LF. Consider the very simple case of pausing an application. Such a use case appeared while extending the Furuta pendulum example into a more full-fledged demonstrator. Figure 3.9

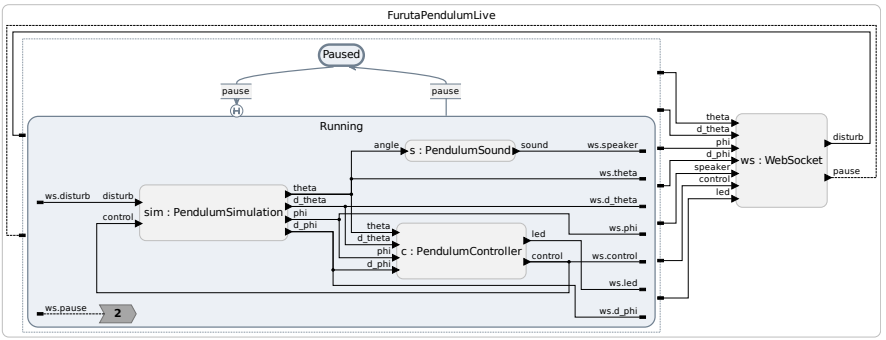
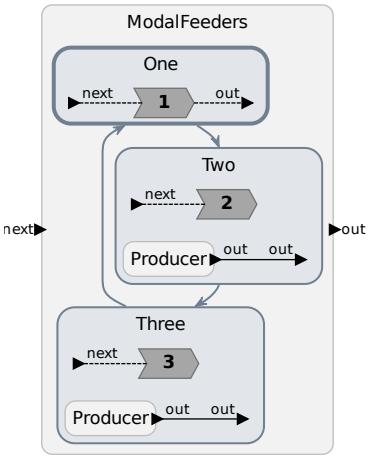


Figure 3.9. The control components of the Furuta pendulum in a simulation setup connected to a user interface that supports pausing.

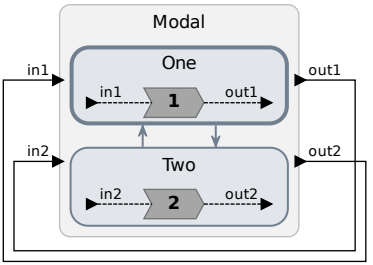
illustrates a program that enables observing a simulation of the pendulum while running. Without going into much detail about the implementation, it communicates via a WebSocket[5] connection with a web-based user interface that displays the state of the pendulum as 3D model (Figure 1.1 is a snapshot of this visualization). The user can interact with the live system by imposing an external disturbance to the pendulum or pausing and continuing the simulation. In this example, modes are used to implement the effect of pausing by simply switching between a `Paused` and `Running` mode. The `Running` mode contains all the reactor instances relevant for the simulation and continues their behavior with a history transition upon entry.

While this model relies on modes and their notion of local time to solve the rather simple task of pausing, achieving the same behavior with non-modal LF is quite cumbersome in comparison. It would require adjusting all reactors inside the `Running` mode that have timing elements. They all would require a new input for pausing, a state variable that remembers the pause status, and all reactions need to stop producing outputs when paused. Since their local time would continue to run, timers and actions would still elapse and trigger reactions. For actions in `PendulumSound`, this would require some bookkeeping to discard invalidated events and reschedule actions

[5] `https://datatracker.ietf.org/doc/html/rfc6455`



(a) Multiple feeders (reaction and two reactors) to the same output port (out) but separated by modes.



(b) A cyclic dependency resolved by the use of modes.

Figure 3.10. Two examples for LF models that can be accepted as deterministic/causal due to the use of modes.

accordingly upon unpausing. Timers, as in PendulumController, could most certainly no longer be used as they do not offer control over their temporal behavior at runtime and would need to replaced by actions.

All in all, this example illustrates the efficient temporal modeling capabilities that come with modes and their notion of local time.

Static Analysis The basic principle of modes is to separate reactor elements into mutually exclusive modes. The pendulum example in its non-modal form illustrates that the same behavior can be achieved without modes on the coordination level. However, the explicit presence of modes enables more advanced structural analyses of LF programs. This lifts certain modeling restrictions imposed under the standard LF MoC and allows accepting more programs. Figure 3.10 illustrates two such examples.

In the absence of modes, an output port can never be fed by multiple connections to reactors or a mix of reactions and reactors, as this is a potential source of non-determinism. Only reactions have an intrinsic ordering, while reactors are inherently concurrent. However, in a model that locates these writers in separate modes, the compiler has access to this additional information. It can accept such structures, provided that all writers are mutually exclusive. Figure 3.10a illustrates such a situation. Both the first reaction and the two instances of the `Producer` write to the `out` port. The modal structure statically ensures that these writes will not happen currently at runtime.

The design of modes and timing of transitions guarantee this property in all cases, except for shutdown reactions, see Section 3.3.3. Hence, these are exempted from this adjustment to the static analysis and are handled as before by the LF compiler.

Furthermore, the same principle applies to causality problems imposed by feedback loops. In this case, the use of modes enables an advanced dependency analysis that takes mutual exclusion into account when detecting cycles. Figure 3.10b illustrates a model that would be rejected in the absence of modes. If both reactions could be active at the same time, it would constitute a causal cycle: `in1` – `out1` – `in2` – `out2` – `in1`.

Hence, the structural information provided by modes enable inferring additional static information about the program that can be used to reduce the conservative over-approximation by the compiler and accept more programs. Yet, a full formal analysis of modal behavior remains future work, see Section 6.2.2.

3.4.2 Feature Comparison with Statecharts

Guided by the goal of creating a lean modal extension, modal reactors only provide a minimal set of built-in features. In comparison, statecharts languages offer a much broader range of language constructs and transition behavior. SCCharts are a prime example of such a design, as they combine many features of common synchronous languages. While the fundamental concepts of states and transitions are the same, many advanced aspects

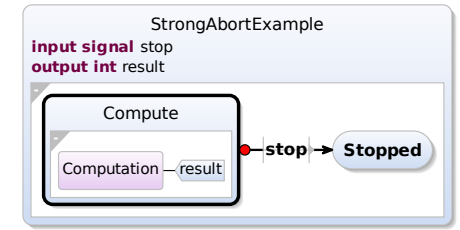
were considered during the design of modes. Hence, some features are implicitly present in modes, while others are deliberately excluded in favor of simplicity. Nonetheless, LF can also utilize the target code in reactions to implement advanced behavior.

Hybrid Modeling A hybrid modeling approach that combines a dataflow notation and statecharts is common in many modeling languages. SCCharts, SCADE, and Ptolemy II all provide such design capabilities, see Section 3.1.1; and likewise does LF.

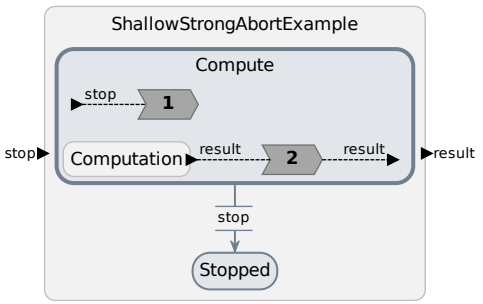
However, an interesting subtlety in the design of LF is its incorporation of pragmatics-aware modeling, discussed in Section 2.5. Other languages follow a rather strict syntactic separation between the dataflow and statecharts domain. Ptolemy II is a prominent example of this layered design. In LF, this separation is more distinct in the diagrams than the textual source. The concept of views facilitates a language design that seamlessly integrates modal structures into the textual syntax of reactors, while preserving a graphical notation that explicitly expresses the state machine nature of modes.

Preemption In synchronous languages, transitions typically apply a form of *preemption* [Ber93] when a state is left. In case of SCCharts, there are strong and weak aborts, see Section 2.3. While the strong variant preempts all inner behavior of the left state, the weak variant grants a "last wish" before leaving.

Technically, transitions in modal reactors always perform a weak preemption, since they allow all inner behavior to execute in the current tick and deactivate the mode afterwards. The main reason for this design is that transitions are triggered by reactions that are inner behavior of the mode. As discussed in Section 3.3.1 on the timing of transitions, mode changes that immediately preempt other behavior are complicated in this context and impose additional dependencies with a negative impact on the parallelization potential. Such problems are also present in other synchronous languages. In SyncCharts, for example, it constitutes a causality error if a state emits a signal that would strongly preempt that state. SCCharts do not have this issue, as they consider strong abort checks ordered sequentially before the execution of inner behavior.



(a) SCChart



(b) LF reactor

Figure 3.11. An example for a strong abort in SCCharts and a similar but shallow implementation in LF.

```
1  target C;
2  import Computation from
       "Computation.lf";
3  reactor ShallowStrongAbortExample {
4    input stop: bool;
5    output result: int;
6
7    initial mode Compute {
8      reset state abort: bool = false;
9      c = new Computation();
10
11     reaction(stop) -> Stopped {=
12       lf_set_mode(Stopped);
13       self->abort = true;
14     =}
15     reaction(c.result) -> result {=
16       if (!self->abort) {
17         lf_set(result, c.result->value);
18       }
19     =}
20   }
21   mode Stopped {}
22 }
```

Listing 3.4. Source code of the ShallowStrongAbortExample reactor.

While the proposed modal reactors model does not include preemption, there were considerations for including this feature, for example in the form of special "initial" reactions that would be executed before any other reactor elements to determine and suppress preempted content. However, such a design was deemed expendable in the face of a lean language design and the fact that LF already offers many ways to influence the transition triggering in reaction bodies by using target language capabilities.

Figure 3.11 illustrates a small example modeling a strong abort in SCCharts and a variation in LF that emulates a form of shallow preemption. Listing 3.4 presents the source code of the reactor. The first reaction sets the abort variable (line 13) to prevent an effect on the result output in the

second reaction (line 16). Admittedly, this exemplifies only a preemption of the behavior of the second reaction but not a true hierarchical preemption of the internal behavior in the Computation reactor. It rather suppresses the observable effect of the behavior in Computation and, hence, could be considered a shallow abort. Nonetheless, it illustrates the general procedure. To explicitly abort the internals of Computation, the reactor would require an additional input to pass on the abort variable downstream as an event and react accordingly inside the Computation reactor.

However, the LF community has not yet requested a more convenient built-in strong preemption feature and, as of yet, there are no use cases that justify an introduction.

Termination SCCharts and SyncCharts also feature the non-preemptive *termination* transition type, see Section 2.3, that is enabled when all inner regions reached a final state. It corresponds to joining one or more spawned threads. However, termination is a control-flow concept that does not make much sense in a dataflow language that does not model concurrency as explicit threads or that does not provide built-in constructs that indicate termination. Modal reactors could give an opportunity to introduce such a feature, but the proposed concept simply embraces the dataflow in LF and keeps the extension lean.

Priorities With the use of reactions as transition triggers comes another difference in comparison to SCCharts and other statecharts dialects. They usually use *priorities* to assign an order to available transitions, with the first enabled transition preempting lower ones. While the preemption aspect was already discussed, modes in LF have exactly the inverse behavior, where the last invocation of lf_set_mode in reaction order determines the actual transition. An implementation for setting modes in reactions that favors the first writer could be easily achieved, but the proposed design favors the analogy to setting output ports.

Immediate and Delayed A timing aspect, predominately present in synchronous languages, is the distinctions between *immediate* and *delayed* transitions, see Section 2.3. Section 3.3.1 already explained arguments on why immediate transitions are not supported in this proposal for modal reactors.

Deferred Some languages, such as SCCharts or SCADE, also provide *deferred* transitions. They suppress the immediate behavior of an entered state. This concept assumes that transitions are instantly processed and, hence, represent a way to enter a state delayed by one tick. Considering the microstep delay for transitions in modal reactors, as discussed in Section 3.3.1, one might argue that these transitions are always deferred. This is also illustrated by the fact that the SCCharts variant of the PendulumController uses a deferred transition to produce a behavior equivalent to the LF model, see Section 1.1.2.

Reset and History While reset and history transitions in modal reactors are equivalent to statecharts, SCCharts feature a further distinction of history transitions into a *deep* and a *shallow* variant, see Section 2.3. While the shallow history only affects the direct elements of the target state and resets nested statechart, the deep variant continues the behavior recursively. In pursuit of a lean design, modal reactors only implement a deep variant.

3.4.3 Modes as Mutations

In the current concept for modal reactors, modes are established as a core language feature and implemented directly in the LF runtime. However, early on in the design process, there was the idea to define and implement modes via mutations [LÍG+19].

Mutations offer an interface to restructure a reactor at runtime. Using this concept as a foundation for modes would mean that modes would be translated into mutations for each modal reactor. The mutations would make sure that at runtime a modal reactor only contains the elements of the currently active mode. Upon transition, the mutations would destroy all these elements and create those defined the target mode.

However, this approach assumes that the lifetime of modes is bound to its activity. Such a design would conflict with the support of history transitions, as already discussed in Section 3.3.3. Mutations would need to store and re-apply the state of modes that are entered with history. Furthermore, the interface for mutations is intended for end-users and does not provide the capabilities to manipulate events sufficiently to implement

mode-local time as in Algorithm 3.1. In the absence of history behavior, an implementation with mutations would be feasible, but history transitions were considered more important.

Another consideration that argues against mutations is their intended use case. Mutations are designed to give the user the opportunity to dynamically adjust the elements of a reactor. For example, instantiating a number of reactors for a parallelized processing of data, such as a map-reduce pattern with a variable input size. In contrast to that, modes are a more static feature that select which parts of the program are active at a certain point in time.

3.4.4 Embedded SCCharts

Section 3.1 discusses various related approaches that represent alternatives to a modal reactor implementation in LF. Let us investigate such an alternative design by conceptually embedding SCCharts as a target language in a non-modal LF. This approach would utilize the polyglot nature of LF to include a notation that can naturally express modal models and facilitates the extraction of mode diagrams.

Concept Figure 3.12 illustrates an example for the proposed design and presents the compilation infrastructure necessary to support SCCharts embedded in LF. The file `ao.lf` in the top right corner represents the source LF program. The reactor uses SCCharts as a target language, but also C to indicate the code synthesis target for the SCCharts. The program models a very simple behavior that waits for the presence of the input `A` and passes its value on to the output `O`. The following `Done` state does not permit a repetition of this process.

In order to create executable code, the LF compilation requires an additional intermediate step. In this step, the SCCharts code is extracted from the reaction and put into a separate artifact, `ao.reaction.sctx` on the right. While the code in the reaction body implicitly assumes the presence of the declared interface, as it usual in LF, the SCChart is now completed with the necessary declarations, to yield a valid model. The events are encoded as signals to represent the event's presence and payload.

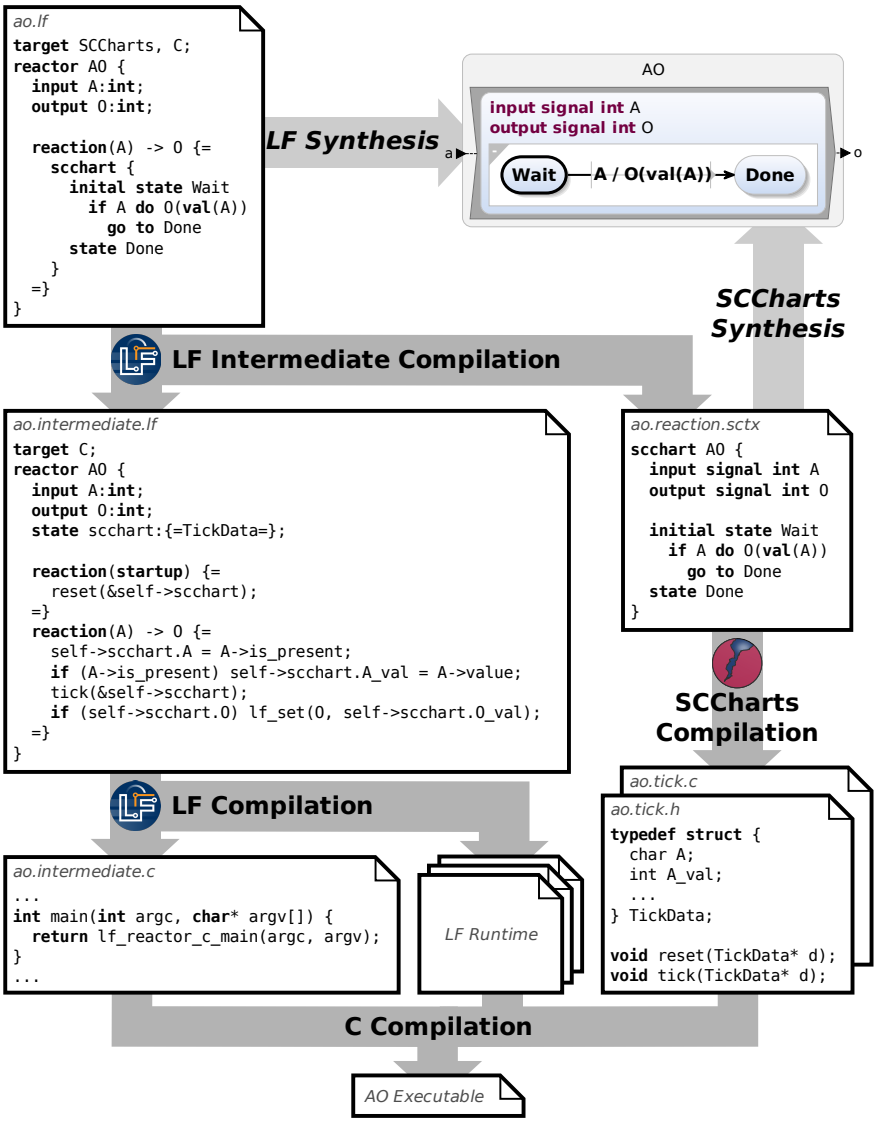


Figure 3.12. Conceptual structure of an LF compilation with an embedded SCChart.

The second output of the intermediate compilation is an adjusted LF source, ao.intermediate.lf on the left. The SCChart in the reaction body is replaced by C code that handles the invocation of the tick functions, as described in Section 2.3.2. Additionally, the program now has a state variable (scchart) to store the internal state of the SCChart and a startup reaction to initialize this data structure.[6]

In the subsequent compilation, the intermediate LF code can be ordinarily processed by the LF compiler, while the SCChart is compiled into its tick infrastructure by the SCCharts compiler. Finally, the C compilation combines all artifacts into an executable.

In addition to the compilation, the LF diagram synthesis can likewise embed the SCChart. By invoking the SCCharts synthesis to create a diagram from the completed SCCharts model, the LF diagram could display the state machine inside the reaction figure, as illustrated in the top right corner.

Evaluation Regarding the goals for a modal model in LF, such a design retains the determinism due to the use of SCCharts and is still compliant to the polyglot black-box approach, as the semantics of LF do not rely on a white-box analysis of the SCCharts code. Yet, there are several disadvantages to such a concept, as already mentioned in the context of similar related work, see Section 3.1.

Most importantly, this design only provides modal reactions, since the SCCharts code does not support embedding reactors, neither practically nor under the black-box abstraction of reactions. Hence, it does not actually constitute a modal notion for the LF coordination layer but only for individual reactions.

Furthermore, this alternative concept reintroduces the problem of non-aligned time. The modal behavior is again confined to a reaction and does not provide for association of timers with modes. However, the concept of dynamic ticks, presented in Chapter 4, improves the timed modeling capabilities of SCCharts in a way that it could handle time locally.

[6] This example uses default names for the tick function and other variables. In the presence of multiple reactions with SCCharts code, this concept requires a naming scheme that prevents conflicts.

Finally, the practical implementation is rather heavy-weight, as it relies on two full compilation stacks for LF and SCCharts. This also includes the diagram synthesis, as it requires an intermediate compilation of the SCCharts model.

Nonetheless, this design also illustrates the powerful integration opportunities that LF offers. For example, Section 6.3.1 will propose a variation of this design that enables SCCharts to utilize LF's distributed execution capabilities.

Part II

SCCharts

Chapter 4

Time

In the comparison between the pendulum implementation in SCCharts and its LF counterpart, there are notable differences in the handling of time, see Section 1.2 and Section 2.4. For SCCharts, this can be condensed into two major opportunities for improvement;

1. the introduction of a more precise and explicit notation for time-related behavior, such as time-outs or periodic executions; and
2. an efficient and precise execution model that embraces the sparseness of inputs and does not trigger reactions for the sole purpose of tracking time.

Modeling Timed Behavior (1.) The abstraction from time in synchronous languages typically comes at the price that all references to physical time must somehow be resolved by the environment. The multiform notion of time [Ber99] is one embodiment of this principle. Following this concept, physical time becomes a second-class citizen, expressed in abstract inputs indicating the passage of time, as the pendulum SCChart illustrates. While this is consistent with the synchronous abstraction, at the end of the day, it makes it harder to for a programmer to express real-time behavior.

Bourke and Sowmya investigated this problem and found that the granularity of time inputs easily imposes imprecision and inconsistencies in time measurements [BS09; Bou09]. Their solution is the introduction of real-time delays. Similarly, LF also has an explicit notation for real-time, to specify timers, define deadlines, and schedule actions. Hence, in order to express precise real-time delays for a program and communicate these to the environment, it is important to have a form of timed modeling capabilities with access to real-time.

Efficient Execution (2.) The SCCharts implementation of the Furuta pendulum counts milliseconds to control its timed behavior, as presented in Section 2.4. Consequently, over the course of a three seconds long execution, (at least) 3000 ticks need to be executed in order to yield the intended behavior w.r.t. to real-time. Moreover, such an execution is only feasible if the computation time for each tick is below 1 millisecond. Otherwise, the behavior would lag behind, as tick execution must not overlap. For the same reason, milliseconds were chosen as a compromise for time granularity in the SCCharts implementation. A higher precision increases the general execution load and may cause delays. Yet, a more coarse-grained resolution will impose delays when timed effects are rounded up to the next millisecond, as it is the case for the sound signal in the SoundController SCChart.

Section 4.5.1 will investigate the tick load of different implementations. It reveals that the LF model requires only 1300 ticks in the same three-second simulation[1], while producing a more precise timing for the sound signal. This is a result of the sparse event-driven execution that only triggers a tick if an input event occurs or an internal timing event fires, such as timers or actions. Hence, a comprehensive timed modeling concept should include a dynamic (sparse) tick execution to decrease the system load. In turn, this opens up space for (theoretically) scheduling ticks onto more precise points in time.

Goals This chapter investigates how to incorporate physical time into synchronous languages, using SCCharts as an example. While such an endeavor is not new and there are many viable solutions in various other languages, see Section 4.1, the goal is to emphasize efficiency in terms of runtime and implementation, as well as modeling aspects in the context of synchronous statecharts. This particularly includes evaluating and incorporating the latest developments in sparse and event-driven execution models, as in LF or with *dynamic ticks* by von Hanxleden et al. [HBG17]. More specifically, the design is guided by the following principles.

[1] This simulation of the pendulum works at a 5 msec pace, which makes tick loads below 3000 possible.

Determinism The semantics should fit seamlessly into the synchronous paradigm and provide deterministic behavior, e. g., outputs are fully determined by inputs. For SCCharts that means there should be no changes to the underlying SC MoC.

Resilience A solution must cope with run-time variations and imperfections of physical timers. It should be possible to avoid accumulations of timer imperfections and to detect variations and lags.

Scalability The number of (concurrent) timers should not be restricted or impose significant overhead per timer.

Fine granularity The specification of time constrains should not be restricted by a specific granularity. For example, it should be possible to specify timeouts of 1 sec and 3.1415926 msec in the same model.

Time composability Time-based constraints should remain their intuitive semantics if composed. E. g., waiting 1 sec twice should mean the same as waiting 2 sec's once.

Simultaneity and order Timers that started in the same tick and run the same duration should expire in same tick.

Lean interface The inference between the model and its environment should be simple, lean, and independent of application specifics or the number of timers.

Seamless compiler integration In the context of SCCharts, any solution should fit into the incremental compilation concept [MSH14; Smy21]. It should consist of a minimal core, with more advanced modeling aspects implemented as extended features, see Section 2.3.

The concept of *timed SCCharts* presented in this chapter embodies these very principles and implements dynamic ticks to create a precise and efficient execution model with a light-weight interface and implementation.

Outline This chapter starts with a brief overview of related work on time in modeling languages and synchronous languages in particular. Next, Section 4.2 will introduce the timed automaton notation in SCCharts and

other modeling capabilities of timed SCCharts. Section 4.3 discusses different execution regimes for a timed model, ultimately settling for dynamic ticks. Then, Section 4.4 illustrates how dynamic ticks are implemented in SCCharts and discusses strategies to deal with imperfections when exposed to physical time. Finally, Section 4.5 evaluates the performance of timed SCCharts and compares the design to event-driven approaches, such as LF.

4.1 Related Work

Time plays an important role if a program has to work in relation to the real world, most prominently in the form of real-time systems. The same holds for distributed execution [LL21]. Many specification models and languages have been developed over the years that provide notions of time. Here, synchronous languages are again of particular interest to this thesis.

4.1.1 Modeling with Time

Specifying a model with explicitly timed behavior requires a capable modeling formalism and a notion of time.

Timed Automata A formalism for timed models that is particularly relevant in the context of state machines and SCCharts are *timed automata* by Alur and Dill [AD94]. Timed automata consist of state-transition graphs with additional real-valued *clocks* that enable expressing timing constraints for transitions. They extend the theory of ω-regular languages into *timed words* that pair the input word with an (infinite) sequence of real-valued time values indicating their occurrence. This results in a dense-time model. Clocks are similar to regular program variables but bound to the continuous flow of time. Having multiple clocks with the option to independently reset their values in addition to their natural progression provides a powerful and flexible modeling concept for the specification of temporal constraints.

Timed automata in different forms and variations have been extensively studied for verification purposes [AD94; ACH+95; OSY94; HNS+94]. In the context of this thesis, they will be used with a focus on code synthesis and

the synchronous MoC. This includes the investigation of practical execution strategies, see Section 4.3.

Altisen and Tripakis [AT05] investigate the effects of execution semantics and platforms on the timed automaton behavior. They propose an implementation methodology that wraps a timed automaton into a *global execution model*, to decouple the real-time access and interpret the original model based on simulated time. This approach makes it possible to keep a fixed behavior for timed automata that is independent of influence of the execution environment on timers, which facilitates platform-independent verification.

This thesis will not focus on the topic of verification, but the interface of timed SCCharts to their environment also facilitates a form of wrapped simulated time. However, as Section 4.4 will discuss, it also supports the opposite approach by providing access to the raw real time. Exposing the model to platform-specific imperfections may yield a different behavior but also enables the model to detect them and adjust its control behavior accordingly.

Multirate Timed Automata Timed automata have been extended in various ways, one example are multirate timed automata (or multirate timed systems) [ACH+95]. There, each clock progresses at its own speed, possibly varying between a lower and an upper bound. This further extends the capabilities of clocks to model timed behavior, e. g., in the context of Cyber Physical System (CPS) [LS17]. Olivero et al. illustrate that multirate timed automata can be mapped to single-rate timed automata [OSY94].

The implementation of timed automata in SCCharts will feature single-rate clocks, but their design as an extended feature also facilitates a more advanced behavior in the future.

Discrete Timed Automata Pinisetty et al. [PRS+17] introduce discrete timed automata to formalize runtime enforcement of CPS, also using SCCharts in the process. Instead of real-valued clocks they use discrete clocks, counting specific timed events, in this case periodic ticks.

While timed SCCharts may also use integer-typed clock implementations to circumvent issues of floating-point arithmetic, discussed later in

Section 4.2.2, they rely on a dense-time model. Furthermore, dynamic ticks are specifically designed to overcome a fixed discrete time progression, as in a periodic execution, see Section 4.3.

Uppaal The Uppaal tool provides an environment for modelling and verifying real-time systems using timed automata [LPY97]. These systems are specified as a network of non-deterministic sequential processes featuring concurrency, advanced data types, and communication channels with synchronization and prioritization capabilities. Uppaal supports model checking and simulation by using symbolic interpretation and statistical model checking techniques [BBB+10].

For timed automata in SCCharts the focus lies more on the aspects of practical real-time modeling in a synchronous context, rather than verification. Nevertheless, with model checking capabilities in SCCharts [Sta19], there is also potential future work in this direction.

CCSL The Clock Constraint Specification Language (CCSL) [AM09] is a notation for expressing clock domains and relations, independent of a specific programming language. Clocks in the sense of CCSL correspond more to the concept of polychronous systems [GTL03], rather than clocks in timed automata. Yet, CCSL not only provides patterns of classical synchronous clock constraints for multiclocking [GG10] or asynchronous clock relations [Lam78], but also real-time representations of physical time. For example, CCSL is used in a simulation and debugging tool for LF programs [DCB+21].

For timed SCCharts, CCSL opens up a future avenue into multiclocked and polychronous systems design, see Section 6.3.2.

Ptolemy The heterogeneous modeling environment of Ptolemy II provides different domains to express timed behavior, such as discrete events or continuous time [EJL+03; Pto14]. It also includes a multirate timed automata implementation, illustrated later in Figure 4.1. Internally, time is a global property of the simulation and provided as a single unit to all components. A floating-point number specifics its resolution, but to ease and harden time arithmetic, time itself is handled in the form of integer multiples of that resolution [CLB+19].

In timed SCCharts, time also conceptually originates from a real-valued domain, but the implementation is adjustable on an application-specific basis, see Section 4.2.2.

ROOM The real-time modeling capabilities of the actor-oriented ROOM language [SGW94] include time-controlled behavior. In contrast to a model-based specification, such as timed automata, ROOM supports the periodic creation of events [SFR97], similar to timers in LF. Additionally, the ROOM virtual machine provides access to a low-level timeout service. Timing constraints in ROOM mainly concern classical real-time scheduling and response times analyses, as they specify bounds on arrival times of time-triggered events and deadlines on event-processing sequences [SFR97].

While timed SCCharts can be used for hard real-time tasks, see Section 4.5.2, this proposal does not investigate the combination with real-time scheduling strategies or analyses.

Statecharts Harel's proposal for statecharts also includes expressing timed reactions [Har87]. Using an implicit notion of timers, a transition can specify a timeout based of the occurrence of an event or the entry of the state. Timers refer to a global notion of discrete time steps that is controlled by the way the model is simulated [HN96].

The effects of different simulation strategies on timeouts are similar to the consideration of logical and physical time in Section 4.4. However, timeouts in statecharts are an abstracted notion of discretized time and do not provide access to real time, limiting the model in reacting to its actual environment. Furthermore, timed SCCharts account for both discrete and real-valued models of time.

4.1.2 Synchronous Languages

Classically, synchronous languages rely on a more abstract notion of time. Yet, there are several synchronous languages that handle time in a more robust and precise fashion compared to the multiform notion.

Esterel In its classical form, Esterel relies on the multiform notion of time [Ber99]. Bourke and Sowmya investigate the implications and draw-

backs of this design, such as imprecisions when measuring time intervals via fixed-paced time inputs and inconsistencies between multiple time inputs in different granularity [BS09]. They propose an extension by real-time delays. These timed delays act as a form of abstract macros that are subsequently implemented using sampling or event-based strategies, similar to the considerations in Section 4.3. However, there is a gap between the expressed timing constraints and the actually provided behavior by the (platform-specific) implementation. This was one motivation for the development of dynamic ticks by von Hanxleden et al. [HBG17].

The proposed concept for timed SCCharts tries to overcome this gap by using dynamic ticks and the option to handle raw physical time inputs in the model.

Zélus Zélus [BP13] mixes both discrete time and continuous time behavior specification in a single synchronous language. It uses a discrete Lustre-like dataflow syntax with automata and combines it with ordinary differential equations for continuous behavior. At runtime, an external solver simulates the continuous time domain and detects zero-crossings, at which the discrete sections of the program can react. The type system and causality analysis ensure that the hybrid segments correctly align and no discontinuities occur during integration of the two domains.

The proposal for time in SCCharts does not aim to express continuous behavior but to provide a timed extension that enables explicit modeling of time-related behavior based on a combination of real-valued and logical constraints.

Argos The statecharts-like synchronous modeling language Argos [Mar92] is a predecessor of SyncCharts [And03] and thus in spirit also of SCCharts. An extension by Jourdan et al. introduces the specification of *temporized states* for verification purposes [JMO93]. States can carry timeouts, similar to statecharts, and must be left before it expires.

Timeouts in Argos are abstract and can either be implemented by dedicated discrete events that are passed to the program but cannot occur simultaneously with other inputs, or by internal timed automata. However, Argos does not provide an accessible notion of time to handle differences in the implementation, as timed SCCharts strive for.

Céu The imperative synchronous language Céu [SIL+17] is strongly inspired by Esterel. However, there are some key differences, such as the notion of time, which is a first-class citizen in Céu. Timers can be used to express real-time delays and expose physical lag to the program to enable adjusting the behavior. Timing events dynamically trigger reactions and the Céu runtime tries to automatically compensate for physical lag.

There are many similarities between the handling of time in Céu and timed SCCharts. However, a core difference is the fact that Céu uses an event-based runtime and thus is more closely related to LF. This includes a notion of logical time for events. In contrast, the concept of timed SCCharts embraces the classical tick function-oriented approach and is not bound to an event queue or event-based processing.

The Sparse Synchronous Model Inspired by the concept of PTIDES [ZLL07; ELM+12], Edwards and Hui developed the Sparse Synchronous Model (SSM) [EH20; HE22]. It combines an event-driven, and hence sparse, execution regime with the principles of synchronous languages. Consequently, it is closely related to Céu and also LF. While externally- and time-triggered reactions are driven by events, the internal program behavior is synchronous and based on activation of routines, similar to reaction in reactors but permitting recursion. In terms of timing specification, the SSM provides a statement that postpones writes to variables based on a real-time delay, while another one can wait for the occurrence a write access.

With dynamic ticks, timed SCCharts also achieve a more sparse execution regime but also provide a more flexible interface for real time behavior.

4.2 Timed Automata in SCCharts

Timed automata are a well-established and extensively studied formalism for the behavior of real-time systems, see Section 4.1.1. The research question investigated in this section is: How can we seamlessly integrate timed automata into SCCharts, while maintaining the underlying MoC and modeling principles and capabilities? In particular, this includes practical considerations, such as a robust execution semantics, efficient and scalable

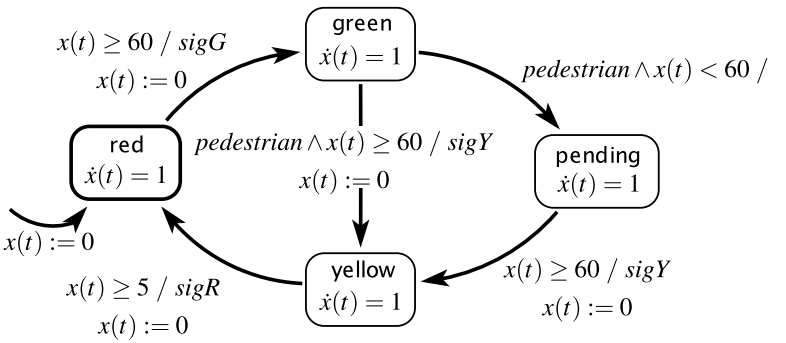


Figure 4.1. Trafficlight controller modeled as timed automaton by Lee and Seshia [LS17] (CC BY-NC-ND 4.0).

code generation, and handling varying granularity in the time representation, imposed by the application environment or hardware. Additionally, this section will investigate the utilization of timed automata into a higher level construct for multiclocked synchronous programs.

The Traffic Light Controller Example The Furuta pendulum example illustrates the need for improved timed modeling capabilities in SCCharts. However, the parts of the model that can be expressed in timed automata are relatively small and are best matched with more advanced concepts, as Section 4.2.3 will illustrate. Hence, to introduce, discuss, and investigate the basic notion and semantics of timed automata in SCCharts, the *traffic light controller* is better suited example. It is a model used by Lee and Seshia [LS17] to discuss (multirate) timed automata in the context of CPS design. Figure 4.1 shows this traffic light controller.

The represented traffic light has three lights, green, yellow, and red, to control the car traffic. An additional button for pedestrians causes the traffic

light to temporarily switch to a red light to stop the traffic and provide for a safe crossing of the street.

The controller handles the timed behavior of a single traffic light. It has a real-valued clock x, a pedestrian input that indicates a pedestrian request for crossing, and three outputs sigR, sigG, sigY. The type pure denotes a signal without payload that is either present or absent at each reaction. For the pedestrian input this denotes the event of pressing the button. Likewise, the outputs use the signal to trigger color changes rather than controlling the state of the light directly. It is assumed that the red light is turned on initially and subsequently emissions of signals will switch from the current configuration to the requested one.

In this notation, the clock is represented by a first-order differential equation on a real number. It can be explicitly set (e. g., $x(t) := 0$) or used as a transition constraint. Time progresses in states controlled by an explicit derivative, in this example one ($\dot{x}(t) = 1$). Furthermore, the resolution of time is expressed in abstract units. For this example, we can assume that one time unit corresponds to one second.

The automaton for the controller consists of four states red, green, yellow, and pending, with red as the initial one. Transitions between states carry optional guards and effects, using the same notation as in SCCharts. One could denote guards as triggers, analogously to SCCharts, but this would already imply a specific execution semantics, as discussed in Section 4.2.1.

The system starts in the red state. When the clock x reaches or surpasses the threshold of 60, the transitions to green is enabled, which will emit the signal for switching to a green light and resetting the time to zero. In the green state the system waits for a pedestrian to push the button, but the following state depends on the passed timed. Case 1, if less than 60 sec passed since entering green, the automaton will transition to pending, but x is not reset. It remains there until the time has reached at least 60 sec, then the yellow light is turned on, the timer is reset, and the state is switched to yellow. Case 2, if the pedestrian signal is received after at least 60 sec have passed, the automaton transitions directly to yellow with the same output and reset. After at least 5 sec in state yellow, the automaton switches to red, signals a red light, and resets the time in x.

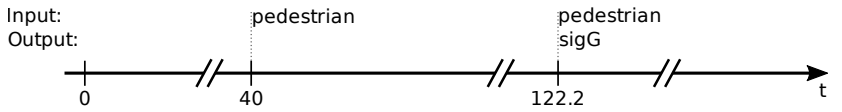
4.2.1 The Eager Semantics

At a first glance, the specification of the traffic light controller seems rather clear and straight-forward. However, when considering specific scenarios, there is some variation in the way the controller may behave.

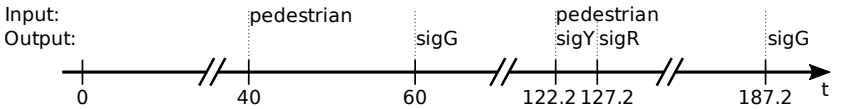
Formal Model The original definition of timed automata [AD94] is based on a timed regular language. It associates each input symbol in a word with a real-valued time stamp. Formally, a *timed word* is a pair (σ, τ), where $\sigma = \sigma_1, \sigma_2, \ldots$ is an infinite word over the input alphabet Σ and a *timed sequence* $\tau = \tau_1, \tau_2, \ldots$, an infinite sequence of time values $\tau_i \in \mathbb{R}$, satisfying monotonicity and progress constraints. Given a timed word, a *run* of a timed automaton is an (infinite) sequence of state transitions, analogous to standard regular languages defined by classic automata. In timed automata, a transition only reads the input if its clock constraints are satisfied. Furthermore, runs have to provide an *initialization*, starting all clocks at zero in the initial state, and *consecution*, relying only on transitions that satisfy the input and clock constraints, while clocks only progress in adherence to the input time and internal resets.

The fact that the automaton is driven by inputs and only implicitly by time (in association with inputs) can be the source of unintended behavior. To concretize this, assume that in the traffic light controller the pedestrian button is triggered at times 40 and 122.2. This constitutes the following input sequence (timed word): $(\langle\{\mathsf{pedestrian}\}, 40\rangle, \langle\{\mathsf{pedestrian}\}, 122.2\rangle)$. For convenience, this notation extends the concept of timed words such that the inputs σ_i do not have to consist of exactly one event, but constitute an arbitrary *input set* that is associated directly with a time stamp. It also permits finite input sequences.

Purely Input-Driven Semantics From a practical perspective, the timed automaton now needs to perform reactions in order to process a given an input sequence, one for each time-stamped input in the sequence. Figure 4.2a illustrates an execution trace of the traffic light controller consuming the previous input sequence. The trace starts at time 0 with the implicit ini-



(a) Purely input-driven execution



(b) Eager semantics execution

Figure 4.2. Illustration of variation in the timed behavior of timed automata based on the execution strategy. The execution traces show the reactions (vertical strokes) of the traffic light controller under different regimes. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

tialization.[2] At time 40 nothing happens because the timing constraint for the transition to green is not met. This assumes that the timed automaton shown in Figure 4.1 has implicit default transitions, enabling the model to remain in the current state if no outgoing transition is available. Otherwise, there would be no admissible run for this input sequence because there is no transition that can accept any input at time 40. Finally, at time 122.2, the automaton transitions to green and emits sigG. Then there is no further reaction due to the absence of further input events.

While this run constitutes a valid execution of the timed automaton, it most certainly does not correspond to the intended behavior of the creator of the model. For example, the output sigG should probably not occur at time 122.2, even though $122.2 \geqslant 60$ certainly holds, but rather at time 60. To reflect such behavior, time (without further input events) also needs to be a trigger for reactions, in particular if the automaton contains transitions that are guarded solely by timing constraints.

Eager Semantics In order to issue additional reactions, the input sequence must be extended by events in the form $\langle \varnothing, \tau_x \rangle$ that originate from the model itself to trigger reactions to time. However, this raises the question at

[2] One could also make this initialization reaction explicit by including τ_0 (defined as zero) in the input sequence: $(\langle \varnothing, 0 \rangle, \langle \{\text{pedestrian}\}, 40 \rangle, \langle \{\text{pedestrian}\}, 122.2 \rangle)$.

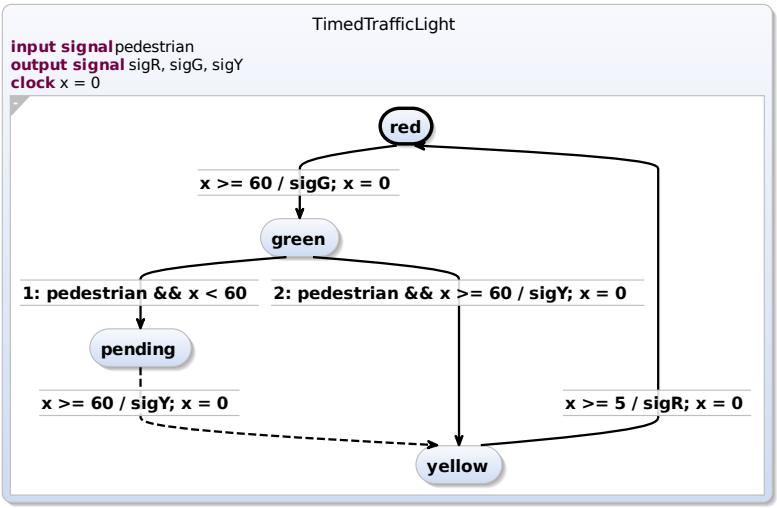


Figure 4.3. The traffic light controller modeled in SCCharts using the timed automaton notation. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

which time (τ_x) a reaction should occur. For example the transition from red to green could occur at any time from 60 onward. Lee and Seshia [LS17] resolve this by assuming that a transition is taken as soon as it is enabled. Thus, the automaton conceptually reacts "continuously."

This assumption can be denoted as *eager* semantics and leads to the trace in Figure 4.2b. It augments the trace in Figure 4.2a by further input events at times 60 (emission of sigG, transition to green), 127.2 (emission of sigR), and 187.2 (sigG again).

This eager semantics constitutes the targeted behavior for timed automata in SCCharts, and Section 4.3 will discuss different approaches to achieve this semantics in practice.

4.2.2 Timed SCCharts

As it turns out, the synchronous MoC fits quite naturally into the execution semantics of timed automata. Likewise, the state machine notation makes

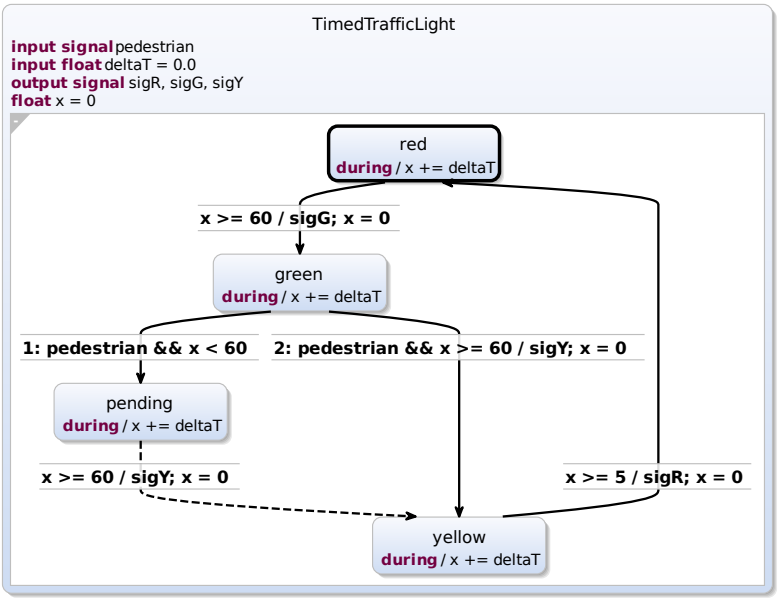


Figure 4.4. The traffic light controller SCChart after the clock transformation. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

SCCharts well-suited for expressing timed automata. With *timed SCCharts*, the modeling capabilities of SCCharts are extended by explicitly timed behavior in the form of timed automata. The following concept is based on introducing a new extended feature: clock declarations.

Figure 4.3 shows the SCChart implementation of the traffic light controller. Despite some minor syntactical differences, the structure of the state machine itself and its transitions and their effects are the same as in Figure 4.1. The new SCCharts keyword clock declares the clock x, which will automatically advance with the external progression of time. As in timed automata, it can be used to trigger/guard transitions and can be set to arbitrary values.

Simple Transformation In line with SCCharts' concept of extended features, clock declarations are replaced during compilation. Figure 4.4 presents

the compiled intermediate result of the TimedTrafficLight SCChart, revealing its actual internal implementation and behavior. The clock x is now an ordinary floating point variable. Additionally, the SCChart received a new input deltaT that is used for the progression of clocks. The only obligation on the run-time environment is that at each tick deltaT is set to the time passed since the last tick. This corresponds to the timed sequence τ that consists of the elapsed time between events.

The progression of time for clock x is represented by during actions in each state. They increase x by deltaT. The during actions are non-immediate to ensure that only the time passed inside a state is considered and clocks are not advanced multiple times. The advancement of x could be multiplied by the slope of the clock to support a multirate timed automata design. However, the implementation does not yet provide a syntax for this feature. Yet, a user could always perform the described transformation manually and adjust the result. In this example, the scaling of the clock is irrelevant, as it is 1 in all states.

Sequential Constructiveness When considering the individual execution steps of the different ticks in the traffic light controller, one can notice that in this implementation x may instantaneously assume up to three different values within a single tick. The value at the beginning of a tick, the incremented value computed by the during action, and the reset value when a transition is taken that resets x to zero. While such destructive updates are a problem for most synchronous languages[3], the SC semantics handle them with ease, as they follow a natural sequential ordering.

Concurrent Clock Access The transformation illustrated by Figure 4.4 is inspired by classical timed automata and advances clocks in each state. In part, this is also a precaution for supporting multirate timed automata. However, in contrast to a simple automaton, SCCharts modeling involves using hierarchy and concurrency, which can create situations in which this approach is insufficient.

[3]Implementing the same concept in a non-SC synchronous language would be a bit more involved. Still, with for example an SSA-like renaming multiple values per tick could be supported in classical synchronous languages [SSH+18b; SSH+18a].

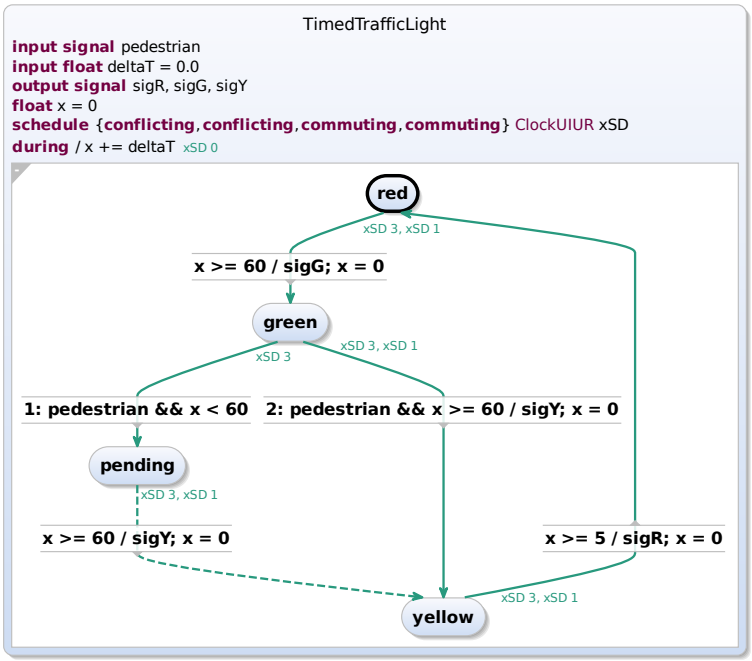


Figure 4.5. The traffic light controller SCChart after the advanced clock transformation using Scheduling Directives.

In the presence of concurrency, multiple states can be active at the same time, but we do not want each of them advancing the same clock. Hence, if multiple concurrent regions access the same clock, the advancement of clocks is no longer handled in each state but in a separate region (during action) concurrent to all these regions. For multirate timed automata this requires a consensus of all regions for the applied slope.

Figure 4.5 shows the result of this advanced transformation approach, ignoring the lack of necessity due the absence of concurrency in the traffic light controller. The main difference to the previous result is that all during actions that handle the progress of the clock x are consolidated into a single during action on top level. However, there is a major problem with this approach, as it can no longer be scheduled under the standard SC semantics.

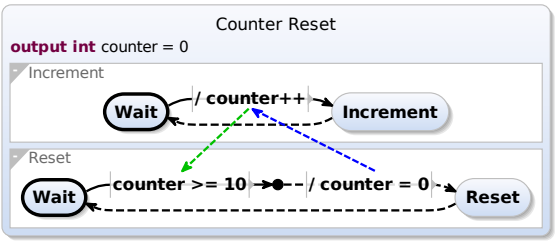


Figure 4.6. A concurrent counter reset, illustrating the fundamental problem with clocks managed by a single concurrent during action [SSH19]. (©2019 IEEE)

The issue is caused by the now concurrent during action and its relations to the resets and readers. Previously, the sequential relation of inner state behavior and outgoing weak abort transitions constituted their ordering, but now they are subject to the concurrent communication protocol. However, the IURP orders updates (here, the clock progression) after the initializations (reset) and both before reads, while the transitions enforce a sequential order from reading the clock to resetting it. This constitutes a causality issue. Figure 4.6 illustrates the underlying problem. It shows a counter (corresponding the clock) that is updated while concurrently read and then reset. Additional arrows indicate the data dependencies imposed by the IURP (blue for initialization → update and green for update → read).

One way to work around this issue could be the encoding of the clock update as an absolute write while users have to express resets as updates (e. g., `x -= x`). However, the SC semantics are not conceptually bound to the IURP, as it is only one possible synchronization protocol implemented for SCCharts, and should not be limited by it. For this reason Smyth et al. created *Scheduling Directives (SDs)* [SSH19]. The issue with clock resets was one motivation to extend the SCCharts compiler to support customized scheduling protocols that override the default IURP.

The SCChart in Figure 4.5 contains the SD `xSD` that introduces a scheduling regime for the clock `x`. The `ClockUIUR` regime is an extended variant of the standard three-staged IURP, but with an additional phase at the beginning for advancing the clock. The SCChart shows that only the during action is assigned to this stage (`xSD 0`), while the transitions indicate that some of

their triggers and effects are subject to phase one (reset) and three (read). The current visualization does not yet support a per statement display of scheduling assignments, as present in the source code. Classical updates (relative writes on x by the user) that would fall in stage two are not present. The declaration of xSD lists stages as conflicting and commuting, which has no further effect in this specific example and will be discussed in more detail in Section 5.4.2.

The use of the custom SD xSD renders the SCChart in Figure 4.5 permissible for the SCCharts compiler, while providing the same behavior as the model in Figure 4.4 because the clock progression is scheduled before any other access.

Similarly, the same issue arises with hierarchy, even in the absence of concurrency. At a first glance, one could think that adding during actions only to inner states but not the superstates, following the previous strategy, would be sufficient. However, regions may terminate, leaving no active state other than the superstate, and this state may not be left at that time (e. g., in the absence of a termination transition). This results in clocks not being updated. Yet, updating clocks only in the top-most superstate, again, creates a concurrent context between this during action and inner resets, and that requires an SD.

Compositional Effects on Time Progression By default the SCCharts compiler tries to use the simple transformation, since it best corresponds to the classical flat timed automata schema. If concurrent or hierarchical use of a clock is detected, an error is raised suggesting the activation of the advanced transformation. This is a manual process because it changes the way clocks interact with other SCCharts features.

The simple transformation assumes that there is only one active state using a specific clock. This state controls the progression of time in that clock, e. g., illustrated by the state-local rates in Figure 4.1. Hence, any effects on the active state of the timed automaton also affect the clock. For example, if the state is suspended, so is the progression of time; and preemption of the state likewise preempts the clock.

In the transformation with support for concurrent access, the behavior is different. Here, the clock is bound to the superstate it is declared in. Hence,

if any contained region or inner state is suspended, it does not affect the progression of time. This is important, since states in other regions may not be suspended and rely on the clock's progression. Only effects concerning the state or region declaring the clock will influence its behavior, since this is the location of the synthesized during action that handles the progression of time.

Time Resolution The transformations illustrated in Figure 4.5 and Figure 4.4 both convert the clock declaration into a float[4] variable. While this is in line with the formal definition of timed automata and enables normalizing time to the common (SI) base unit seconds, it subjects the practical implementation in SCCharts to the known limitations and imprecision of floating point representation and arithmetic in computers, such as quantization errors [CLB+19]. For demonstration purposes or in models without a known hardware context, such as the traffic light controller example, such an approach is acceptable. However, as soon as the model is supposed to run on hardware and meet its specified timing behavior, a more robust integral-based resolution is usually desired. This for example to preserve the associative law for additions. Yet, this requires a more hardware-platform-specific implementation, which is difficult to provide generically.

For example, if targeting a nanosecond resolution in C, the standard time library can provide this precision on Linux, but Windows only yields milliseconds or requires a different API. Embedded processors also vary in supported time access and resolution. Furthermore, effectively handling time in nanoseconds requires integer types with more than 32-bit, which is supported on common platforms but cannot be guaranteed when working with embedded processors.

LF solves this issue by providing a reasonably platform-independent time API for each target language. In C, these are macros[5], such as SEC in line 17 of Listing 2.4. Time specifications on the LF coordination level, for example 15 msec in line 14 of Listing 2.6, internally use the same macros. A preprocessor directive can switch these macros between micro and nanosecond precision.

[4] In SCCharts float is an abstract floating point number and does not express a specific precision limitation.

[5] https://github.com/lf-lang/reactor-c/blob/4c97e960d9a40d60dfb3725678dbd38176023f45/include/core/tag.h

While this approach is appropriate for LF, since it controls the compilation of the execution environment for LF programs, SCCharts follow a more abstract approach in accordance with the concept of generating tick functions, see Section 2.3.2. The deltaT interface likewise aligns with this concept and puts the extraction of time into the hand of the external environment.

For timed automata in SCCharts, the float type acts as the default type. However, the transformation can be configured to synthesize any type. For example by annotating the SCChart with @IntegerClockType "int64_t", the code generator produces host code type for 64-bit integers in C, see Section 2.3.3. Yet, the timing guards may require manual adjustment to the targeted resolution. This can be done by appropriate constants, as in line 6 of Listing 2.5, or using methods introduced alongside OO in Section 5.3.1. The timed automaton implementation of the Furuta pendulum in SCCharts presented in the next section will illustrate this approach by using an SCCharts-based class to provide time conversion utilities. This way, one can achieve a reasonably platform-independent design that is easily adaptable to changes in hardware. Alternatively, an SCChart can be specified with implicit knowledge about the external resolution of time, as in Figure 4.23 in the context of the DS demonstrator.

Furthermore, timed SCCharts provide the special type time for use in combination with clocks. During compilation, time declarations will automatically adjust their type to the one used in clocks and deltaT, i. e., they comply with the @IntegerClockType annotation. This enables the user to store time independent of the actual resolution and host type, and keeps hardware-specific modeling at a minimum.

4.2.3 The Furuta Pendulum in Timed SCCharts

In the Furuta pendulum example, both the PendulumSound and PendulumController use timed behavior and can be re-modelled using timed SCCharts. The actual changes only involve replacing the milliseconds counters by clocks.

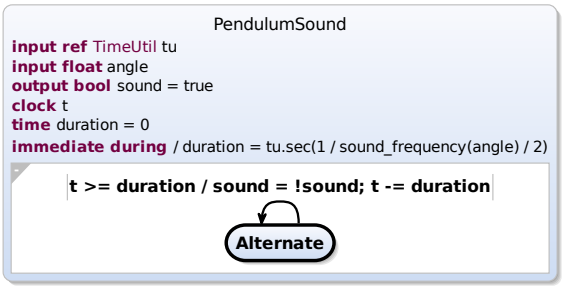


Figure 4.7. The PendulumSound component modelled as timed SCChart.

Pendulum Sound Figure 4.7 illustrates the PendulumSound SCChart as a timed automaton. In contrast to the original implementation, see Section 2.4.2, this component no longer requires the msec input, the msecs counter, or the during action in state Alternate. Instead, it relies on the new clock t to track time automatically. Considering that the transformation for timed automata will reintroduce the same components to track time in clocks shows the nature of extended features.

Compared to the traffic light example, this model does not use a fixed value as timed guard but another variable to model the angle-dependent shifting of the threshold for the sound signal. This extends the classical timed automaton model [AD94], which only permits constant thresholds. While fixed value constraints ease verification, this example shows that variable thresholds are also relevant in practice. The transformation of timed automata in SCCharts naturally supports this feature.

Another difference to the previous implementation is the use of the time type for the duration variable. This ensures the time resolution and type matches the clock t. In the same spirit, the sec method is used in the during action to convert the half cycle length into the appropriate resolution. The method is provided by the tu object of type TimeUtil and is passed as an input to the SCChart. Section 5.3 will provide a detailed introduction of these OO features. In this example, TimeUtil represents an interface for time conversion, inspired by the LF macros. The concrete object that is passed as input will implement these methods for a specific time resolution, e. g.,

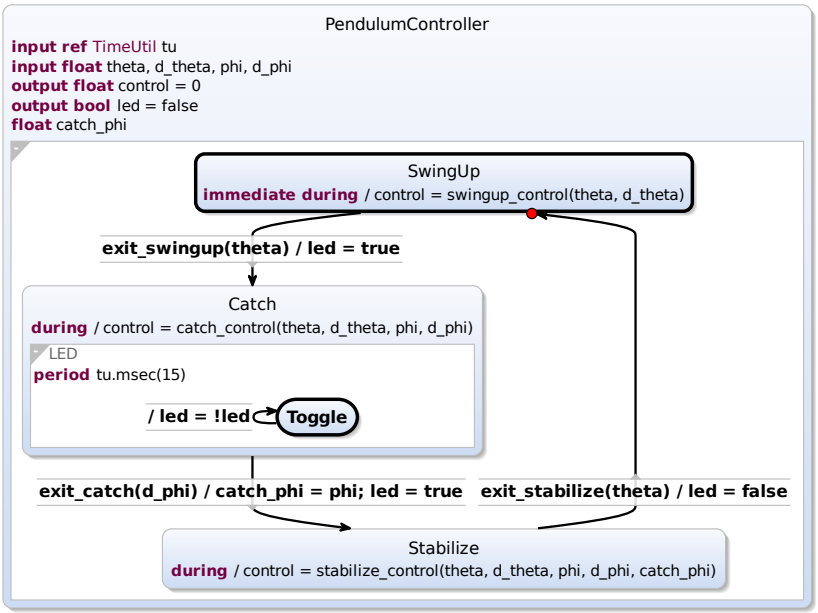


Figure 4.8. The PendulumController component modelled as timed SCChart.

nanoseconds. In the future, the interface and common implementations may be provided as a standard library by the SCCharts tool. This approach illustrates how the model can be kept independent of the underlying time resolution and easily adapted to new platforms. Alternatively, one could use hard-coded factors in the model, such as SEC_TO_MSEC in Listing 2.3 or invoke host code functions to achieve a similar design.

A last difference that should be noted in comparison to the traffic light example is the way the clock t is reset. Instead of setting the clock value to zero, only the duration threshold is subtracted. This design is a precaution for timing imperfections in the actual execution, if t is bound to physical time. Section 4.4.3 will discuss this "soft" reset in detail.

Pendulum Controller In the PendulumController component, the only timing aspect is the periodic blinking of the LED during the Catch mode. While

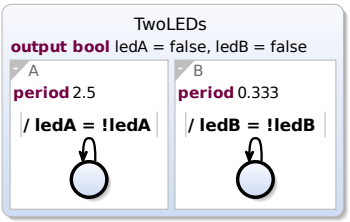
this again can be solved with a timed automaton, the SCChart model in Figure 4.8 illustrates the use of another extended feature in timed SCCharts: *periodic regions*. The period directive confines a region (or state) to a periodic activation. In this example, the state machine in the LED region toggles the boolean value unconditionally in each step, but the period will ensure that these reactions are 15 msecs apart. The next section will illustrate that this feature has a straight-forward translation into timed automata and can be considered syntactic sugar.

4.2.4 Multiclock SCCharts

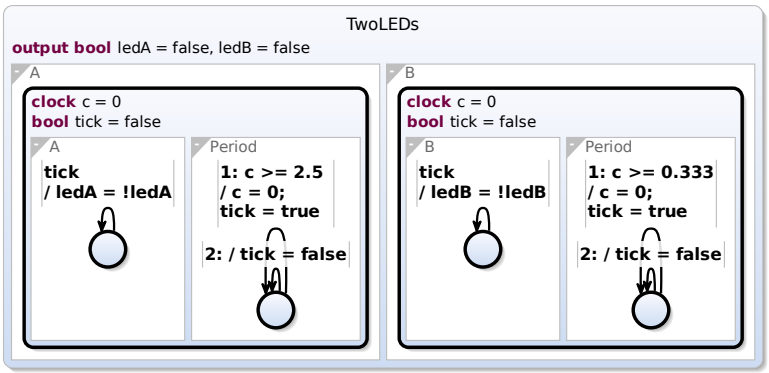
Timed automata naturally support multiple clocks and so does its SCCharts implementation, as there is no restriction of the number of clock declarations. In synchronous languages, there is also the concept of multiclocking [GG10], for example in SIGNAL or Multiclock Esterel by Berry and Sentovich [BS01]. However, in this context the term "clock" does not describe real-valued time measurement but a hardware clock that drives a hardware circuit or similarly designed software. In other words, this notion of clocks refers to the source of discrete synchronous ticks.

In multiclocked systems, different parts of the program are activated by different clocks. These clocks refine a base clock and can be considered additional inputs to trigger specific parts of the program. Polychronous systems [GTL03], as in SIGNAL, go beyond refining clocks and support loosely coupled clocks synchronized by clock relations and hierarchies.

Multiperiodic SCCharts With the period directive, timed SCCharts provide means to express multiclocked SCCharts based on real-time. Periodic regions and states subject their inner behavior to a new clock that refines the base clock of the model. Instead of deriving their pace directly from the discrete ticks of the base clock, periods bind it to a real-time clock used in timed automata. This is especially relevant for modeling with dynamic ticks, as presented in Section 4.4, because dynamic ticks do not provide a periodic base clock to refine but instead derive the pace for ticks dynamically from inner timing requirements.



(a) SCChart with multiple periodic regions



(b) Transformed SCChart

Figure 4.9. Example of a multiclocked SCChart that has two LEDs blinking in different frequencies.

Transformation Figure 4.9a illustrates a multiclocked SCChart using periodic regions. The example is inspired by the LED handling in the Catch mode in Figure 4.8. It simply toggles two LEDs, ledA and ledB on and off. Each in separate regions and with different periods, 2.5 sec for ledA and 0.333 sec for ledB.

Figure 4.9b shows how this extended feature is transformed into a timed automaton during compilation. Both regions A and B are transformed individually but follow the same pattern. The inner states of the region are moved into a new super state that declares a new clock variable c and

a boolean flag tick. The tick variable acts as guard for all reactions in the original state machine. In this example, they are added to the self-transitions toggling the LEDs in each region. If any transition or action has its own guard, it would be conjuncted with tick. This prevents the inner SCChart from performing any action unless enabled by tick. This approach corresponds to the concept of suspension, present in synchronous languages, such as Esterel or SCCharts. In this case, tick is initialized to false, which means that no reaction takes place in the initial tick. However, there is also an *immediate* period directive that initializes tick to true and enables an initial reaction.

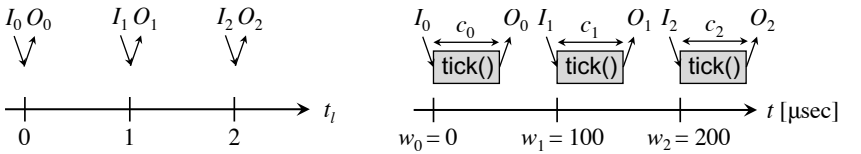
Additionally, there is a new region Period with a single-state timed automaton for each of the periodic regions. In each tick the clock c reaches its period's threshold, the clock is reset and tick is set to true, enabling the reaction in the region that now holds the user-specified behavior. Otherwise, represented by the transition with the lower priority (2:), the variable is set to false.

By introducing the clock directly at the level of the period directive, the guarded state or region is only activated if the given amount of time has passed since the entering/start of the state/region or its last activation. In the PendulumControl in Figure 4.8 this ensures that the blinking behavior is aligned with the activation of the Catch state (cf. Figure 3.1).

Clock Relation Specification While the period directive is a simple way to achieve a form of multiclocking in SCCharts, it also sets a first cornerstone for future work toward polychrony and distributed execution [GG10]. There are some initial efforts to combine multiclocked SCCharts with the CCSL by André [And09] to establish a formal specification for the relation of clocks [SHM+20]. CCSL was also used for the specification and validation of timing requirements in Esterel [AM09] and the simulation and debugging of LF programs [DCB+21]. Section 6.3.2 will discuss this topic in more detail.

4.3 When to React?

Timed automata use timing constraints on transitions and real-valued clocks to express timed behavior. It is clear that if a constraint is not met, the transition must not be taken. However, when the constraint *is* satisfied,



(a) Logical time: time is discretized into logical ticks 0, 1, etc. Input I_i is synchronous with output O_i, the reaction time is abstracted to be 0.

(b) Physical time: the computation of the i-th reaction, corresponding to logical tick i and the i-th call of the tick function, begins at *wake-up time* w_i. Inputs are read at the beginning of the computation, outputs are written at the end of the computation.

Figure 4.10. Different timing abstractions [HBG17]. (©2017 IEEE)

the automaton *can* react. Section 4.2.1 presented the eager semantics that tightens this specification such that the automaton *should* react as soon as possible. While in a theoretical model, it is possible to react at any time and perfectly meet this semantics, in reality, an execution regime can only approximate the eager semantics.

The same holds for the abstraction of time in synchronous languages. Under the *synchrony hypothesis* a reaction does not take time and the program runs in logical time steps, as illustrated in Figure 4.10a. However, in practice a tick takes time to compute (c_i) which create a temporal separation between inputs and outputs, as in Figure 4.10b. Additionally, the tick computations must not overlap because the synchronous program must be able to atomically access its internal state and prepare it for the subsequent execution.

Furthermore, the question of when an automaton should react is not restricted to this "timed" setting. Instead, it is relevant to synchronous programming in general. While LF comes with its own event-driven execution environment, classical synchronous languages, such as SCCharts, synthesize a tick function (Section 2.3.2), which puts the invocation of ticks into the hands of the tick environment. And, as already discussed in Section 4.2.1, this can have implications on the final behavior of the model.

Hence, a more thorough investigation of different execution strategies is necessary to determine drawbacks and limitation. The goal is to identify a

strategy that fits the desired eager semantics, while being resilient, precise, and efficient.

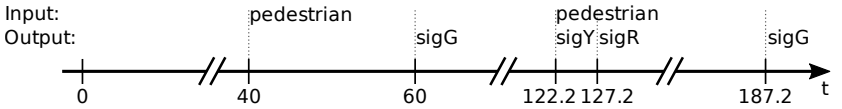
Figure 4.11 shows traces for the traffic light example using different execution strategies for the tick function. Figure 4.11a recalls the trace from Figure 4.2b as it represents the intended execution behavior under the eager semantics.

Input-driven Execution As already discussed in Section 4.2.1 an entirely input-driven execution, as in Figure 4.11b, does not match the intended behavior for timed automata. Without a trigger based on time, transitions with only a timing constraint do not trigger when their condition is met but, in this example, will wait for the next pedestrian input. Consequently, the runtime should provide a strategy that also performs reactions based on time.

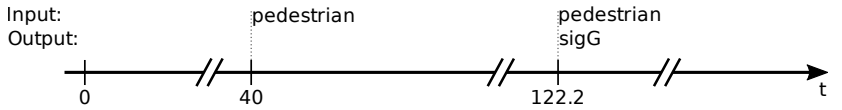
No-delay Execution A common and very simple approach for tick execution is to create a loop without delay [DDR04]. Listing 2.1 illustrates a corresponding implementation. Ticks are triggered as soon as possible (ASAP) after each other, in order to enable the program to react to as many points in time as possible. Figure 4.11b shows the traffic light trace for this strategy. The reaction markers are deliberately irregular, since the time between ticks depends on the individual execution time of each tick.

The reaction times on the timeline also include a question mark in their decimal place. This should indicate that the reaction to an input at that time will (most likely) be subject to an unknown delay. This is caused by the fact that this strategy always has a tick in execution, which makes it unlikely that an input arrives exactly between two ticks and can be immediately processed. Hence, the question mark digit represents this remaining tick execution time until the model can actually try to react to the input.

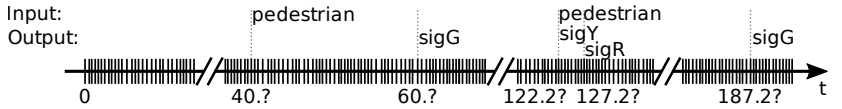
Another disadvantage of this approach is the maximal load that it puts onto the execution platform, while the vast majority of ticks yield no output due to the absence of inputs or insignificant passage of time. Hence, most tick invocations are wasted processor time and energy, which is problematic especially in embedded use cases. This illustrates the need for a more sparse execution that economizes resources by reducing the number of ticks to a relevant minimum.



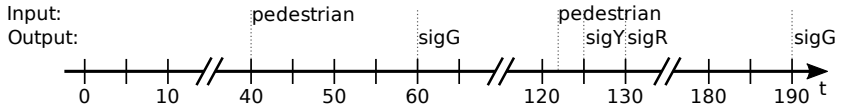
(a) Eager semantics execution as reference (Section 4.2.1)



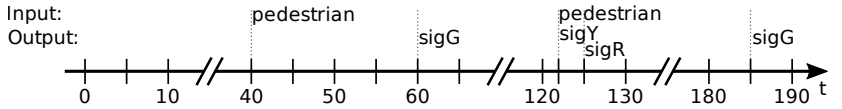
(b) Input-driven execution (same as Section 4.2.1)



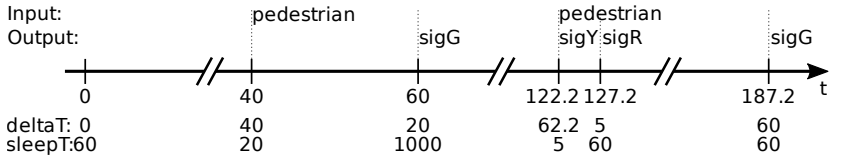
(c) No-delay execution



(d) Periodic execution with rate 5



(e) Time-input-triggered execution, multiform notion of time



(f) Dynamic tick execution

Figure 4.11. Execution traces of the traffic light controller based on different execution strategies. Vertical strokes denote reactions. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

Periodic Execution A straight forward alternative to a no-delay execution is the introduction of a delay that results in a periodic invocation of the tick function. In this strategy, one fixed global period is determined by analyzing the timing constraints of the model, its environment (e. g., poll rate of sensors), and sometimes also its worst case reaction time, to ensure on-time execution of ticks.

Figure 4.11d illustrates a trace with this execution semantics. For the traffic light example, the period is 5. It is the greatest common divisor of the two relevant timing constants 5 and 60 in the model, which constitutes a sufficient sample rate for the system's timing constraints. However, since this is the only source of tick invocations in this strategy, it subjects inputs to the same processing pace. This causes the pedestrian input occurring at time 122.2 sec to be processed in the next period at time 125 sec, consequently the sigR signal is also emitted at time 130 sec. This behavior might be sufficient, especially when there are corresponding sample rates for hardware sensors, such as the pedestrian button. Nonetheless, it does not facilitate on-time reactions, as desired in the eager semantics.

While less pronounced than in the no-delay execution, this approach is still relatively inefficient when it comes to tick invocations. For example, for a delay of 60 sec as in the red state, there are always 12 ticks executed, even though the transition can only be taken in the 12th tick.

The Multiform Notion of Time With the multiform notion of time in classical synchronous languages the execution follows an input-driven approach, but with explicit inputs for the passage of time. This results in a time- and input-triggered execution. The progression of time in the model is then measured by counting occurrences of the dedicated time signals.

In contrast to the periodic execution, it does not require an analysis of the program to determine a global rate but simply serves the inputs defined by the modeler. While this makes the concept quite flexible, it can easily lead to temporal inconsistencies, in particular if multiple input signals are used to model time, as discussed further by Bourke and Sowmya [BS09].

For example, in the traffic light model the relevant timing thresholds are 5 and 60. Now, to model this with multiform time, one could introduce two separate timing inputs, e. g., fivesec and sixtysec. However, waiting for the

next occurrence of sixtysec does not necessarily mean the same as waiting for 12 occurrence of fivesec.

To circumvent this problem, the trace in Figure 4.11e is based on a model that only introduces a single timing input for the greatest common divisor of all timing constraints, in this case fivesec, which can be used to derive the 60 seconds threshold. As the trace illustrates, the system reacts every 5 seconds, always with fivesec present. Additionally, there is a reaction at time 122.2 sec, when pedestrian is present but fivesec is absent.

Consider time 122.2, when the pedestrian input is processed and sigY is emitted. Since time is measured by counting fivesec events, and the last such event has occurred at time 120, the pedestrian event is effectively considered to have taken place at time 120. Consequently, sigR is *already* emitted at time 125 instead of 127.2. This reduces the delay between yellow and red to only 2.8 sec, contrary to the intended 5 sec in the specification. Similarly, sigG is emitted at time 185, which is also earlier than in the trace in Figure 4.11a.

For this specific trace, one could comply with the eager semantics by increasing the granularity of the discrete time input, i. e., using an event for 0.1 sec passed. However, this would in turn increase the number of reactions and load on the system significantly, getting closer to a no-delay-like trace.

A similar problem is present in the PendulumSound component with millisecond inputs, see Section 2.4.2. Here the delays stem from the wave lengths of notes, which are real numbers, and would require a infinitesimal timing input to be captured by accumulation in counters. In this case the approximation by milliseconds reduces the precision of the produced output, similar to the traffic light trace. Section 4.5.1 will investigate this imprecision and compare it with the dynamic approach.

Dynamic Ticks Neither the periodic nor the time-input-triggered execution strategy was able to match the timing of the eager semantics. The problem is that both took a static approach to the granularity and pacing of time events. In periodic execution the program was analyzed for a globally matching pace, and in the multiform notion of time the timing inputs imply the relations to time. However, the relevance of internal timing triggers depends on the program's state, and external stimuli are simply unpredictable. Therefore, a more dynamic approach is needed.

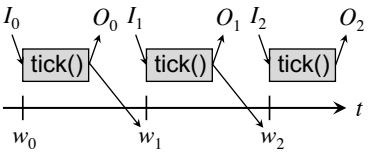


Figure 4.12. The basic idea of dynamic ticks: the wake-up times are controlled by the tick function itself [HBG17]. (©2017 IEEE)

The concept of *dynamic ticks* by von Hanxleden et al. [HBG17] is designed to fill this gap. Instead of a fixed pace for tick functions, it enables the program to compute and communicate its own recurrence. Figure 4.12 illustrates this idea. In addition to the outputs, the tick function yields its desired wake-up time w_i. In order to produce a wake-up time the program also requires access to time. In the concept of dynamic ticks, time is considered a continuous entity and only discretized by the occurrence a tick, equivalent to producing a timestamp for an event. Hence, time is kept in its real-valued domain (or platform specific representation, see Section 4.2.2). This circumvents the previously discussed problems that stem from an over-discretization of time. Furthermore, this approach preserves the determinism of the synchronous system [HBG17].

However, the concept, as discussed so far, only handles the recurrence of ticks originating from timing considerations inside the model but not from external sources. In order to facilitate on-time reactions to inputs outside the determined wake-up time, dynamic ticks must be combined with input-driven triggering. This fits naturally into this dynamic approach because when such an additional tick occurs, the program can simply decide to either reaffirm or update the previous wake-up time.

Figure 4.11f illustrates the trace for the traffic light example under this dynamic tick execution. This strategy matches the eager semantics, while producing the minimum number of ticks necessary to provide the intended behavior. The additional numbers below the tick line indicate the communicated current and wake-up time, using a relative notation, `deltaT` and `sleepT` that will be introduced in Section 4.4. The reported sleep time at time 60 is a special value that could just as well be infinity, since it indicates

that there is no time related triggering from within the model. At that point, the program requires an external stimulus, which then appears in the form of the pedestrian input at time 122.2 and then again issues a wake-up time at 127.2.

It is important to note that the LF semantics produces the same execution behavior as dynamic ticks, since both follow the same dynamic approach. A difference can be found in the form of communication. While in dynamic ticks a single wake-up time is set, LF programs are able to produce multiple future events (via actions and timers). These events will enter the event queue together with external inputs. Then, ticks are executed dynamically as soon as the logical time progression hits the timestamp of the earliest event. Section 4.5.3 will investigate these differences in more detail.

4.4 Dynamic Ticks in SCCharts

As discussed in the previous section, the concept of dynamic ticks is well-suited to efficiently produce the behavior specified by the eager semantics. It embodies a sparse execution mechanism that refrains from introducing additional ticks without intended workload. At the same time, it requires only a minimal communication interface to achieve the dynamic invocation of ticks.

This section investigates the concrete manifestation of dynamics ticks in SCCharts. Technically, this includes the lean integration of the communication interface and support for automated sleep times based on the timed automaton notation. Additionally, this section addresses research questions regarding imperfection of physical time and means to achieve a resilient timing behavior in applications.

4.4.1 A Dynamic Tick Environment

Figure 4.13 illustrates how dynamic ticks integrate into the tick environment of SCCharts. The components in red are new compared to the initial version presented in Section 2.3.2. At the core of dynamic ticks is the minimal interface of communicating current and wake-up time. This design uses

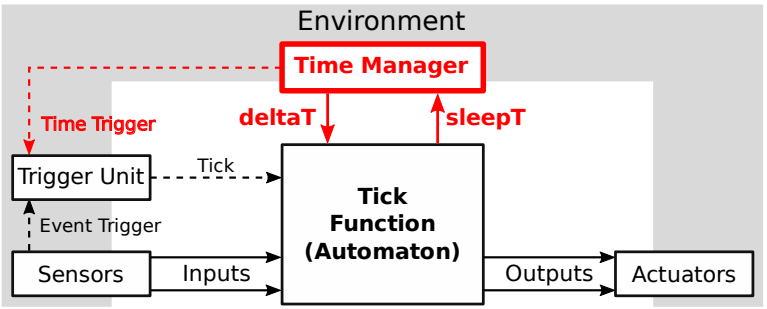


Figure 4.13. A dynamic tick function and its environment. Components in red are new compared to Figure 2.5. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

a relative notation with an input deltaT for passing the time elapsed since the last tick and a sleepT output to communicate the time until the next time-related reaction is expected. The new input and output simply extend the environment of the tick function. A Time Manager is responsible for providing time and triggering a tick when the requested sleep time expired. The triggering by events remains the same, as previously discussed.

The conservative nature of this extension shows that this structure is still fully within the standard synchronous execution model. The execution of the system is divided into logical ticks, which read inputs and produces outputs. Conceptually, deltaT is an input like any other input, and sleepT is an output like any other output. It also upholds the general requirement of determinacy: given a trace of inputs (including deltaT), the output trace (including sleepT) is fully determined.

Tick Loop Implementation While the basic design of the dynamic tick environment and its communication interface is clearly defined, its implementation depends on the platform and the specific use case. For example, it could use an interrupt routine for inputs or requires a special API for accessing time in a precise resolution.

Listing 4.1 shows a fairly generic implementation for a dynamic tick loop. It picks up the example from Listing 2.1 and runs a version of the PendulumSound component with the dynamic ticks interface, which is the

```
1  #include "PendulumSound.h"
2  #include "HardwareMockup.h"
3  int main(int argc, const char* argv[]) {
4    TickData model;
5    reset(&model);
6
7    double last_tick = curr_time();
8    while (1) {
9      double tick_start = curr_time();
10     model.deltaT = tick_start - last_tick;
11     model.theta = read_from("theta");
12     tick(&model);
13     write_to("speaker", model.sound);
14
15     last_tick = tick_start;
16     double sleep = MAX(model.sleepT -
           (curr_time() - tick_start), 0);
17     awaitInputOrTimeout(sleep);
18   }
19 }
```

Listing 4.1. Tick loop example for dynamic ticks.

model described in Section 4.2.3 but compiled with the sleep time inference that will be introduced in Section 4.4.2. Compared to the previous no-delay version in Listing 2.1a, it implements a Time Manager and waits for triggering ticks instead of immediately starting the next.

Before the start of the tick loop, the last_tick variable is declared in line 7 and initialized with the current time. Each tick execution begins with storing the start time (line 9). Then, the deltaT input is computed based on the time between the current and last tick. Afterwards, the program provides the remaining inputs, invokes the tick function, and processes the outputs. At the end of this tick computation, the start time of the current tick becomes the last tick time (line 15) and the sleep time is computed (line 16). The sleep time is based on the sleepT output but subtracts the execution time of the tick function. This compensates the fact that deltaT is a stable input for the synchronous tick function and consequently sleepT represents a sleep time relative to this point in time. A maximum function prevents negative sleep times. Alternatively, this step could raise a runtime error because a negative sleep time indicates that the program requires an execution pace that cannot be satisfied due to its own execution time. The awaitInputOrTimeout function can be considered an abstraction of a platform-specific interrupt and waiting routine. Here it is a blocking call that returns as soon as any input can be read from the hardware *or* the provided sleep time expires. Only then the loop repeats itself to execute the next tick.

Logical or Physical Time? LF uses a notion of time that includes a logical and physical timeline (or even multiple ones [LMS+20]). Events are processed in timestamp order and always on time w.r.t. to logical time, even if physically impossible. Physical time represents the wall clock time during execution and is always in advance of logical time (or equal to), potentially triggering deadlines if processing timeframes are violated. For example, if two events occur after each other but with a delay smaller than the execution time of the processing tick, it is impossible to process the second event at its physical time of occurrence. Furthermore, it is a property of the event-based processing with logical time that the program is able to recognize the event during execution, store it, and catch up on its processing. Sant'Anna et al. provide a detailed discussion on the consequences of different sampling strategies in the context of Céu [SIL+17].

Timed SCCharts neither rely on such an event-based semantics nor introduce an explicit separation between logical and physical time. The concept of dynamic ticks only relies on the time provided by the environment via `deltaT`. Hence, it is very well possible to create a tick environment that executes the SCChart with logical time, see Section 4.5.3, by hiding the natural execution lag from the program. This would keep the program in a "perfect world." As a consequence, `deltaT` will always be equal to the previously requested `sleepT`, if input events are not interrupting. This perfectly simulates the eager semantics.

However, dynamic ticks also work without this additional requirement on the environment. Listing 4.1 represents a tick environment with physical time input. In this example, the `curr_time()` function is supposed to directly access the system's clock, also illustrated by the fact that it is used to determine the execution time of the tick function in line 16. This approach can be very valuable for a modeler, as it implicitly communicates physical lag to the program and enables an adjustment of the behavior. For example, a model can measure its own execution time and modify its workload, enter a "degraded" mode if timing overruns occur, or even maintain a local logical timeline. Section 4.4.3 will discuss modeling options in timed SCCharts that deal with the exposure to a physical time input.

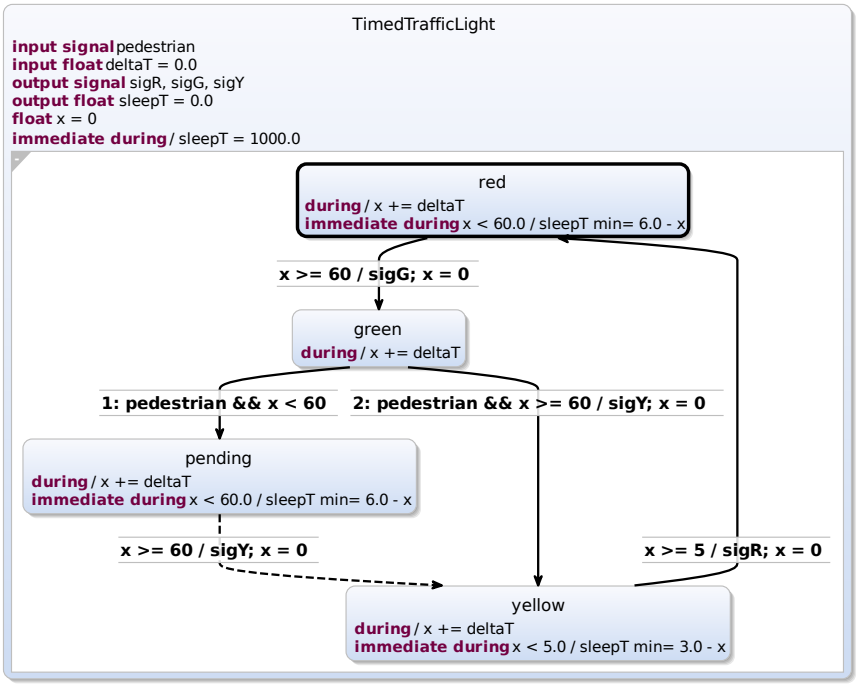


Figure 4.14. The traffic light controller SCChart after the non-concurrent clock transformation producing dynamic ticks. (Publ. in [SHM+18; SHM+20] ©2018 IEEE)

4.4.2 Sleep Time Inference from Timed Automata

An SCChart is compatible to dynamic ticks as soon as it provides the specified interface. Of course these variables can be used to manually track time and compute sleep times, but the notation of timed automata in SCChart offers a more elegant way to automatically infer sleep times. As presented in Section 4.2.2, the transformation of timed automata in SCCharts already relies on a time input named deltaT and thus is in accordance to the dynamic tick interface. The same transformation can be configured to additionally produce an overall sleep time for the model.

Transformation Figure 4.14 shows the traffic light controller example in its transformed state (based on Figure 4.4) with full support for dynamic ticks. The SCChart now declares the additional output sleepT. Additionally, the root state contains an immediate during action that initializes the sleep time to 1000. This presumably large default value is simply denotes that there is no active timeout and could be just as well infinity. The transformation can be configured to use different default values. Afterwards, the value of sleepT is updated by the states requesting an earlier wake-up time. Each state that has transitions with a trigger involving a clock-based guard creates an immediate during action for each constraint (state green is special case discussed later). If the timing condition can be met in the future (e. g., x < 60.0 for the x >= 60 transition of state red), it registers the remaining time for the trigger point of the guard (consequently, 60.0 - x) in sleepT. The min= is an update assignment that sets sleepT to the minimum of its current value and the right-hand side expression. The result is a consensus of all active time-related guards on the closest relevant triggering time. The specifics of the sleep time calculation will be discussed later.

With the provided sleep times and a dynamic tick environment, the model will now behave as illustrated in Figure 4.11f and comply with the eager semantics. The program reacts to the pedestrian input at time 40, but the state of the automaton does not change. However, as illustrated by deltaT and sleepT presented under the timeline, the dynamic ticks adapt to the input-triggered invocation and correctly compute a new sleep time of 20. After the output of sigG at time 60, no timing constraint is available. Hence, the model has to rely on inputs for triggering and defaults to 1000 in sleepT. The trace further shows that the reaction to the pedestrian event at 122.2 is likewise on time, and the output of sigR is exactly 5 sec after this event.

How to Compute Sleep Times The main task in computing automatic sleep times is to detect if and which passage of time causes a transition to be enabled in the future. The transformation uses a static analysis of the timing bounds in the outgoing transitions of states for this task. In order to facilitate its implementation, it is subject to certain restrictions in the timing constraint specification. More specifically, it considers timing constraints if the form $c \geqslant ltb$, where c is a clock and ltb some expression

for a *lower timing bound*. As illustrated in Figure 4.14, the difference between *ltb* and the current clock value determines the requested sleep time for this constraint (e. g., `sleepT min= 60.0 - x`). To simplify the detection of lower timing bounds, the implementation does not handle negations of timing constraints.[6] Furthermore, constraints that specify an *upper* bound do not contribute to the sleep time since they, considered separately, do not require time to pass to be enabled and hence would result in a sleep time of zero.

Another case in which the passage of time has no triggering effect, despite the presence of a timed guard, can be found in the `green` state of Figure 4.14. Here, both outgoing transitions primarily depend on the `pedestrian` input, and `x` only distinguishes *which* of the two paths is taken. To detect such non-triggering timing constraints, assume that the i-th outgoing transition of some state has a guard $G_i = C_i \wedge T_i$, where C_i is a condition that does not depend on time and T_i is a timing constraint. Assume that no guard is currently active, i. e., $\bigvee_i G_i = \texttt{false}$, and that T_1 specifies a lower timing bound *ltb*. If $\exists i$ such that C_1 implies C_i and $\neg T_1$ implies T_i (i. e., whenever the *ltb* has not been reached yet, T_i holds), T_1 is considered *non-triggering*. This will prevent a contribution to the sleep time computation. The implementation further simplifies this condition and assumes that C_1 and C_i are the same boolean guard, and T_1 and T_i are negations of each other. In the example, the guards on the outgoing transitions from `green` fulfill that criterion. Taking `pedestrian && x >= 60` for C_1 and `pedestrian && x < 60` for C_2, the compiler classifies 60 to be a non-triggering *ltb* and does not compute a sleep time for it.

Timing Bounds The concept of computing sleep times based on lower bounds is closely tied to the eager semantics. In a perfectly eager execution, it would be sufficient to write $\texttt{x} \geqslant \texttt{60}$ as $\texttt{x} = \texttt{60}$. However, considering real-valued time and a realistic implementation with physical time and possible timer imperfections, the first option is more robust and thus preferable. The following section will take a closer look at handling such imperfections.

Open timing intervals, specified via $>$, are not supported by the proposed concept because they would imply a request for an infinitesimally

[6] Note that this simplification does not limit expressiveness, as for example, $\texttt{!(x} < \texttt{10)}$ can be written as $\texttt{x} \geqslant \texttt{10}$.

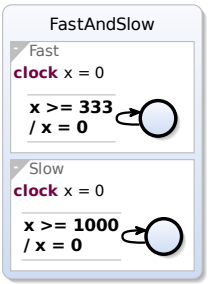
larger sleep time in a real-valued time domain, while in an integer-based time representation, it would be trivial to express the same timing bound in the $\geqslant$ notation.

4.4.3 Dealing with Physical Time

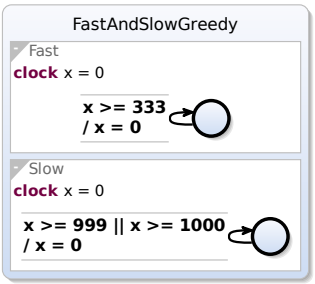
As already discussed, dynamic ticks can be operated in an environment with logical or physical time inputs. While logical time ensures a perfect eager semantics, physical time might be preferable in cases where the program should be able to detect timing imperfections and adjust the behavior of the system from within the model. In LF, watchdogs and deadlines provide such means to react to a deviation between logical and physical time. Yet, LF's notion of physical time is slightly different from the physical time input discussed here in the context of SCCharts, since physical time continues to progress during execution of an LF program. In contrast to that, the `deltaT` input in SCCharts is held stable during execution, even if determined based on physical time.

With timed SCCharts exposed to physical time, this raises the question: Can we react in time? If, for example, more time than the minimum of a specified lower bound passes, it is possible that other transitions also get enabled or disabled, which may change the expected behavior. Assume the example that a state is entered when at least 60 sec passed ($x \geqslant 60$) and is immediately (in the same tick) left when at most 80 sec have passed ($x < 80$), without any reset of the clock. With eager semantics, the state will be entered after a time of 60 and then left immediately. If the tick is delayed due to physical lag, for example to react after a time of 80 for the first time, then the state is entered but can never be left.

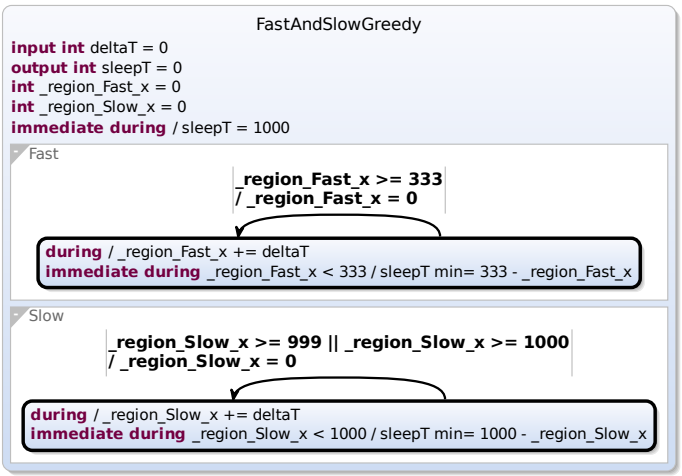
One could argue such a system is designed badly and advocate the use of logical time to perfectly match the eager semantics. Alternatively, one could consider this design a deliberate expression of a deadline. Specifically, if the execution environment approximating the eager semantics in the real world fails to meet the 60 sec trigger point by an additional margin of 20 sec, the system must not advance to the next state. Section 4.5.3 will discuss a similar model.



(a) SCChart motivating the use of soft bounds.



(b) SCChart using soft bounds in the trigger in region Slow.



(c) Transformed SCChart with soft bounds

Figure 4.15. Motivating example for using soft bounds in dynamic ticks. (Publ. in [SHM+20])

In addition to such modeling options, timed SCCharts provide strategies and language constructs to deal with physical delays in timing bounds and on clocks. This enables the user to create models that yield a more robust and desirable behavior in a realistic physical environment.

Hard vs. Soft Bounds—A Greedy Semantics An eager execution will always try to react to inputs on time. However, if two events (input or time) occur so close that their separating time falls below the execution time of the tick function, the second event can only be processed with inevitable physical delay. Since timed SCCharts are not strictly bound to an event-based processing in logical time as LF, they provide the concept of *soft bounds* to handle such situation differently. The soft approach loosens the regime of the eager semantics and leads to a *greedy* semantics.

Figure 4.15a illustrates a minimal SCCharts example to motivate soft bounds. The SCChart has two regions Fast and Slow, each one uses a timed automaton to react. Every time its threshold is reached, it resets its clock. The SCChart uses an implicit time resolution in microseconds. Hence, Slow should react every millisecond and Fast three times faster. Starting at time zero, the third reaction of region Fast will be at 999 usec, leaving only one microsecond to invoke the reaction of Slow, which might be infeasible for the environment.

With soft bounds such short sleep times can be avoided. The idea is to widen the timed reaction window of a transition, speculating to possibly "piggyback" on a somewhat earlier reaction invoked by another state. At the same time, the transition should still request its own sleep time to ensure that it is triggered. In timed SCCharts, the modeler may replace the *hard bound* `x ⩾ 1000` in region Slow by a *soft bound* `x ⩾ 999 || x ⩾ 1000`, as illustrated in Figure 4.15b. If enabled, the compiler detects this pattern and adjusts the computed sleep time, as presented in Figure 4.15c. The state now only requests a sleep time of 1000 usec, as for the original hard bound specified with `x ⩾ 1000`. However, at run time the transition may already be taken at time 999 usec, thus subsuming the sleep time of 1000 usec. This favors earlier reactions over late reactions, prevents very small sleep times, and possibly reduces the total number of reactions by processing assuredly delayed reactions ahead of time.

Hard vs. Soft Resets—Handling Physical Lag in Clocks With physical time fed to deltaT, timer imperfections affect the clocks in SCCharts. Ticks executed *before* the requested sleep duration are normal in the presence of external input events. These do not affect the timed behavior since the

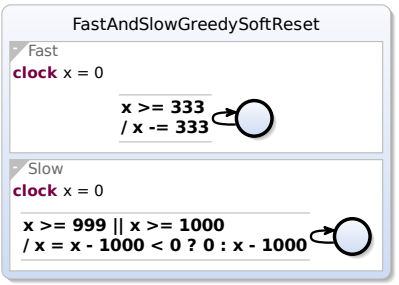
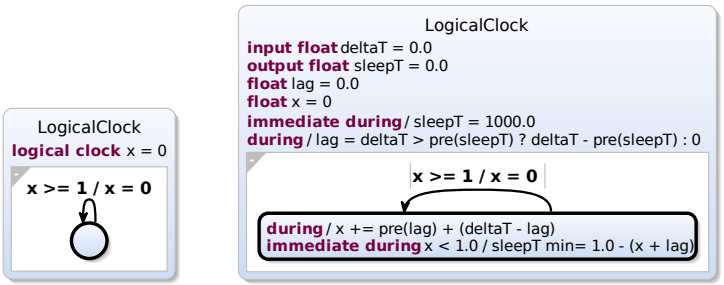


Figure 4.16. Variant of FastAndSlowGreedy SCChart using soft reset in both regions. (Publ. in [SHM+20])

clocks will register this intermediate time, compute a new sleep time, and related time constraints do yet not trigger. However, when a tick is executed *after* the requested sleep time, the additional delay time will be present in all clocks. This fact should be considered when performing a reset on a clock.

A *hard reset* sets the value of the clock to an absolute value, as presented in Figure 4.15. Alternatively, one can use a *soft reset* that takes into account the potential lag on a clock. Figure 4.16 shows a variant of the previously introduced FastAndSlowGreedy SCChart that uses soft resets in both regions. Each resets its clock x to the amount of time that exceeds the expected wake-up ($x - 333$ and $x - 1000$). Due to the soft bounds in region Slow, it is legal to take this transition with 999 usec, which would result in a negative clock value. Therefore, the maximum of 0 and $x - 1000$ is used to assign x.

A consequence of hard resets is that clocks start to drift as soon as the tick function is invoked slower than the expected wake-up time. For example, if region Fast in Figure 4.15b wakes up at 335 usec, it would reset the clock to 0 and request a sleep time of 333 usec, disregarding the 2 usec that additionally passed. Hence, the (earliest) next wake-up would be at 668 usec and this drift increases as the delays accumulate over time. This violates the set goal of temporal order and simultaneity. Soft resets compensate this accumulation of timer imperfections by leaving the lag on the clock and consequently include them in the sleep time computation.



(a) Source SCChart **(b)** Transformed SCChart

Figure 4.17. Example for using a logical clock with dynamic ticks.

The Fast region in Figure 4.16 would reset its clock to 2 if it woke up after 335 usec and consequently would only request a sleep time of 331 usec. The Slow region behaves similarly.

For period directives, introduced in Section 4.2.4, the compiler can be configured to automatically synthesize soft or hard resets.

Logical Clocks The previous example illustrates that a model is able to detect the discrepancy between the expected idealized (logical) time and the provided physical time. As a continuation of the idea, timed SCCharts provide *logical clocks* that are driven by a physical time input but automatically hide potential lag from the user.

Figure 4.17a illustrates a small SCChart that uses a logical clock. This annotation causes the transformation to synthesize a different progression mechanism for the clock, as well as an adjusted sleep time computation. In addition to the dynamic tick interface, the transformed SCChart in Figure 4.17b has a new variable lag that is set by a during action in each tick, except the initial. If the elapsed time is greater than previously required sleep time, then this deviation is stored in the variable. Otherwise, no lag is present, for example if a reaction was triggered by an input event before the wake-up time. The progression of time in x is no longer directly bound to deltaT but follows a derived logical timeline that hides the lag. For the increment of x, the current lag is removed from deltaT and the lag in the

previous tick is added because it was hidden at that time. However, this procedure only works if the sleep time computation is adjusted as well. It adds the current lag to the value of the clock x before computing the remaining time for the timing condition. This corresponds to an implicit soft reset for logical clocks and issues a compensation of the detected lag.

Since x progresses on its own virtual logical timeline, it might request negative sleep times to catch up with reactions that are already past physical time. This is similar to the delta compensation implemented in Céu [SIL+17]. Since the environment cannot physically go back in time, deltaT will not be negative, resulting in an increase of lag, both in reality and accumulated in the lag variable (deltaT - pre(sleepT)). If the program or the inputs do not grant a sufficient idle period, the logical time might not be able to catch up with physical time. However, this is in the nature of logical time processing and is likewise the case for LF or Céu.

4.5 Evaluation

From a design perspective, the use of timed automata as a notation for timed behavior in SCCharts is plausible. Both the traffic light controller in Section 4.2.2 and the Furuta pendulum controller in Section 4.2.3 illustrate that timed SCCharts provide sufficient modeling capabilities to express various time-related behavior. Aside from modeling aspects, dynamic ticks are a powerful concept for timed execution. This evaluation investigates if it can deliver in terms of performance, sparse execution, and precision (Section 4.5.1 and Section 4.5.2). Furthermore, there are some differences but also links to event-based languages, such as LF, that are worth a closer look (Section 4.5.3).

Reflection on Goals As a first informal evaluation, the initially set goals are examined for their fulfillment in the proposed concept (cf. page 100).

Determinism By accessing time only through regular inputs [HBG17] and by fully relying on the SC MoC provided by SCCharts, timed SCCharts ensure deterministic behavior.

Resilience Section 4.4.3 discussed multiple options to cope with run-time variations and imperfections, for example, the soft reset that avoids accumulations of physical lag in clocks.

Scalability All clocks in SCCharts receive their time from a single input, deltaT, only imposing a minimal management and storage overhead for each. Furthermore, the concept of dynamic ticks combines all internal timers into a single sleep time, eliminating the need for tracking timers individually.

Fine granularity Timed SCCharts are not restricted to a specific resolution or granularity of time. While floating point values enable encoding arbitrary timeouts, the synthesized types can be adjusted to match whatever granularity is required or provided by the environment. Dynamic ticks take care of an efficient execution regime that does not require superfluous intermediate reactions for matching arbitrary timeouts, as it is the case, e. g., in periodic execution.

Time composability With a single input source for time that is sensitive to ticks, timed SCCharts achieve a time composability and prevent inconsistencies as it is the case with multiform time [BS09]. In the presence of physical time, soft resets for clocks preserve the composability of consecutive uses of the same clock in relation to other clocks.

Simultaneity and order Similar to previous properties, simultaneity and order in timed SCCharts are a consequence of the single time input and the properties drawn from the synchronous SC MoC.

Lean interface Dynamic ticks come with the smallest possible interface for communicating time in both directions. They impose minimal requirements on the environment and are easy to implement in a tick environment. Section 4.5.3 will continue on this topic.

Seamless compiler integration Clocks, timed guards, period directives, and dynamic ticks are all implemented as extended features in SCCharts and seamlessly integrated into the existing compiler without further measures.

4.5.1 Sparse Pendulum Execution

One motivation for dynamic ticks in SCCharts was the observation that periodic execution and the multiform notion of time can lead to a notable overhead in reactions. This tick load consumes valuable resources, while many reactions are for the sole purpose of tracking time and have no outside effect. Section 4.3 already discussed the conceptual workings and implications of the different execution regimes. The following experiment investigates the concrete tick loads based on the Furuta pendulum models.

Setup The experiment for the Furuta pendulum was conducted in software.[7] A simulation replaced the real-world behavior of the pendulum. Its implementation is based on a simple forward-Euler simulation by Eker et al. [LEJ+02] and thus corresponds to the control logic used in the controller implementation, see Section 2.4. The simulation was configured to work at a pace of 5 msec. All involved models and source files are available online.[8]

In the experiment, three variants of the PendulumController and the PendulumSound component were simulated over an interval of 3 seconds, which is sufficient to swing up and stabilize the pendulum. The test involved the SCCharts using the multiform time approach (Section 2.4), the timed SCCharts variants (Section 4.2.3) with dynamic ticks, and the LF implementation with modes (Section 3.2). The overall program was extended to log the timing of ticks and the outputs of both components.

Behavior Figure 4.18 illustrates the observed behavior of the pendulum and its additional outputs. The pendulum (Theta) starts in a downward position ($+\pi$) and is accelerated upward by the arm. However, the induced momentum is not sufficient to get the pendulum into an upright position (0) and it starts to fall down again at about 400 msec. It passes its downward position and swings back up on the other side ($+\pi \rightarrow -\pi$). This time it reaches a higher point and is caught by the controller, indicated by the

[7] While it certainly would be interesting to conduct an experiment with a real-world Furuta pendulum and future work may make up for this, for the measurement of tick behavior a software simulation is sufficiently suited. See Section 4.5.2 for a real-world experiment with dynamic ticks.

[8] https://github.com/a-sr/furuta-pendulum

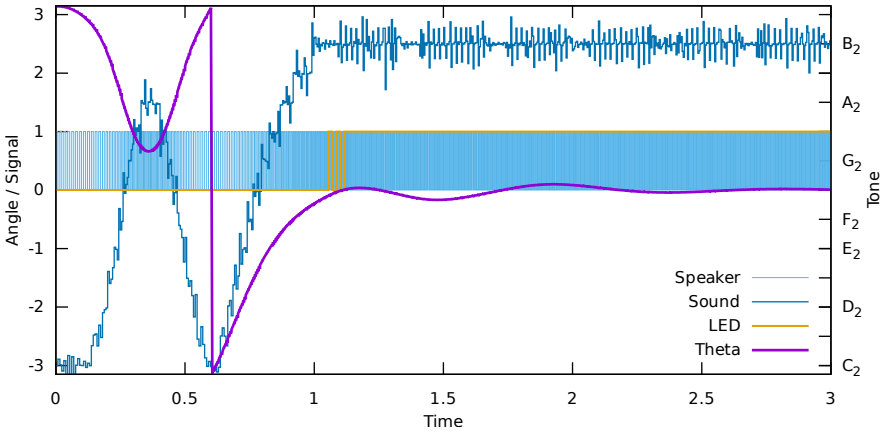


Figure 4.18. Observed pendulum behavior during simulation.

LED signal oscillating at around 1.1 sec. The catch phase is successful, and the pendulum is stabilized in its upright position. The square wave signal for the Speaker has high frequency due to the nature of the sound. The plotted Sound line indicates the notes the signal produces. As expected, they correspond to the pendulum angle. The jitter is discussed later.

This chart is based on the LF behavior but is representative for the SCCharts variants as well. All models produce the same pendulum behavior, only the timing differs, which primarily affects the sound.

Tick Load Table 4.1 shows the execution measurements for all three variants. The SCChart with multiform time produces 3001 ticks over the measurement period. It reacts every millisecond due to the need to process the msec input, plus an initial tick at time 0. The behavior of all inner modules align at multiples of this input (simulation at 5 msec, blinking LED at 15 msec) and thus do not issue additional ticks.

The implementation with dynamic ticks performs only 1208 ticks. 601 of them are required to satisfy the 5 msec pace of the simulation and the remaining ones are ticks that are requested by the PendulumSound module to produce a square wave signal with the correct cycle length.

Table 4.1. Results of the Furuta pendulum simulation.[9]

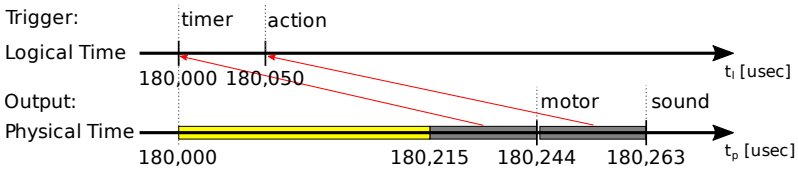
Measurement	SCCharts with multiform time	SCCharts with dynamic ticks	Lingua Franca
Number of Ticks	3001	1208	1300
Average Response Time (nsec)	144,871	196,791	181,388
Average Tick Time (nsec)	18,505	23,662	26,984

As expected, the dynamic ticks are able to reduce the tick load compared to the effectively periodic execution in the multiform variant. However, the fact that LF performs 92 ticks more requires a closer look, since LF should theoretically also correspond to an eager execution.

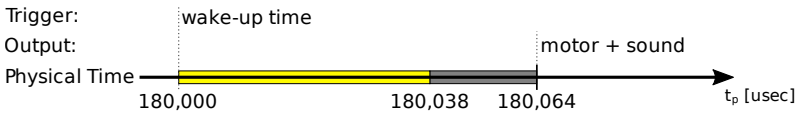
Tick Discrepancy between LF and Dynamic Ticks A first indication for a difference in the tick count was already discussed in Section 2.4. For demonstration purposes, the implementation of the PendulumSound reactor issues a zero-delay action to "immediately" toggle the sound output, which requires an additional tick execution. Yet, this design decision only accounts for 3 of the additional ticks in the LF execution.

Another aspect described in Section 2.4 is the discarding of actions that are no longer relevant. In the current version of LF, scheduled actions cannot be re-scheduled or removed. Hence, if a change in the angle of the pendulum issues an adjustment of the sound, the previously scheduled action will still trigger the reaction for the now deprecated sound frequency. The program does not change the sound output in these cases, nevertheless,

[9]The experiment was conducted on an Ubuntu 20.04 system with an Intel Core i5-7300U CPU and 16GB RAM. The code was compiled using the GCC compiler with optimization level O3. The response time represents the delay between the time produced outputs and their theoretical time to happen (i. e., logical time/wake-up time). All three implementations use the POSIX function clock_nanosleep to wait between ticks. The tick time is measured from the time of leaving the sleep function to its reentering. Hence, the measurement covers the business and probing logic (which is identical in all three models), the model-specific control structures, and the language-specific runtime infrastructure. The average does not include the initial and last tick, since these are outliers due to the need to set up the program and finalize the results. The SCChart models were synthesized using the netlist approach and the LF program was generated as a single-threaded application.



(a) Lingua Franca execution



(b) SCCharts execution with dynamic ticks

Figure 4.19. Comparison of a representative execution sample, illustrating the reduction of ticks in SCCharts with dynamic ticks. The sample shows the ticks on their respective timelines. The yellow boxes indicate the wake-up lag and gray ones the tick computation time.

it results in 29 additional ticks for LF in this experiment. While future versions of LF will probably provide an API that facilitates handling such cases, it is also notable that the concept of sleep times in dynamic ticks naturally handles such situations. In each tick both the clock t and the timing threshold (duration) may change, see Figure 4.7, and consequently the computed sleep time automatically shifts accordingly.

With additional 32 ticks now accounted for by the reactor implementation, this still leaves a difference of 60 ticks. These are a result of LF's logical time and event-based processing. At a given time, the SCChart reacts to the provided inputs, which in the experiment is physical time (provided via deltaT). As presented in Section 4.2.3, the implementation is aware of physical time lag and uses soft resets. In contrast to that, LF maintains an event queue that tracks all events in logical time and processes them even if physical time has advanced beyond that point. This results in additional ticks in certain cases.

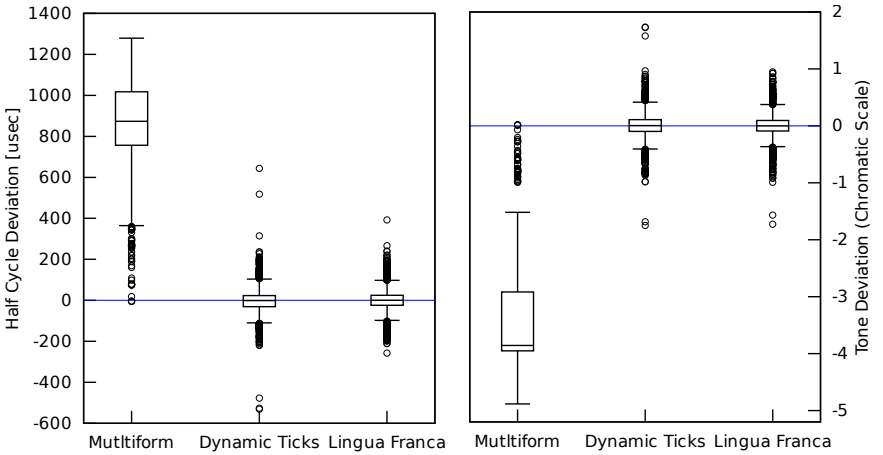
Figure 4.19 illustrates a characteristic sample found in the experiment. The LF execution is associated with a logical and physical timeline, see

Figure 4.19a. At 180,000 usec the timer event that triggers the simulation and indirectly the PendulumController is due. On the physical timeline, the start of the tick is delayed by a lag in the sleep process (yellow bar). The tick computation starts at 180,215 usec and finishes with the motor output 29 usecs later. Table 4.1 also lists the average lag and execution time as general reference. On the logical timeline all of this happened instantaneously at 180,000 usec. Then, there is a subsequent event from the logical action in PendulumSound, scheduled for 180,050 usec. In logical time, this event is processed on time in a separate tick. On the physical timeline this tick lines up directly after the previous one because it is already past due. The sound signal is ultimately updated at 180,263 usec.

In the SCCharts execution in Figure 4.19b, there is also a tick planned at 180,000 usec for the simulating and motor control. Again, the actual output is delayed by the physical wake-up lag and execution time; in this case a relatively low response delay (cf. Table 4.1). However, there is no additional tick and the timelines are individually scaled to illustrate their conceptual correspondence. Instead, the reaction executes both the PendulumSound and PendulumController component, and updates both outputs at the same time. This is the case because there are no timestamped events or logical time for the SCChart. When the tick starts, there are 180,038 usecs on the clock and the program reacts in accordance to these inputs. Note that the planned time for a sound signal change differs between the SCChart and LF, because the SCChart uses physical sleep times for the cycle timing, while the PendulumSound reactor uses logical actions. Therefore, the sound reaction in SCCharts is not due at 180,050 usec but earlier[10], which justifies a sound output with 180,038 usecs on the clock.

One could argue that from an event-driven perspective, the dynamic ticks use a greedy approach, as described in Section 4.4.3. However, this does not mean the behavior of the SCChart is not in accordance with its semantics, it simply uses a different time input and correctly reacts in accordance to that. Furthermore, if the SCChart would use a logical clock,

[10]The experiment did not capture the internal variables of the execution, hence, the exact time is unknown. Yet, it is certain that it was due between 180,000 and 180,038 usec because the output was produced, and the triggering time was a multiple of 5 msec and thus originated from the simulation and not the PendulumSound component.



(a) Wavelength deviation **(b)** Tone deviation

Figure 4.20. Deviation of the sound signal from the expected angle-dependent note.

as described in Section 4.4.3, it would simulate the LF behavior and issue a negative sleep time to catch up on the sound signal event. However, since this point is already physically gone, it would only further delay the sound output as it is the case in the LF trace. Yet, it is important to note that the use of logical clock does not supersede an external event queue with logical time, because it only introduces an internal notion of logical time and this only for reactions to time. As illustrated by Sant'Anna et al. in the context of Céu [SIL+17], an external event queue in combination with logical time has the advantage that no external events are missed, as it can be the case with other sampling methods. However, from the design perspective of SCCharts, this is a question of the environment and how it samples and handles inputs.

Sound Jitter Figure 4.18 illustrates that there is a jitter in the sound signal. Figure 4.20 presents the individual deviations of the produced sound from the timing of the expected note. Since the multiform time variant is only modelled with a 1 msec granularity, it performs expectantly worse than the

other two variants. Figure 4.20a shows that the half cycle length of the signal is often nearly 1 msec longer than desired. This is caused by the fact that the condition on the trigger rounds up to the next millisecond, resulting in an additional delay. This is further intensified by physical delays in the wake-up process. Since notes do not progress linearly in their wavelength, Figure 4.20b presents the deviation in terms of notes. Here, one can see that the 1 msec pace causes the audible sound to be 3 to 4 semitones below the expected one.

In both dynamic ticks and LF, the deviation is much smaller and the program is mostly able to strike the right note. The jitter is caused the variations in the wake-up lag. If a small lag is followed by a larger one, it results in a longer wavelength. In the inverse case, the observed cycle is shorter.

The experiment shows that dynamic ticks in SCCharts and LF are on the same level when it comes to timing of the sound. However, a significant factor in the jitter and strength of the deviation is the wake-up lag. With over 100 usec (Table 4.1) on average, it requires further investigation in the future, see Section 6.3.3.

4.5.2 A Hard Real-Time Demonstrator for Dynamic Ticks

To investigate the performance of dynamic ticks in a more time-sensitive context, a real-world demonstrator was developed, built, and tested by Boysen under my supervision [BSH20a; BSH20b; BSH20c]. The goal was to create a reasonably cheap and easy to implement demonstrator that embodies a hard real-time problem with scalable timing challenges. This should then act as a testbed for an SCChart with dynamic ticks to investigate its real-time capabilities. The result was the "Disk-and-Sticks demonstrator," in short DS demo.

Setup Figure 4.21 shows an annotated image of the DS demo setup. On the left is the Motor Controller receiving an input from a Signal Generator to control the speed of the experiment. In this case, the controller is a Field Programmable Gate Array (FPGA) board, but the setup is designed for interchangeable controllers. The controller is connected to two Motor Drivers.

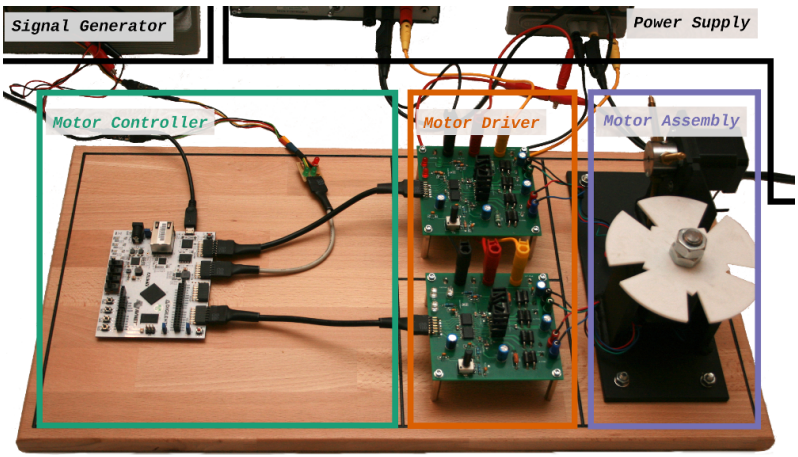


Figure 4.21. DS Demonstrator setup with annotations [BSH20b]. (©2020 IEEE)

These driver boards are specifically designed [BSH20c] to give the controller *direct* control over the power supply to the two stepper motors in the Motor Assembly on the right. At the heart of the demonstrator is the assembly of the two stepper motors, arranging the disk and sticks in a 90 degree angle. Figure 4.22 presents a more detailed schematic. If the motors are running synchronized in an exact 3-to-5 ratio, the sticks can pass through the disk.

Stepper motors are well-suited for such a task, since they lock into step locations and thus provide good repeatability and precision. Additionally, when controlling a stepper motor directly, the steps in the real world directly correspond to reactions in software, imposing real-time requirements on the software.

Timing Challenges For the controller, the task at hand is to drive two stepper motors in a synchronized way. Any deviation would let the sticks and disk collide and destroy the assembly. Stepper motors move based on a rotating magnetic field created by magnetizing coils. In the DS demo, the motors consist of a permanently magnetized rotor, surrounded by a stator containing two separately controllable coil sets. Both the stator and rotor are multitoothed, and energizing a set of coils will cause the rotor to snap into

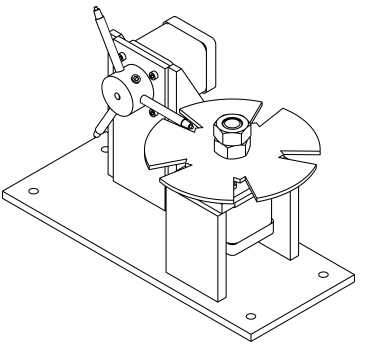


Figure 4.22. Technical drawing of the DS motor assembly [BSH20b]. (©2020 IEEE)

the next position (step) by aligning itself to the magnetic fields. The direct control requires the controller to decide which and when to magnetize coils. An increasing target speed automatically raises the required number of steps both in the motor and the controller.

The stepper motors in the DS demo have a nominal voltage of 4.2 V. However, they can easily reach about 11,000 rounds per minute when powered with 30 V and without the disk/stick to reduce the load. This raises an additional challenge. Increasing the voltage also increases the current drawn by the coil when powered over time. At voltages beyond the nominal value, it will eventually destroy the coil. Hence, to safely operate the motor at higher voltages and speeds, the power supply needs to be temporarily decreased, to limit the resulting current. While it is common to handle this aspect in the motor driver, the boards in the DS demo pass this task to the controller to impose another critical hard real-time requirement. The driver boards support the controller in this task by sensing the current drawn by the motor and signaling an overcurrent event.

Controller Model The DS demo controller is modelled in SCCharts and manages the power supply to each individual coil in both motors. For an in-depth presentation of the model, please consult the corresponding publications [BSH20b; BSH20a; BSH20c]. A crucial component of the controller is the OverCurrentProtection illustrated in Figure 4.23. For each coil output,

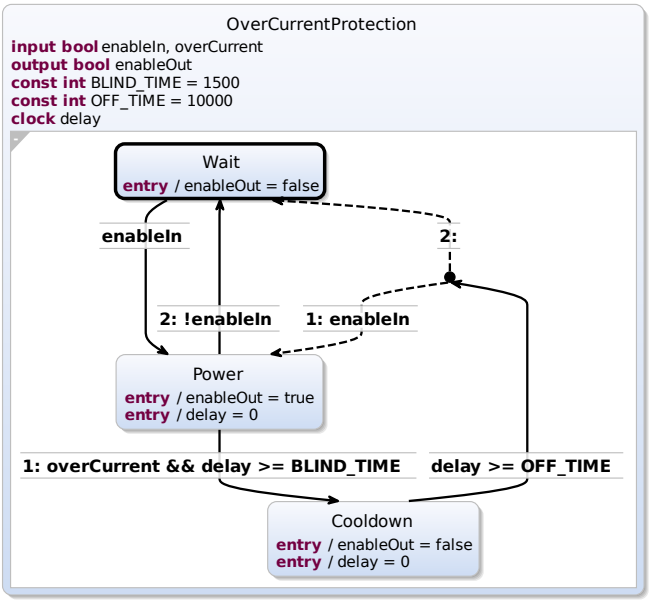


Figure 4.23. The timed SCChart handling overcurrent protection in the controller [BSH20b]. (©2020 IEEE)

one instance of this SCChart protects the coil from overcurrent damage by temporarily disabling the power supply for a constant time. During this off-time, the coil is discharged through protection diodes in the driver board.

The SCChart has three states: Wait, Power, and Cooldown. In normal operation, it simply passes on the enableIn input to the enableOut output. When the motor is powered, an overcurrent event may be reported by the driver board. This causes the SCChart to go into the Cooldown state and disables the coil. However, this transition has an additional timing guard that only enables this transition if the BLIND_TIME has passed. The model implicitly uses an integer clock with nanosecond resolution, hence, the threshold is at 1500 nsec. This blind time has its origin in the parasitic induction of the resistor used to measure the current. The Cooldown state

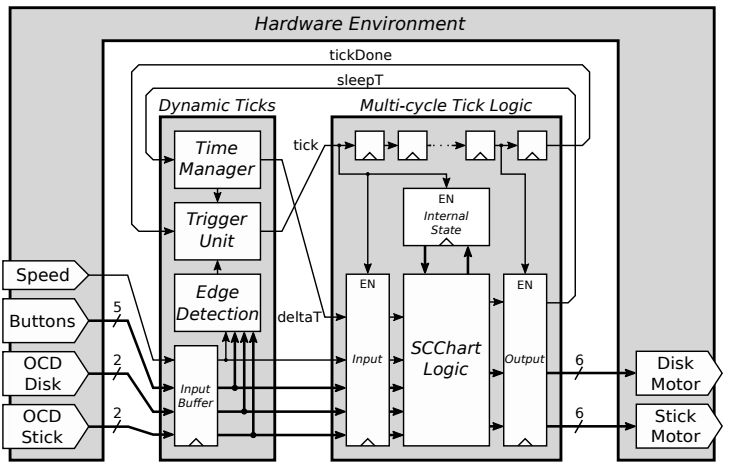


Figure 4.24. General structure of the dynamic tick environment with multicycle tick execution, implemented in the DS demo [BSH20b]. (©2020 IEEE)

is left after 10 usec. Then, the SCChart either enters the Power state if the controller still requires a magnetization of that coil, or continues in Wait if the coil was switched off in the meantime.

Dynamic Ticks for FPGAs The demonstrator setup is designed to support different controller hardware. In the experiments, a Rasberry Pi and an FPGA were used. While the Pi can use a classical software environment, as in Listing 4.1, to create a dynamic tick environment, there are different options for the FPGA.

The simplest approach is to synthesize the SCChart into a netlist in VHDL, add input and output processing, and deploy that directly to the FPGA [Joh13]. However, this approach restricts the maximum speed of the FPGA's base clock to the computation time of the generated logic. One motivation for dynamic ticks is to detach ticks from these strict periodic regimes to facilitate more precise reactions in between. Hence, Boysen developed a multicycle tick environment with dynamic ticks. Figure 4.24 illustrates the structure of this environment as a concretization of the concept in Figure 4.13.

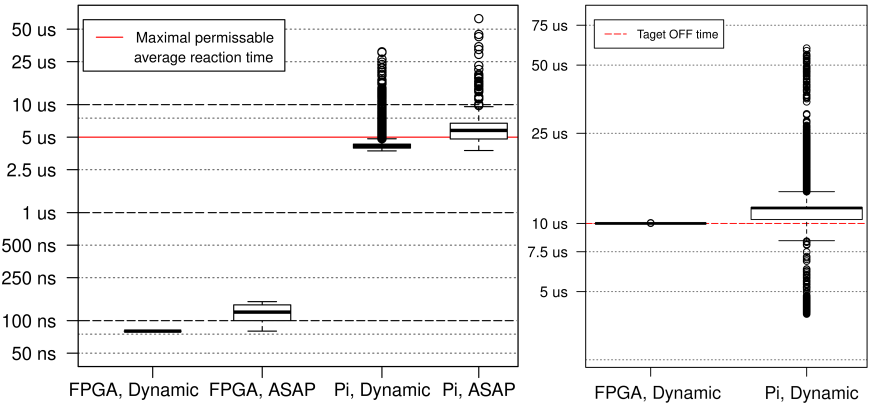
In this environment, the stateless SCCharts Logic is surrounded by various registers. Inputs are held stable during tick computation by the registers on the left. They only update (via EN) at the start of the tick, indicated by the tick signal, similarly for the Internal State and Output. During tick computation the tick signal ripples through the registers on top and finally indicates the end of the computation (tickDone). The Trigger Unit in the Dynamic Ticks section only starts a new tick if no tick is currently executing (tickDone) and it is triggered by an input event via Edge Detection or an expired sleep time.

This design enables a base clock speed for the FPGA that is a fraction of the tick logic's computation time and thus facilitates more fine-grained input sensing and wake-up times. The clock speed is determined at compile time by selecting the number of ripple registers in relation to the logic's computation time.

Response Time The DS demo offers a realistic real-world environment imposing scalable timing requirements to evaluate dynamic ticks. Figure 4.25 illustrates the results of two experiments conduced both with the FPGA and the Raspberry Pi.[11] In a first experiment, the response time of both controllers and the dynamic tick environment was tested to expose physical lag. It represents the delay between the input of the function generator and the resulting output change. The experiment was run at a relatively low speed with 400 steps per second and without the disk and stick setup, to generally assess the capabilities of the controllers, while protecting the assembly.

Figure 4.25a illustrates the response time results. The Pi performs worse than the FPGA, as expected for a general purpose processor with a Linux operating system. The horizontal red line indicates a response time that enables safe operation with correct overcurrent protection (5 usec, to reliably produce the 10 usec off time in Figure 4.23). The Pi with dynamic ticks barely meets this threshold. Furthermore, the fact that this Linux kernel is not real-time capable and may interrupt the control process results in

[11] The FPGA board is a Digilent Arty A7 35 and the Raspberry Pi a model 3B. The same SCChart controller was compiled to VHDL and respectively C, using the netlist approach. For the measurements, the FPGA board itself was used as a logic analyzer. However, when probing the FPGA this way, the analyzer and controller logic share the same 100MHz clock. To account for that, the results include a 10ns offset in response time (worst case advantage) for the FPGA.



(a) Comparison of response time between an ASAP execution (no-delay in Section 4.3) and dynamic ticks.

(b) Comparison of deviations in the produced off time with dynamic ticks.

Figure 4.25. Comparison of the FPGA and Rasberry Pi controller in experiments on the DS demo (logarithmic scale on both y axes) [BSH20b]. (©2020 IEEE)

various outliers significantly exceeding the mean. This effect is not present on the FPGA.

Figure 4.25a also shows the difference of dynamic ticks to a no-delay triggering (ASAP). On the FPGA, the dynamic tick environment has a constant response time, while ASAP causes a notable variation. This substantiates the claim in Section 4.3 that executing ticks without expecting relevant behavior can obstruct and delay reactions with crucial outputs. A similar variation can be observed for the Pi.

Furthermore, the ASAP triggering illustrates how many consecutive ticks are possible on the platform. On average, the Pi was able to perform about 260,000 ticks per second and the FPGA 1.25×10^7. The dynamic tick execution only issued about 1000 ticks per second.

Timing Precision As already discussed in the context of sound in the Furuta pendulum, variations in physical lag can cause outputs with a fixed distance to deviate from the modelled behavior. In a second experiment

with the DS demo, the speed was set 0 and the motor was supplied by 10 V causing a constant need for the overcurrent protection, illustrated in Figure 4.23. Figure 4.25b shows the measured deviation of the control outputs from the expected 10 usec long off-time. Again, the timing of the FPGA is nearly perfect, except for a few minor outliers. The results for the Pi show stronger deviations, caused by the high variation in the response time.

Compared to results for the pendulum sound deviation in Figure 4.20a, it is notable that the deviation interval on the Pi is much smaller in absolute terms. Despite a lower processor frequency on the Pi, the deviation interval is around 50 usec for the off time versus 200 usec in wavelengths for sounds. Similarly in Figure 4.25a, there is only a peak of around 25 usec in additional response time delay. The most probable explanation is the fact that Boysen did not rely on the system's sleep function but used an implementation performing busy waiting. While this contradicts the idea of reducing resource consumption, it also shows that the sleep and wake-up process is an important factor in the timing precision, and should be furthered investigated, see Section 6.3.3.

Furthermore, the deviations in Figure 4.25b tend to lean towards positive values, while in the sound signal in Figure 4.20 they are distributed more evenly. This is a consequence of the hard reset in the overcurrent protection SCChart in Figure 4.23, which ignores additional physical lag on the clock and thus promotes positive deviations.

Final Results In the final demonstration, the FPGA was able to operate the entire DS demo setup with up to 100 stick/disk crossings per second. This corresponds to 1,200 rounds per minute on the disk. With 400 steps per motor rotation, and at least 10 ticks per step due to sampling/synchronization logic, this corresponds to 80,000 ticks per second.

4.5.3 Event-Based Designs

There is a difference between the processing of timestamped events in LF and dynamic tick execution in SCCharts, as already illustrated by the skipping of some ticks in the presence of physical lag, see Figure 4.19.

This disparity continues in the interface between the model's logic and its runtime environment. While dynamic ticks rely on deltaT and sleepT, an event-based system, such as LF, Céu, or the SSM, see Section 4.1.2, require an external event queue that stores, orders, and processes pending events.

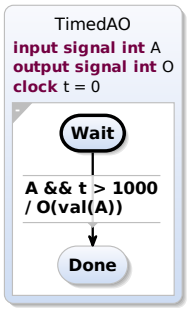
Observations The dynamic tick interface is the smallest possible communication channel for time in and out of a model. With no further requirements on the communication of time, this places a great degree of freedom on the developer. Similar to the concept of the tick function itself, it facilitates an application-specific implementation, while ensuring only the deterministic input output behavior of the program itself. The program can be provided with a notion of logical time or with physical time inputs, raising the need and opportunity to address timing imperfections in the model, as discussed in Section 4.4.3.

In LF, the time model is more specific, as it provides logical time, as well as physical time that progresses during execution. This especially facilitates the modeling of distributed system with different machine clocks, an area that is not yet substantially supported by SCCharts. In turn, this event and time architecture also increases the effort to implement a platform-specific runtime environment.

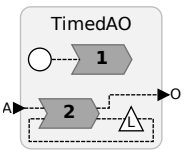
The tick environment for the FPGA in the DS demo in Figure 4.24 appears relatively complex but only consists of components that are quite easy to implement. Creating support for VHDL in LF will be more demanding, since it requires implementing the event queue and execution infrastructure for reactions. While this is a one-time effort, the implementation also has to be reasonably platform-independent, since different FPGA boards come with different capabilities. This is a concern that SCCharts, with its tick function approach and minimal environment requirements, simply passes on to the developer and the concrete use case.

Timed SCCharts Embedded in Lingua Franca While timed SCCharts with dynamic ticks do not require an explicitly event-based environment, they are still suited to operate in such a context. To demonstrate such a scenario, we revisit the concept of SCCharts embedded in LF from Section 3.4.4.

Figure 4.26a illustrates a timed variant of the AO SCChart that now only emits O if an A occurs at least 1 microsecond after the start. Figure 4.26b



(a) SCChart



(b) Embedding LF reactor

Figure 4.26. Timed variant of the AO SCChart and its embedding Reactor.

```
1  target C;
2  reactor TimedAO {
3    input A: int;
4    output O: int;
5    logical action sleep: time;
6    state tick: time = 0;
7    state scchart: {=TickData=};
8
9    reaction(startup) {=
10     reset(&self->scchart);
11   =}
12   reaction(A, sleep) -> O, sleep {=
13     if (A->is_present ||
14         (sleep->is_present && sleep->value == self->tick)) {
15       self->scchart.deltaT = lf_time_logical_elapsed() - self->tick;
16       self->scchart.A = A->is_present;
17       if (A->is_present) { self->scchart.A = A->value; }
18       tick(&self->scchart);
19       if (self->scchart.O) { lf_set(O, self->scchart.O_val); }
20       self->tick = lf_time_logical_elapsed();
21       lf_schedule_copy(sleep, self->scchart.sleepT, &self->tick, 1);
22     }
23   =}
24 }
```

Listing 4.2. Source code of the TimedAO reactor embedding the SCChart.

shows the embedding LF reactor, with its source code listed in Listing 4.2. In addition to the previous implementation in Figure 3.12, this reactor creates a dynamic tick environment for the generated tick function of the SCChart. In line 15, it provides the model with the logical time elapsed since the last tick. The sleepT output is used to schedule the sleep in line 21 that will trigger a wake-up. Similar to the procedure in the PendulumSound component described in Section 2.4.2, it requires the handling of deprecated actions. Hence, the action carries the tag of tick that it corresponds to and line 14 ensures that only the sleep time computed at the most recent tick can actually trigger a tick in the SCChart.

With LF's runtime infrastructure in mind, this design illustrates how an event-based dynamic tick environment with logical time execution implementation for SCCharts could look like. This design was tested in the context of the Furuta pendulum[12] and produced the same behavior as the LF implementation.

Modeling Internal Events in SCCharts In addition to managing external input events, the event queue in LF also handles the events produced by actions and timers. It stores them and advances in time until the next event is due. The same fundamental principle is also used in Céu and the SSM. In timed SCCharts, a similar effect results from the sleep time computation. The minimum function in the sleep time update, described in Section 4.4.2, considers all active timeouts and determines the closest in time to request a wake-up at that time. The main difference is that the timeouts are not stored as events in an external queue. With this in mind, we can investigate a timed SCChart example that strives to simulate a simplified version of LF's actions.

Basically, scheduling an action and reacting to its occurrence could be expressed in timed SCCharts by a single transition. A timing guard would carry the scheduled delay and reaching the target state would indicate the occurrence of the event. However, LF also provides deadlines that match the event scheduled on the logic timeline against the physical timeline. The idea of the following example is to bundle this combined behavior into a dedicated data type for an event.

Fundamentally, an event produced by an action is a signal in the sense of synchronous languages. It is present only at a specific time and otherwise absent, additionally it may carry a value. Section 5.5.2 will describe how the new OO features in SCCharts can be used to implement signals as user-defined types by modeling an SCCharts-based class. In anticipation of these features, the TimedSignal will present an LF-inspired event class that can be scheduled for future occurrence based on logical time and can indicate a deadline violation based on physical time. To keep the example simple, the model will only support a single future schedule per TimedSignal and omits carrying payloads.

[12] https://github.com/a-sr/furuta-pendulum/tree/master/lf/scchart-embedded

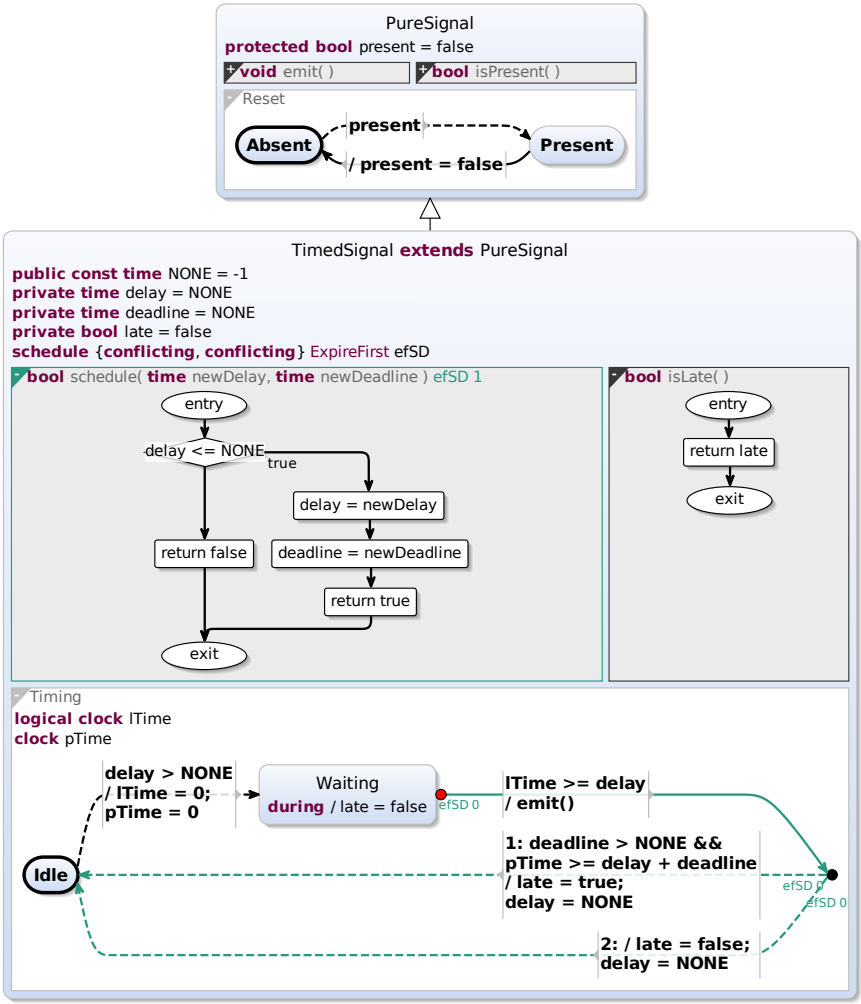


Figure 4.27. The TimedSignal class in SCCharts extending the basic behavior with a scheduling and deadline option.

The Timed Signal Class Figure 4.27 illustrates the TimedSignal class modelled in SCCharts. It extends the PureSignal class from Figure 5.17 (presented later in Section 5.5.2) to inherit the basic reset behavior and an interface for emission and presence checks. Figure 4.27 recapitulates this model and puts both in a generalization relation.

The TimedSignal has three private variables, one for storing the scheduled delay, one for a deadline offset, and a late flag that indicates a violated deadline. The public constant NONE is used to indicate the absence of scheduled events or a deadline. The class provides two additional methods, schedule to issue a future presence of this signal and isLate as a getter for a deadline violation. The main behavior of the TimedSignal is specified in the Timing region, which declares the clocks lTime and pTime. The use of a logical clock mimics LF's event processing behavior, i. e., it may try to catch up on events in the past, contrary to the behavior discussed in Section 4.5.1. The use of SCCharts' regular clock for physical time does not fully match the LF semantics, which checks the current physical time during execution, whereas this implementation uses the physical time at the start of the tick execution.

The diagram in Figure 4.27 visualizes the method's bodies in a CFG notation. The schedule method first checks if there is currently a pending event (delay <= NONE) and only if this is not the case, it sets the delay and the potential deadline. In the end, it returns if scheduling was successful. As a consequence, only a single future occurrence can be scheduled. The example could be further extended to support a fixed or dynamically sized queue, or alternatively reschedule the existing event. This would also provide an opportunity to permit multiple concurrent calls to schedule, which is prohibited in this single element queue implementation.

If a future occurrence of a TimedSignal is scheduled, the Timing region switches from Idle to Waiting, resetting both clocks and, delayed by one tick, the late flag. When the delay expires in relation to the logical time, the TimedSignal emits itself and resets the deadline. Immediately after (modelled as a transient connector state), it checks if there is a deadline and whether it is violated in regard to the physical clock, setting late accordingly.

While this example does not replace an external timestamped event system, which in turn could enable decentralized distributed execution as in LF,

it at least illustrates a notational bridge to synchronous modeling in SCCharts. Any SCChart could now instantiate a TimedSignal (via **ref** TimedSignal ts) and invoke its methods in triggers and effects, e. g., ts.schedule(2, 0.5) or ts.isPresent() && !ts.isLate().

Scheduling Intricacies in the Timed Signal Model This implementation of the TimedSignal supports scheduling a new event in the same instant the previous one expired, i. e., the TimedSignal instance is present. Yet, since all invocations of schedule will be concurrent to the Timing region and both will read and write the delay and deadline variables, it cannot be scheduled with the IURP because this would result in cyclic dependencies. Hence, there is the SD efSD (cf. Section 4.2.2) that establishes an "expire first" regime and orders the check and reset of the current delay and deadline (efSD 0) before potential invocations of the schedule method (efSD 1). The use of a method facilitates this design because it allows to indirectly subject any invocation of this method to this regime by associating the SD with the method's declaration. This aspect is further discussed in Section 5.4. The strong abort, used in the outgoing transition of the Waiting state, can preempt the sleep time computation that will be synthesized into the Waiting state and allows to first process the timer expiration check and then, upon reschedule, compute the sleep time based on the new delay value.

Chapter 5

Object Orientation

The OO paradigm has proven itself as a powerful design and programming concept that facilitates abstract and modular design of large and complex systems. Unlike in the previous chapters, the Furuta pendulum cannot sufficiently act as a motivating example in this case. It is deliberately chosen as a small introductory model tailored to a concrete use case, whereas the strengths of OO become particularly evident when used in larger software systems that require some form of abstract reusable components or generic functionality. Nonetheless, there are again two research aspects driving the proposed extension of SCCharts;

1. the integration of OO modeling capabilities into the domain of synchronous languages and statecharts modeling, in combination with a pragmatics-aware language design focus; and
2. an advanced interface to OO host languages that provides flexible strategies for establishing deterministic behavior.

Object-Oriented Modeling (1.) Most general-purpose programming languages popular today[1] support OO concepts, such as encapsulation of data and functions, inheritance, and message passing. In software engineering, the OO paradigm is often combined with a model-based approach, for example in UML, to create well-designed software architectures. Today, many software engineers are well-trained in programming languages, such as Java, C++, C#, or Python, so that OO design techniques are second nature to them. It is compelling to exploit the benefits of OO also in a specialized

[1] https://www.tiobe.com/tiobe-index/

domain, as is it the case with synchronous languages for embedded and safety-critical systems.

Actor-based languages are already closely related to objects, since they embody the message passing between objects and combine it with a notion of concurrency [Agh86]. Therefore, many of these languages support OO features, such as a class-based design, inheritance, or subtyping [Agh86; LLN09]. LF is not different and supports methods, inheritance, overriding reactions, and type parametrization.[2] With dataflow SCCharts, there is also an actor-based notation in SCCharts that can benefit from such features.

However, the focus of this thesis is the integration into the more control-flow-oriented statecharts notation of SCCharts. An OO notation in SCCharts facilitates expressing the internal behavior of objects as statecharts, enables user-defined types, and provides more reusable or easily adjustable implementations. With Section 4.2.3 and Section 4.5.3, the previous chapter already illustrates some of these aspects. Furthermore, a model-driven approach also facilitates integrating other design methods, such as class diagrams. While such an approach is not new, best exemplified by Harel's O-charts [HG96], its usability is often subject to conceptual limitations due to a graphical syntax. Once again, a pragmatics-aware language design that combines classical OO programming with graphical OO designs methodologies can mitigate notational obstructions in the modeling process.

Host Language Objects (2.) Aside from "bare-metal implementations", OO programming languages become more prevalent, even in embedded systems. In an embedded market study from 2019 by Aspencore with nearly one thousand participants, 39% of developers name an OO language as the primary language for their next project [Asp19]. Especially for a synchronous language such as SCCharts, which also aims to provide a modeling environment beyond classical embedded systems, one needs to take this trend into account. For example, SCCharts are currently commercially used in a Java-based tool for operations control of railway lines; and in a Bachelor's thesis by Raschkowski [Ras21], SCCharts were integrated into a game engine written in C++. In such contexts, the classical approach of

[2] Some of these features were developed relatively recent and are not supported in all target languages yet.

an input output signaling interface can quickly reach its limits, and a more direct interaction with the OO host environment is required to apply effects or coordinate external behavior.

The design of LF that directly embeds the target language into the coordination layer naturally adapts to an OO programming language. In SCCharts, host functionality is integrated into the expression language to provide an abstract synchronous programming layer, e. g., with external functions presented in Section 2.3.3. This concept is borrowed from Esterel and was initially designed for C as host language. However, this impairs the integration of objects because it relies on a parameter-based causality interfaces and the assumption that functions are free of side effects. Yet, it is in the nature of methods to hide internal data from their interface and have a side effect on the object itself.

Hence, in order to interact directly with host language objects from within SCCharts or any synchronous language, these objects require an OO-aware representation. To facilitate deterministic concurrent use, special scheduling regimes are required to handle causality relations hidden by methods. The idea is that an object itself should specify the regulations for its deterministic access and the program should automatically adhere [AMP+18].

Goals The main goal in this chapter is to harness the benefits of the well-established OO paradigm to improve modeling of synchronous systems and embrace the presence of objects in the host language. While there are many OO languages and some synchronous languages already support some object-based capabilities, SCCharts provides a unique context that combines modern pragmatics-aware statecharts-oriented MDE with a powerful and sound synchronous semantics. To focus the effort of this OO extension, the language design is guided by the following principles.

Conservative The introduced OO features should be carefully selected and conservatively restricted to prevent violation of fundamental virtues of synchronous languages.

High-level The OO modeling capabilities should not require a low-level host language support for OO but offer an implementation as extended features. This would make the benefits of an OO design also available

to classical embedded targets, such as C. For host language objects, a black-box approach is expedient that is not bound to specific OO host languages or relies on parsing external code.

Pragmatics-aware The language design should offer a common OO programming syntax and experience, while utilizing the integration of graphical views to enable OO design methodologies and dedicated visualizations.

Concurrent New OO features should embrace the built-in concurrency of synchronous languages and, where possible, aid in designing concurrent behavior.

Deterministic The semantics should fit seamlessly into the synchronous paradigm and should not introduce any non-deterministic behavior.

The concepts of *OO SCCharts* and *deterministic host objects* presented in this chapter are designed along these lines and provide an OO extension to SCCharts, while also representing approaches applicable to other languages in this context. In contrast to previous work, see Section 5.1, they include modern developments on pragmatics-aware modeling and flexible scheduling regimes, such as SDs. The range of OO programming features and functionalities is wide and implementing all of them easily exceeds the scope of this thesis. Therefore, the proposed concepts focus only on some core features and conservatively restricts advanced functionality that requires more extensive analyses, e. g., to ensure determinism. While this enables an investigation in terms of language design and OO modeling, it also lays the foundation for future extension of functionality.

Outline Section 5.1 presents an overview of the related work. Section 5.2 starts with a brief discussion on OO features and an assessment of their applicability to the domain of synchronous languages. Next, Section 5.3 will introduce the OO modeling capabilities in SCCharts. Then, Section 5.4 will discuss the challenges of integrating host language object in a synchronous context and will present a concept that utilizes flexible scheduling regimes to ensure deterministic object access. Finally, Section 5.5 evaluates the new design capabilities of the proposed concepts.

5.1 Related Work

The OO paradigm has a long history and a variety of programming languages contributed to the notion of objects [Cap03]. Today, there are many languages that have been developed with OO at its core, such as Java or C++. Some others received an OO makeover or extension, for example Ada in 1995 [Ada16], O-charts by Harel [HG96], OO Petri Nets [Ess96], OO-VHDL [SMC95], Objective ML (later OCaml) [RV98], ObjectCurry [HHN01], or Actors [Agh86; LLN09]. Additionally, OO programming languages are accompanied by powerful analysis and design methodologies; most prominently in the form of the UML [Obj11].

In order to focus the scope of this section, it covers relevant work on OO in embedded systems, synchronous languages, and statecharts.

5.1.1 Embedded Systems

In the context of embedded systems, C is still the predominant programming language, but OO languages are also gaining a place [Asp19]. Safety often plays an important role in this context and is usually standardized by the industry, for example in DO-178C [DO-12] for avionics or ISO 26262 [ISO18] in the automotive industry. Some languages are specifically built to facilitate formal verification or certification, for example SCADE or Ada. However, this task is more complicated for general purpose languages, such as C.

The MISRA Standard for C++ The industry often uses guidelines and language subsets to improve robustness and safety of software. For example MISRA-C++ [MIS08] by the Motor Industry Software Reliability Association (MISRA). The defined language subset can be considered a best practice approach established in industry rather than based on formal methods. It consists of large sets of informal rules, for example prohibiting recursion or the use of union types. Regarding OO, all basic features of C++ class design are generally retained. Yet, they are often restricted, for example multiple inheritance is forbidden such that classes may not be derived from more than one non-interface base class, just like in Java.

These guidelines show that there is a demand for C++ and consequently OO in the industry, but there are some manifestations of OO in C++ that may

be error-prone and raise safety concerns. An important aspect that is not sufficiently covered in these guidelines is concurrency [Rog11]. Synchronous languages and SCCharts, with their built-in deterministic concurrency, provide an advantage in this regard.

Ada An OO language that is specifically designed for safety-critical embedded systems is Ada [ISO12]. Ada features a strong type system and a design-by-contract methodology. Its class-like packages support inheritance with overriding, multiple inheritance on interfaces, abstract types, subtyping, and polymorphism with dynamic dispatching. For the use in high integrity systems, Ada proposes ways of eliminating and mitigating common safety-related vulnerabilities that are associated with OO [Ada16]. Its approach utilizes Ada's capabilities in compile and runtime checking, contract-based programming, and formal verification. At the same time, the language remains methodologically neutral by keeping the OO features optional. If a program does not make use of OO, then these capabilities will remain inactive and do not impose any potential run-time penalty.

In contrast to Ada, OO in SCCharts focuses more on the aspect of MDE. Yet, its design likewise makes OO an optional feature and utilizes static restrictions to create a conservative subset.

Rust Started by the Mozilla Foundation, Rust[3] aims to reduce programming errors in development and shows great potential for the use in safety-critical systems [PCO19]. It is best known for its ownership model that guarantees memory-safety and thread-safety [JJK+17]. Mutability of references is always expressed explicitly and imposes a multiple readers single writer synchronization, similar to synchronous languages. In terms of OO, Rust provides object-based encapsulation of data and behavior but rejects classical inheritance because it is considered an unwanted overhead to always inherit all superclass behavior. Instead, Rust uses a *trait* system, similar to interfaces, that is implemented on a per-class basis and enables subtyping and polymorphism.

The scheduling regimes for objects, discussed in Section 5.4, are similar to the ownership model, since they regulate access to objects and thus

[3] `https://www.rust-lang.org/`

implicitly assign exclusive but also shared ownership. In terms of OO, SCCharts will follow the more common approach of providing classical inheritance, and if there are concerns about efficiency in a specific use case, the user can simply skip this feature.

5.1.2 Synchronous Languages

There are some synchronous languages that include a notion of objects either in their language or during code generation. However, to my knowledge none of them fully embraces an OO design including inheritance or incorporates visualizations tailored for object relations.

Blech The imperative synchronous language Blech [GG18], developed by Bosch Research, is inspired by Esterel and the SC MoC. In addition to classical synchronous programming constructs, it provides C-like abstract data structures, instantaneous functions, and reactive *activities* that can span multiple ticks. Listing 5.1a illustrates a small example with a `Counter` struct and an activity for automatic incrementing. In the initial proposal [GG18], activities and functions could be associated with a data struct in a class-like fashion. However, Blech never capitalized on this object-based design by introducing visibility restrictions, inheritance, or subtyping. Instead, the association was removed, and an optional module system now loosely bundles data types and related behavior.

In terms of scheduling, Blech relies on an interesting mixture of classical synchronous white-box causality and a black-box approach [GGM+20; GGM+22]. Blech permits sequential memory updates, but in the presence of concurrency, code sections are interleaved and variables follow the classical write-before-read protocol. However, activities are not partitioned or further analyzed, instead, their causality interface is directly expressed by their parameters, which are separated into a read-only and read-write list. Note that in line 4 of Listing 5.1a, the `Counter` struct is defined in the second parameter list because the activity writes the contained `value` variable.

In OO SCCharts, methods correspond to Blech's functions and regions to activities, but they are combined in a more feature-rich OO notation.

```
struct Counter
  var value: int32
end
activity Counting ()(c: Counter)
  repeat
    c.value = c.value + 1
    await true
  end
end
@[EntryPoint]
singleton activity Main ()()
  var c: Counter = { value = 0 }
  run Counting()(c)
end
```

(a) Struct and activity in Blech

```
data Counter with
  var int value;
end
code/await Counting (var& Counter c) ->
        NEVER do
  every 1ms do
    c.value = c.value + 1;
  end
end
// Main
var Counter c = val Counter(0);
spawn Counting(&c);
await FOREVER;
```

(b) Data structure and code block in Céu

Listing 5.1. Data structures in Blech and Céu, illustrating a counter with an additional routine that increments the value every tick respectively millisecond.

The concept of deterministic objects in Section 5.4 also utilizes a black-box perspective but provides a more flexible approach than the separation into writers and readers.

Céu Similar to Blech, Céu [SIL+17] supports structured data types. While inspired by objects, it also follows the same C-like approach and separates data and instantaneous or reactive procedures [San18; SIL15]. Listing 5.1b illustrates the previous Counter example in Céu, with the code/await block as a counterpart to Blech's activities. In contrast to Blech, the abstract data structures provide subtype relations with inheritance for fields, supplemented by a parameter-based dispatching mechanism for code blocks. Memory allocation is automatically managed based on lexical scopes and finalization handlers. To establish determinism, intra-instant communication of shared variables is prohibited in Céu, and concurrent code sections are scheduled non-interleaved in lexical (source code) order.

OO SCCharts aim for a more clear OO design, including the characteristic encapsulation of data and behavior. This, in combination with classical synchronous causality analyses based on data access, also enables a more fine-grained data-oriented mechanism for establishing deterministic be-

havior. Yet, in terms of dynamic instantiation and references, Céu is more advanced, which could also inspire improvements of OO SCCharts in the future, see Section 6.3.4.

Lustre The synchronous dataflow language Lustre [HCR+91] and its closely related siblings SCADE [CPP17] and Zélus [BP13] are not OO. Similar to Blech, they only provide support for simple data records. Yet, they recognize the useful structuring and encapsulation principles embodied by objects, and all utilize object-based intermediate languages for modular compilation [BCH+08; BBD+17; BCP+15].

The OO concept in this thesis makes this powerful structuring capability available to the programmer at the source level and further facilitates reusability and adjustability, e. g., by including inheritance.

In an extension to Lustre, Caspi et al. introduce *Scheduling Policies (SPs)* [CCG+09], which is based on an object notation. They bundle data and different modes of operation in the form of method-like dataflow nodes. SPs then describe the constraints for a sound access of these objects. While this concept inspired the modeling of SPs for deterministic objects in Section 5.4.3, it instead uses the SP formalism by Aguado et al. [AMP+18] that is compatible to the control-flow-oriented SC MoC and can act as a prescriptive contract for scheduling.

Synchronous Objects André et al. [ABP+97] introduce *synchronous objects* as an extension of the *reactive object model* [BDS96]. The basic idea is to conceptualize regular synchronous modules as objects that have a state, associated behavior, and a communication interface based on signals. This facilitates using OO design methodologies and models for specifying the different classes, their instantiation, and communication in a system. In their approach, interconnections between objects must be acyclic because communication is instantaneous and objects are considered black-boxes that cannot interleave with each other. Asynchronous objects from the host's domain are wrapped into *interface objects* to enter the synchronous messaging mechanism.

While André et al. support SyncCharts, Esterel, and Lustre for the specification of behavior in synchronous objects, they do not introduce OO to the languages itself, as it is the case with OO SCCharts. Instead,

regular synchronous modules are wrapped into objects and managed in way that is more closely related to an actor-oriented approach, such as LF (cf. Section 3.4.4).

synERJY The synchronous language *synERJY* [BPS04] founds its syntax on a subset of Java to facilitate an OO programming style. A program is separated into classes that can declare interface signals, local variables, and methods, as well as instantiate other synchronous classes. A constructor contains a synchronous subprogram that can be specified as a mixture of Esterel-like imperative code, Lustre-like dataflow equations, or a SyncCharts-inspired automaton notation. Objects run concurrently and only communicate by signals. While similar to synchronous objects by André, synERJY establishes a global white-box schedule for the entire program with a write-before-read ordering. If a causality cycle is detected, an explicit precedence specification for the involved operations (e. g., assignments or method calls) is requested from the user to resolve it.

There are many similarities between synERJY and SCCharts, also in regard to OO concepts. However, OO SCCharts more comprehensively include OO features, such as inheritance, and combine it with a model-driven approach. In terms of scheduling, synERJY's precedence system embodies the same principle envisioned by SDs and utilized for deterministic objects.

Lingua Franca With reactor inheritance, reaction overriding, type parameters, and methods, LF already supports OO designs. These features aim to improve reusability and adjustability of reactors and facilitate providing generic functionality in libraries.

While the notation and semantics are different in SCCharts, OO capabilities are introduced for the same reason. The concepts developed for OO SCCharts, e. g., visualizing object relations or statecharts inheritance, could also be transferred to LF in the future, see Section 6.2.1.

5.1.3 Statecharts

Statecharts are a popular formalism for specifying the behavior of embedded and reactive systems. There are various different dialects and variations of

statecharts. This also includes concepts for OO modeling. However, none of these OO variants features a deterministic synchronous semantics.

Objectcharts One of the first OO statecharts approaches was developed by Coleman et al., called *Objectcharts* [CHB92]. The basic idea is to model an entire software system in a top-down design process across multiple modeling notations. A *configuration diagram* represents inheritance, interfaces, and usage relations between classes, as well as their instantiation and the communication. Then, Objectcharts are used to characterize the lifecycle and state of classes. Yet, they do not explicitly describe the behavior of objects but the state-dependent acceptance and triggering of *services*, which are the abstract communication interface of these objects, similar to ports in LF. Objectcharts support inheritance to add new services and adjust transitions of the extended parent classes.

With their role as specification of communication admissibility, Objectcharts correspond more to the modeling of SPs in Section 5.4, than the OO extension proposed for SCCharts. In contrast to SCCharts, there is no provision for determinism in Objectcharts.

O-charts Following the idea of Objectcharts, Harel and Gery [HG96] present their own OO statecharts language. Again, two separate notations are used; UML-like *O-charts* to specify the class structure, inter-associations, and inheritance hierarchies; and statecharts to specify stateful behavior in classes. In contrast to Objectcharts, these statecharts are not limited to expressing the acceptance of messages but specify the actual implementation of operations. Therefore, C++ is directly integrated into statecharts as expression language. The proposed semantics features a run-to-completion concept with two different ways of communication; broadcasted *events* that are queued and then processed one at a time in subsequent reactions; and *operations*, corresponding to method calls, that immediately hand over control to the callee statechart and block until it returns. Consequently, there is no simultaneity in events. Moreover, concurrent reactions to an event may be non-deterministic. O-charts permit inheritance between classes and the corresponding statecharts can add new states, change transition targets, and introduce additional hierarchy or concurrency. Yet, Harel and Gery prohibit alterations that remove elements or change their containment hierarchy.

This is intended to prevent radical changes in behavior, in the sense of behavioral subtyping [LW94] or language refinement [LS17]. Yet, it is still possible to arbitrarily modify triggers and effects. Subsequent work by Syriani et al. proposes a more restricted set of rules to further ease static verification [SSL19].

The work by Harel and Gery is an important inspiration for OO SCCharts, however, there are some key differences. First, OO SCCharts do not introduce different notations for specifying a system but use a pragmatics-aware approach that utilizes views, e. g., to derive class diagrams similar to O-charts. Second, SCCharts use a synchronous semantics that provides a robust way to handle simultaneity and instantaneous communication. Third, the OO proposal uses a more coarse-grained overriding mechanism based on regions, which is closer to method-based overriding in major OO languages. By default, the proposed concept imposes no restrictions but can be extended to preserve behavioral consistency, e. g., via model checking, see Section 6.3.4.

Rhapsody Statecharts The work on O-charts resulted in the *Rhapsody* semantics of statecharts [HK04] and the commercial tool *Rational Rhapsody* by IBM, which implements O-charts in the form of UML class diagrams and statecharts with inheritance.[4] When adjusting derived statecharts in Rational Rhapsody, individual states, transitions, and other elements can be set to *inherited*, *overridden*, or *regular*. Inherited elements are fully in sync with their base definition and cannot be changed; overridden elements are adjustable but will also adapt to subsequent changes in their original definition, e. g., a modified state will be deleted if the original state is removed from the base statechart; and regular elements are completely decoupled from their base class definition.

These rather intricate relations stem from the fine-grained overriding mechanism but also seem to be influenced by the editing process that is imposed by a graphical-only modeling approach, as present in Rhapsody, O-charts, and Objectcharts. With a more classical overriding approach on regions and text-based modeling, OO SCCharts aim for a more natural and pragmatic programming experience.

[4] https://www.ibm.com/docs/en/rhapsody/8.2?topic=statecharts-statechart-inheritance

ROOMcharts In ROOM [SGW94] the system is composed of interconnected actors, and the behavior of each actor is described by a ROOMchart [Sel93]. Its notation mostly follows the classical statecharts formalism by Harel [Har87] and uses a run-to-completion semantics. Yet, it does not feature concurrent composition or broadcast communication because these aspects are only expressed on the actor level. ROOMcharts support inheritance similar to Harel's OO statecharts but do not impose restrictions on overriding states or transitions. Selic argues that such a language design "can be severely limiting in practical applications" [Sel93].

Again, OO SCCharts feature a different granularity for inheritance, and they embrace a synchronous semantic that includes concurrency and instantaneous communication inside SCCharts. Furthermore, the combination with actors in ROOM corresponds to dataflow SCCharts, discussed in Section 2.3.1, which also can benefit from the new OO features.

5.2 General Discussion and Assessment

The OO paradigm has established itself as a popular design principle in software development. There are numerous languages, design tools, and workflows based on this concept. Of course, there are also controversies regarding its success and underlying principles, but the general approach has proven to be successful [Ald13]. Cook once stated: "I believe the academic community as a whole has not adopted objects as warmly as they were received in industry." [Coo09] Clearly, OO is not the "holy grail" of programming paradigms. Programming languages evolve and ideally try to find approaches that fit the domain and the needs of developers best. This is also illustrated by OO consideration for the actor model [LLN09]. The OO paradigm is simply a very powerful and commonly used approach for structuring and designing software.

With this in mind, this thesis does not aim to enter into a wide-ranging discussion about OO programming. Instead, the goal is to enrich synchronous programming by OO facilities, as far as they fit. This especially since these principles have rarely been embraced by synchronous languages, see Section 5.1.2. Reasons might have been a focus on the C target, bare

metal embedded platforms, or potential risks in unconstrained dynamic behavior that might obstruct ensuring deterministic behavior. This section provides a brief overview on the core features of OO, while investigating and assessing possible implications and precautions in the context of synchronous languages. This facilitates creating a conservative OO extension for SCCharts.

The Notion of Objects Of the different definitions for OO that appeared over time, the one by Wegner [Weg87] has been the most accepted one, according to Capretz [Cap03]. It is also the one best suited for the concepts of this thesis. Wegner defines an *object* as "a set of *operations* and a *state* that remembers the effect of operations." [Weg87] Therefore, objects express data abstraction similar to abstract data types (ADTs); for a more detailed comparison see Cook [Coo09].

Wegner further distinguishes *object-oriented (OO)* and *object-based* languages. While object-based refers to the support of objects as a language feature, OO additionally requires a notion of classes and class hierarchies for objects.

object-oriented = objects + classes + inheritance [Weg87]

From this foundation originate three important characteristics of objects: encapsulation, inheritance, and polymorphism [Cra07; GM10].

Encapsulation Objects apply *encapsulation* or *data hiding* in which protected information (variables) are combined with legal operations (methods) to mutate the internal state of an object. To further decouple objects from each other, operation invocation is handled under the principle of *message passing*, where the implementation of the required operation is selected by the receiving object. *Classes* act as templates for constructing objects and unify objects of the same kind. They can have multiple *interfaces* that specify a set of abstract operations that the class has to implement.

Encapsulation is a well known principle in synchronous languages, expressed by modules in imperative styles and actors or nodes in dataflow notations. Compared to many general purpose languages, such as Java or C++, synchronous languages usually handle the invocation of operations

via signals and not methods. However, Blech and Céu illustrate that functions can generally be supported in synchronous languages as well, see Section 5.1.2.

Inheritance With *inheritance*, classes can be derived from one another. A subclass can reuse, alter, or extend the existing properties and behavior implementation of its superclass, to create a new kind of class adjusted to an extended purpose. Expressing common ancestry of classes facilitates *resource sharing* in the definition of objects. In this regard, inheritance is a flexible syntactic reuse mechanism.

In synchronous languages, reusing code comes in the form of modules or subprograms, e. g., activities in Blech. Traditionally, synchronous compilers perform a static macro expansion of modules, see Chapter 2. Yet, there are also modular compilation approaches for synchronous languages, for example in Blech [GG18] or based on interface theory for synchronous block diagrams [BCR12; TL18]. The incremental definition of classes via inheritance is a static feature and can be handled via macro expansion as well, as later illustrated in Section 5.3.2.

In the literature on OO, a point of criticism to inheritance is the *fragile base class problem* [MS98]. When a subclass is allowed to override method implementations that are also used internally by the base class, then changes and maintenance on the base class affect and will be affected by potential alterations in subclasses. Sabané et al. conducted a study on this problem and found no significant evidence that the presence of such internal overriding led to more fault-prone software [SGA+17]. Yet, in general, a more open and adjustable system requires a more careful design process.

Multiple Inheritance Conceptually, there is no restriction on the number of direct parent classes. *Multiple inheritance* describes the support for extending more than one class. However, multiple inheritance can lead to the *diamond problem* [Cra07; GM10], in which a class inherits the same function from two different superclasses with different implementation. Hence, the challenge when supporting multiple inheritance is to resolve this *ambiguity*. Some languages simply prohibit inheritance of behavior from more than one superclass. Python resolves in lexical order of listed superclasses. Java

with its default implementations in interfaces requires a reimplementation of ambiguous methods to unify their behavior in the deriving interface. In C++, method calls with multiple competing definition must explicitly name the class that should provide the behavior. Lee et al. created an ultrametric space for multiple inheritance in actors that resolves the behavior via distance in the model [LLN09].

While there are sound technical solutions to support multiple inheritance, it may complicate code understanding and maintenance, and is often advised against, for example in industry coding guidelines, see Section 5.1.1. Hence, it should be considered carefully.

Delegation It is important to note that even if inheritance is a core concept of OO, it is not the only way to achieve reusability. *Composition* is a viable alternative, for example in the form of the *delegation pattern* [GHJ+95].

Gamma et al. present a point of view that favors composition over inheritance [GHJ+95]. They argue that inheritance is essentially a white-box approach that gives access to the internal state of an object, while composition is able to achieve the same result while retaining full (black-box) encapsulation. Rust is an example for a language that follows the principle of Gamma et al.

This became a controversial topic in the OO community and requires a closer look, since it might be interpreted as inheritance being superfluous or adverse. Tempero et al. investigate the use of inheritance in large Java projects [TYN13]. Their studies conclude that "there is no need for concern regarding abuse of inheritance." They state that the observed use of inheritance is justified, especially when it comes to subtyping and adjusting internal behavior. Kegel and Steimann analyze the practicality of refactoring inheritance into delegation in Java projects [KS08]. Their results show that this "is neither always possible, nor generally trivial to perform" and "the refactored code exposes so much technical overhead that the benefit of the refactoring must be questioned."

Modules already provide capabilities for compositional design in synchronous languages, and the proposed concept for OO SCCharts will not obstruct a delegation pattern.

Prototype-Based Inheritance Wegner's object notation uses a class-based approach. However, there are also languages that apply *prototype-based* inheritance, such as JavaScript. In these languages, there are no real class definitions.[5] Instead, objects act as prototypes for each other, when they are created at runtime. New objects can be manipulated to replace or add members to the object. Such a classless design is often used in interpreted languages without static typing.

In synchronous languages, strong static compile-time assurances for determinism and correctness are a key feature. This is a fundamental opposite to prototype-based inheritance that essentially shifts the object definition to runtime.

Type Inheritance In the presence of a type system, inheritance often also expresses *subtype* relations, where an object is allowed to substitute objects of its superclasses or interfaces. However, these are conceptually separate concepts [CHC89]. The main difference between inheritance and subtyping is that inheritance establishes a syntactical relation for reusing code between classes, while subtyping ensures the compatibility and substitutability in a type system. Subtyping or *type inheritance* is a form of *polymorphism*. In practice, most OO languages use the same syntax to express these relations, which results in subtype acceptance potentially restricting reusability via inheritance. Yet, there are also ways to uncouple these concepts [CHC89].

Forms of Polymorphism In general, polymorphism is the applicability of operations, functions, and variables to more than one type. Cardelli and Wegner differentiate four forms of polymorphism [CW85], presented in Figure 5.1. Since this is not a topic specific to OO but of type theory and programming languages in general, they build upon exiting work and extend it by inclusion polymorphism to capture subtyping in OO. Inclusion belongs to the category of universal polymorphism that describes the handling of an infinite number of types. In the same category is parametric polymorphism, for example embodied by generics in Java [GJS+15] (e. g., in Listing 5.2). Ad-hoc polymorphism is restricted to a finite number of types and covers method overloading and operation-dependent type casts (coercion).

[5] In ECMAScript 6 there are optional class definitions, but they only act as syntactic sugar for JavaScript's existing prototype-based inheritance.

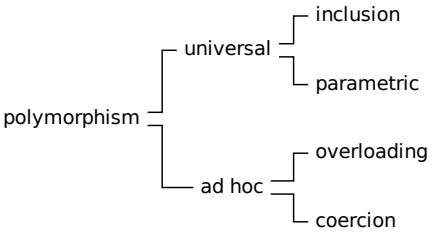


Figure 5.1. Varieties of polymorphism (based on [CW85])

```
1 interface<E> Stack {
2
3   public E pop();
4   public void push(E item);
5 }
```

Listing 5.2. A simple generic interface for a stack in Java

Polymorphism permits the presence and use of different type-dependent implementation that is selected based on the syntactic or runtime context. Hence, it must be carefully considered in the context of synchronous languages, since it can hamper or jeopardize establishing deterministic behavior when the actual implementation and its causal relations are unknown.

Inclusion Polymorphism With inclusion polymorphism an object can have multiple types. Assuming the coupling of type and implementation inheritance, it inherits the types of all superclasses. As such, it is applicable to any context expecting a supertype. However, as Cook et al. describe, subtyping bound to inheritance without any restrictions can lead to an insecure type system [CHC89]. To ensure that a subtype object can work in the context of a supertype, many languages implement additional constraints for the validity of a subtype relation, most commonly the availability of operations with the correct type[6] or a model refinement relation [LS17]. However, there is also the concept of *behavioral subtyping* or the Liskov substitution principle [LW94] that additionally requires a semantically permissible substitution of the behavior.

Consider the Java interface in Listing 5.2. Any Java class that implements this interface is a valid subtype of `Stack` because it will provide the `push` and `pop` methods. However, there are no formal or programmatic assurances that `push` and `pop` actually provide a first-in-last-out semantics as intended

[6] For example, a subtype usually must not change the return type of a method, e. g., `bool getValue()`, if the base type specifies `float getValue()`.

by a stack. Behavioral subtyping would use model checking, verification, and formal methods to provide such an assurance.

Behavioral subtyping is employed in the context of Harel's O-charts or as *local type consistency verification*[7] in Ada [Ada16]. Lee and Xiong present a behavioral type system for concurrent component-based designs [LX04] that uses an extended form of interface automata [AH01].

Since synchronous languages lend themselves well to model checking and static analyses, such techniques can be applied in this context as well, in order to restrict subtyping to a desired degree.

Operation Dispatching If objects can substitute objects of a supertype and provide different behavior, this requires a type-depend selection mechanism to execute the correct behavior. In the OO concept of message passing, this is considered *dispatching*.

If it is possible to deduce a single implementation at compile-time, e. g., by using constant object references or via type inference, *static dispatch* directly resolves the invocation. Otherwise, *dynamic dispatch* is required to select the correct implementation at runtime. Classically, this is implemented in a virtual function table, but there are also alternative approaches, such as explicit type switches with rigorous type inference, as performed by Zendra et al. [ZCC97].

Synchronous languages usually perform static data analysis at compile-time to ensure determinism. This naturally favors static dispatch to resolve polymorphism. There are many powerful static analyses to infer and determine types [CW85; PS91; CCZ97], which are profoundly applied in languages such as Rust, Ada, and functional OO languages. Dynamic dispatch has an implicit runtime overhead and makes it more difficult to establish determinism based on data access if the access can only be determined at runtime. This makes it less applicable to synchronous languages.

However, the need for dynamic dispatch stems from the presence of unconstrained runtime-mutable object references. In synchronous languages, pointers are rarely supported as they often impede a static causality analysis.

[7] Local type consistency verification is a strategy to mitigate subtype vulnerability and is specified in DO-332 [DO-11], which is an OO supplement to DO-178C [DO-12], a standard for software safety in avionics.

Blech supports mutable and immutable references and uses an ownership model inspired by Rust to ensure a single writer, but it is not subject to subtyping in the first place. Céu has mutable pointers and subtyping but uses a lexical order for deterministically scheduling threads and, hence, does not need to consider the effect of references on a data-oriented causality analysis.

Parametric Polymorphism The second form of universal polymorphism enables functions or classes to carry *type parameters* that act as a placeholder to define functionality independent of the actual argument type. Listing 5.2 illustrates an interface for a stack in Java that has a generic type parameter `E`. This allows to predefine the `push` and `pop` methods to this arbitrary but consistent type.

In statically typed languages, such as Java and Ada, type parameters are resolved statically at compile-time. This procedure is very similar to macro expansion, which makes this feature easily applicable to synchronous languages.

Ad-Hoc Polymorphism *Overloading* permits the same identifier to refer to different functions or operations based on the involved types. A class can have multiple methods with the same name but different parameter types and lists. Another example is the plus operator, which in many programming languages performs a concatenation, when applied to strings, instead of a numeric addition. Resolving the actual implementation based on the type is again subject to static or dynamic dispatching, hence, the same considerations for synchronous languages apply here.

Coercion describes automatic type conversion. For example, in the case of a string concatenation with plus, the second operand, if not already a string, is usually automatically converted into one. In synchronous languages, this feature is usually handled by the type system or the host language.

Final Remarks The OO paradigm combines different principles and mechanism into a powerful design and programming concept. As it turns out, many of the features are of a static nature and can be adapted to a synchronous context that relies on a static causality analysis to establish determinism. Similarly in LF, all OO features are currently supported only

for static use. Yet, with growing support for mutations, there is also the questions of admissibility of subtype reactors that are dynamically inserted into a reactor.

In general, dynamic components with polymorphism enable designs with a high level of interoperability and flexibility [Ald13]. However, they can impede static analyses and may jeopardize determinism. Hence, a conservative approach has to restrict these features to ensure the integrity of its semantics, e. g., as illustrated by Ada. Even without dynamic aspects, OO has a lot to offer. Creating reactive classes, reusing code via inheritance, or adjusting behavior via overriding can be valuable modeling capabilities for synchronous languages when dealing with large and complex systems.

The OO proposal in this thesis will follow a conservative path and restricts itself to features that can be handled statically, see summary in Section 5.5. As it turns out, this already covers a significant portion of aspects relevant to the language design of SCCharts. Once the language features are present, semantic restrictions can be lifted in the future, for example by advancing the static analysis, revisiting scheduling granularity, or introducing black-box approaches, as Section 6.3.4 will discuss.

5.3 Object Orientation in SCCharts

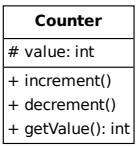
This section presents a conservative extension of SCCharts by OO. While there are OO statecharts notations and some objects-based synchronous languages, this is the first OO synchronous statecharts dialect that combines both under a pragmatics-aware approach. The research question at hand is: How can we introduce OO modeling features into a statecharts notation, while remaining on the safe semantical terrain of the synchronous MoC? This includes utilizing the benefits of textual modeling in combination with graphical views to create a natural OO programming experience. Furthermore, it involves the consideration of classes as an OO refinement of synchronous modules, methods that represent a classical OO programming notion and benefit from the imperative sequential nature of the SC MoC, inheritance not only in terms of code reusability but also for adjustability of behavior, and safe aspects of polymorphism.

Additionally, this section describes a *high-level implementation* approach that relies on the static macro expansion principle and treats these new OO features as syntactic sugar. The transformation results can be inspected and verified on source level, thus grounding their semantics in the existing sound execution model of SCCharts. This also preserves the current mechanism of a global white-box causality analysis to establish determinism under the SC MoC. The goal is to build upon the well-established and tested compilation and code generation mechanisms of SCCharts, without imposing new requirements or significant overheads. A further extension into a modular OO code generation is considered future work, see Section 6.3.5.

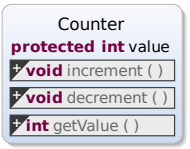
5.3.1 Class Modeling

Traditionally, SCCharts model an entire system as hierarchically and concurrently composed statecharts. Modules facilitate this process by dividing the system into subprograms, modelled as individual SCCharts, as illustrated by the Furuta pendulum example in Section 2.4. However, in line with the new OO modeling perspective, SCCharts modules can also be interpreted as classes. An SCChart has a name, can contain local variables and provides behavior usually associated with its regions. The OO extension of SCCharts introduces new capabilities to utilize SCCharts as classes, hereafter referred to as *SCCharts-based classes*. This aims at modeling user-defined OO data structures in SCCharts, as already illustrated by the `TimedSignal` in Section 4.5.3. Indirectly, it also enables a more imperative programming style for specifying behavior in classical SCCharts modeling.

Methods Regions represent the inner behavior of states that execute when the state is active. With the OO extension, SCCharts can now specify *methods* alongside regions. Their behavior is only executed upon invocation and is restricted to be instantaneous, i. e., methods must not contain any synchronous delay (pause). This limitation is a design decision to establish a clear separation between the implicitly active and stateful behavior in regions and instantaneous effects to method invocations. It is in line with other synchronous languages, such as Blech, which separate functions and activities, or Céu with `code/tight` and `code/await`. LF also provides instantaneous methods.



(a) UML notation



(b) SCCharts notation

Figure 5.2. Visual representations of a Counter class.

```
1 scchart Counter {
2   protected int value
3
4   method increment() { value++ }
5   method decrement() { value-- }
6   method int getValue() { return value }
7 }
```

Listing 5.3. Textual SCCharts-based Counter class.

With methods, SCCharts grow closer to a classical class notation, as illustrated by Figure 5.2. Figure 5.2a shows the UML class diagram [Obj11] of a Counter class. This model of a counter will act as a running example for various aspects presented in this chapter. It consists of a protected[8] integer field value and three methods: one for incrementing the counter, one for decrementing it, and a getter method to return the current value. The same information is also available in the SCChart in Figure 5.2b. Additionally, the given SCChart includes implementations for the three methods, reflected in the source code in Listing 5.3.

In contrast to regions, methods do not contain a state machine but instantaneous imperative code. Their bodies are written in a subset of the Sequentially Constructive Language (SCL) [HDM+14], which is a minimal imperative language fully equivalent to the SCG described in Section 2.3. Aside from their classical role in object design, methods improve the expressiveness of SCCharts beyond the usual trigger-effect-pattern in transitions and provide reusable code snippets to perform classical algorithmic computations.[9]

[8] Figure 5.7a requires this visibility to enable the later implementation of a reset method in the deriving this class.

[9] Methods have already proven themselfs useful in a student project involving SCCharts. The students wanted to find the shortest path and hence implement Dijkstra's algorithm. They modelled a state machine with only transient states and immediate transitions. Clearly, this was not a fitting notation for such an algorithm. An alternative would have been factoring out this functionality into the host language. However, since the SC MoC already provides a foundation for such sequential behavior, methods closed the notational gap and offered the students imperative programming capabilities within SCCharts to naturally implement the algorithm.

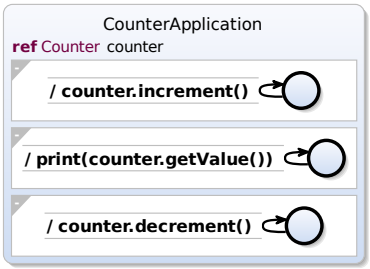


Figure 5.3. The CounterApplication SCChart using Counter class. (Publ. in [SSM21])

In Figure 5.2b, the graphical representation of methods uses a region-like notation. They are displayed in gray, to distinguish them visually from regions and illustrate their inherent inactivity in the absence of an invocation. If expanded they reveal a graphical SCG representation of the SCL code specified in the textual source, the example in Figure 5.4a, will later illustrate such a view. In fact, this representation is actually only a CFG because the subset of SCL that is permitted in methods excludes the synchronous pause and concurrency, as these aspects should be expressed in classical regions. Since such a diagram represents only one possible view, a user can also switch to a declaration-like notation with a textual preview of the method's body or just the method's signature. Some subsequently presented models will use this more compact representation, e. g., Figure 5.5.

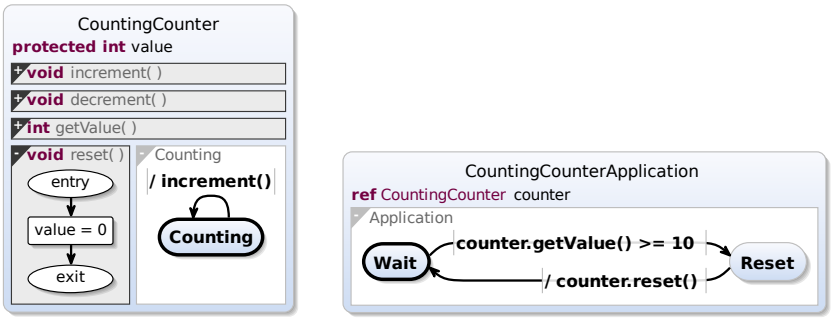
Instantiation Now that SCCharts can represent a class, such as Counter, they can be used for instantiating an object. The SCChart in Figure 5.3 declares an instance of Counter as the counter variable. The three regions invoke the methods counter.increment(), counter.decrement() and counter.getValue() concurrently in every tick. The SC semantics and its IURP prescribes a scheduling order where the printing (reading) of the value comes after the other concurrent method invocations (updates). As a consequence, the counter value is always zero at the end of the tick. This may not appear useful, but the behavior of this example will be used in Section 5.4 to illustrate custom scheduling on objects and has no further implications on the example in this context.

The notation **ref** Counter counter corresponds to the instantiation of SCCharts as actors in dataflow regions [Smy21]. The classical notation for referenced SCCharts [SMS+15] (**state** S **is** Module(), e. g., in line 12 of Listing 2.3) is unsuitable for instantiating object, since it is designed to embed the referenced behavior without providing a handle to the object. Nonetheless, such a notation will reappear in the context of inheritance in Section 5.3.2, where it is used instantiate a state by anonymously extending a class. This, in turn, permits implicit access to its inherited members.

In SCCharts, the declaration of a variable with SCCharts type is considered an instantiation. Each such variable is an immutable reference to an automatically created instance. This takes a conservative approach to the problems imposed by mutable pointer, dynamic instantiation, and runtime polymorphism, as discussed in Section 5.2. Yet, future work can build upon this notation and lift these restriction when there is a more advanced analysis for mutable references, see Section 6.3.4, or per-object assurances for determinism, as presented in Section 5.4.

Bindings The instantiation of objects upon declaration establishes a composition relation between the object and its instantiating SCCharts. However, objects might need to interact with each other without such a containment. In classical programming this is solved by passing object references. Yet, this conservative proposal refrains from explicitly introducing pointers, instead it relies again on the principles of macro expansion. In referenced SCCharts modules, inputs and outputs of modules act as temporary aliases that are lexically replaced upon expansion. This is also supported for SCCharts-based classes. Conceptually, binding inputs and outputs resembles passing references in a constructor, and it also appears similarly in the code, see Listing 2.3. Hence, an input ref declaration can be used to refer to an object instantiated outside the scope of the SCChart. Section 5.3.3 will present how this can be combined with subtyping and Section 4.2.3 already illustrated the practical application of this mechanism with the TimeUtil class.

Mixing Methods and Regions While methods offer a way to modularize reusable sequential instantaneous behavior, regions retain their previous role of specifying stateful behavior, now directly associated with an object. Figure 5.4 illustrates an extended version of the Counter class in Figure 5.2b.



(a) The CountingCounter SCCharts class.

(b) Use of CountingCounter class in CountingCounterApplication.

Figure 5.4. Example of an SCChart class with a region.

Each object of this class will autonomously increment its value in every tick. This behavior is defined by the region Counting, containing a single state with a self-transition that invokes increment.[10] Section 5.3.2 will illustrate how to use inheritance to base the definition of CountingCounter on Counter, instead of redefining it.

Figure 5.4b illustrates an SCChart that creates a variable counter based on the CountingCounter. The program simply waits until 10 ticks have passed and then resets the counter value in the next tick and starts again. Manually invoking increment or decrement would slow down or speed up this progress.

High-Level Transformation SCCharts-based classes can be considered an extended feature and be replaced by more basic language elements. During compilation, ref declarations are transformed into a more low-level class representation, whose syntax is strongly inspired by classes in Java. This *native class* type is also available to users and enables a more classical approach for defining classes. However, its main purpose it to provide an integration of objects and classes from host languages into SCCharts, as described and utilized in Section 5.4. In this case, it is used to ease the down-

[10] A during action would be a more compact notation, but the purpose of the example was to explicitly introduce a region.

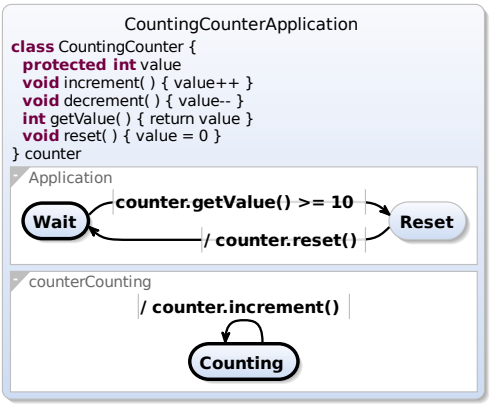


Figure 5.5. First step in the high-level transformation of the CountingCounterApplication, replacing the SCCharts-based class by a native notation.

stream compilation of SCCharts-based classes. Figure 5.5 shows the result of this translation step for the previous example of the CountingCounterApplication from Figure 5.4b. The class declaration of CountingCounter contains the counter variable and the three implemented methods. Since native class declarations do not support regions, this transformation integrates them into the instantiating SCChart. The region is renamed to counterCounting to enable multiple instances of such a class. The method call counter.increment() was adjusted to refer to the instance variable, since the region is no longer in the same scope as the increment method. Instantiating regions of SCCharts-based classes inside the declaring SCChart is only possible due to the static instantiation and the read-only restriction of the ref declaration. In turn, this binds the lifetime of an object to its declaring state.

In the next step of the transformation, method invocations are statically expanded and replaced by the body of the method, known as procedure inlining [ASU07]. Parameters are directly resolved to their invocation arguments by a constant/copy propagation. Figure 5.6 shows the result of method inlining for the CountingCounterApplication. Note that this model is manually created to illustrate this procedure because in the actual implementation, the method inlining is performed in a later transformation

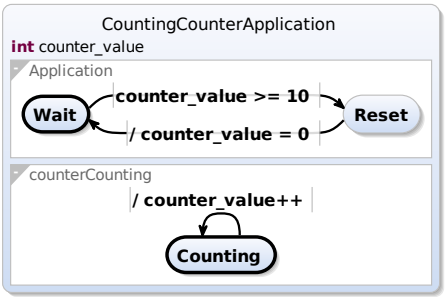
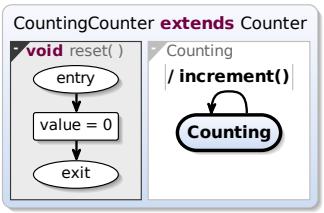


Figure 5.6. Conceptualized second step in the high-level transformation of the CountingCounterApplication with method inlining and simplification of the class data structure.

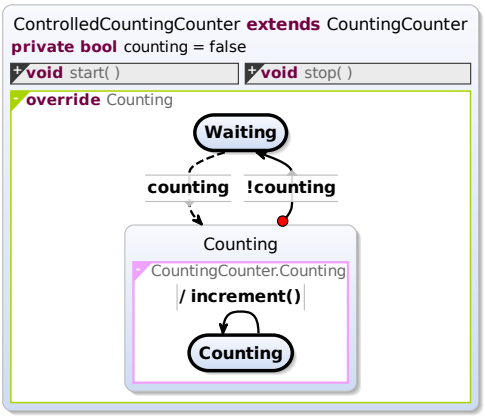
step at the SCG-level. There is no conceptual difference to when the inlining is performed. For the SCCharts compiler, only technical considerations resulted in implementing this functionality at a lower level.

In addition to the method inlining, the Figure 5.6 also illustrates the result of "unpacking" the class Counter. With static instantiation, read-only references, and inlined methods, the remaining members in a class's data structure can be extracted and turned into individual variables, in this example counter_value. With such a conversion, the last trace of the notion of objects disappears. This illustrates that with the high-level approach used here, the OO features introduced so far can be treated as syntactic sugar. Moreover, it fully grounds the proposed language elements in the existing definition of the SC semantics and SCCharts.

Class Transformation in Practice While Figure 5.6 illustrates how classes can be transformed into classical SCCharts without any notion of OO, the SCCharts compiler will not always perform this second step. By default, it will only convert the SCCharts-based classes into the native class notation, as in Figure 5.5, and maintain this structure even in the generated code. The reason is that every relevant host language can express structured data, including C and VHDL. To enable this approach, the dependency analysis for SCCharts was extended to handle such data structures.



(a) The CountingCounter SCChart extending the Counter in Figure 5.2b.



(b) The ControlledCountingCounter SCChart extending the CountingCounter and overriding the Counting region.

Figure 5.7. Examples of inheritance and overriding in SCCharts-based classes.

In the same sense, the inlining of methods can be disabled. Yet, if method invocations are kept, the compiler will treat them as atomic units, which limits the potential for interleaving and may result in a rejection of a program as not SC schedulable, see Section 6.3.5. The current SCCharts compiler is able to pass on classes into the generated code as far as it fits. I. e., in C, classes are split up into structs and functions with access to that struct. This approach maintains a better association between the generated code and the original model, and establishes a foundation for an OO code generation in the future, see Section 6.3.5.

5.3.2 Inheritance

Inheritance is an important OO concept that expresses commonalities between classes. The CountingCounter in Figure 5.4 is an example where basing the class upon the Counter in Figure 5.2b can eliminate the need to copy the common implementation. Figure 5.7a illustrates a variant of this SCChart

that uses the new extends keyword to declare a superclass and only defines the reset method and Counting region, while inheriting the variable and methods from the Counter SCChart. Ultimately, this class has the same properties as the one in Figure 5.4.

OO SCCharts support the common visibility modifiers private, protected, and public on variables, methods, and regions. This enables the reset method to access the value variable directly, while in the CountingCounterApplication the access to this internal variable is restricted.

Overriding Inheritance unfolds its full potential when overriding enables the adjustment of inherited behavior to the purpose of the extending object. Overriding in classical imperative OO programming languages, such as C++ or Java, is a rather straight forward process because it applies only to methods. An overriding method completely supersedes the overridden implementation. However, there are more options if the behavior is modeled in a statecharts notation, as Harel's OO statecharts illustrate. Harel's inheritance features fine-grained altering of states, transitions, triggers, and effects, since these represent the implementation of different methods. In contrast to that, inheritance in SCCharts supports top-level *region overriding* and classical *method overriding*. This unifies the overriding mechanism for both methods and regions. It further facilitates future development on OO code generation that might similarly consider a region an atomic scheduling unit, see Section 6.3.5.

Figure 5.7b illustrates overriding in the context of the counter example. The ControlledCountingCounter extends the previously discussed CountingCounter, adds two new methods, start and stop, that set and unset the additional boolean flag counting and overrides the existing Counting region. The green highlighting indicates an overriding region. The previous behavior is replaced by two states, Waiting and Counting. Only if counting is true, the SCCharts will enter the Counting state and will perform automatic increment of the counter value. As soon as counting becomes false, this behavior is aborted. Hence, a user now has control over the counting process. Furthermore, OO SCCharts support accessing the implementation of superclasses while overriding, similar to languages such as Java [GJS+15]. In this example, this mechanism is used to instantiate the previous (super)

implementation of the Counting region inside the new Counting state. The diagram indicates this by referring to it as CountingCounter.Counter in the region title and highlighting the region in the purple, in accordance to the existing referenced SCCharts notation. If more fine-grained overriding is desired in the future, this mechanism could be extended to pick individual elements from the overridden implementation.

Multiple Inheritance OO SCCharts support multiple inheritance. As discussed in Section 5.2, this must include handling ambiguities. SCCharts uses the same strategy as for default methods in Java 8 interfaces [GJS+15]. If multiple superclasses contribute regions or methods with the same identifier but different implementations, they must be overridden to define a single unambiguous definition. Otherwise, the model is rejected by the compiler. Likewise, SCCharts are rejected if the inheritance hierarchy is cyclic, name clashes occur, or an ambiguous super scope is accessed.

As indicated by Listing 5.3, there is no syntactical distinction between SCCharts-based classes and interfaces. This is a contrast to other OO languages, such as Java. The underlying design decision was to subordinate the SCCharts-based class design to classical SCCharts modeling. Hence, OO SCCharts feature a set of annotations: @Interface, @AbstractClass, and @Class. These annotations activate validation rules that issue an error if the SCChart does not comply with these OO design concepts (based on the definition in Java [GJS+15]) and enforce their consistent use across class hierarchies. They act as a design guideline and, hence, the @Class will discourage multiple inheritance of classes (providing behavior), as discussed in Section 5.2. If multiple inheritance is desired, it must be explicitly enabled by the annotation, i. e., @Class[MultipleInheritance] true.

Anonymous Classes in States The example in Figure 5.7a illustrates the use of inheritance for SCCharts-based classes, and Section 5.3.1 discussed that the classical module expansion syntax for states is insufficient for instantiating objects. However, with inheritance and overriding, there is an opportunity to revive and advance this concept. The **state** S **is** Module() pattern (e. g., in line 12 of Listing 2.3) creates a state and fills it body with the contends of the Module. The **state** S **extends** Module() {<body>} syntax will have the same effect. However, it enables adding behavior in its body with

access to the superclass definition, and more importantly override existing methods or regions to adjust the implementation for this specific state. This corresponds to the concept of *anonymous classes*, e. g., in Java [GJS+15]. Hence, in addition to its role in SCCharts-based class modeling, inheritance can act as an advanced macro expansion and adjustment mechanism for states in the classical design of SCCharts.

The Logger Example Figure 5.8 illustrates an example that uses inheritance on individual states to create a modular but easily adjustable design. The presented SCChart is a simplified model of a real-world example.[11] In the underlying scenario, incoming messages, here reduced to messageA and messageB, must be processed differently depending on the state of the application. By default, a received message is logged. This common behavior is modeled in the DefaultLogger that has separate concurrent regions for processing each message. The state machine in each region immediately logs the message's content on an info level, if received. Then, it switches to the Logged state and returns to the receiving state in the next instant to process further messages. The input messages are declared as valued signals in the MessageReceiver SCChart, extended by the DefaultLogger. Valued signals provide a combination of presence indication and payload (accessed via val), similar to an events in LF. In the actual application represented by LoggingApplication, the behavior differs from the default logging behavior depending on the state. In this simplified example there are two states in the application, ACausesError and BCausesError, that alternate triggered by the next input. Each state inherits the behavior of the DefaultLogger. In state ACausesError the handling of messageA is altered by overriding region HandleA. If messageA is received, an error is logged and the Error state is entered but not left, ignoring future occurrences of messageA in this state until reentry. Analogously, the state BCausesError handles error from messageB.

[11] An industrial partner from the railway domain uses SCCharts to replace handwritten state machine code by models and generated code. In the context of a C++ project, the developers found the need for states to have common default behaviors. An example is the described logging of messages. It is only reasonable to address such a use case by means of OO, especially since C++ developers are already used to this methodology.

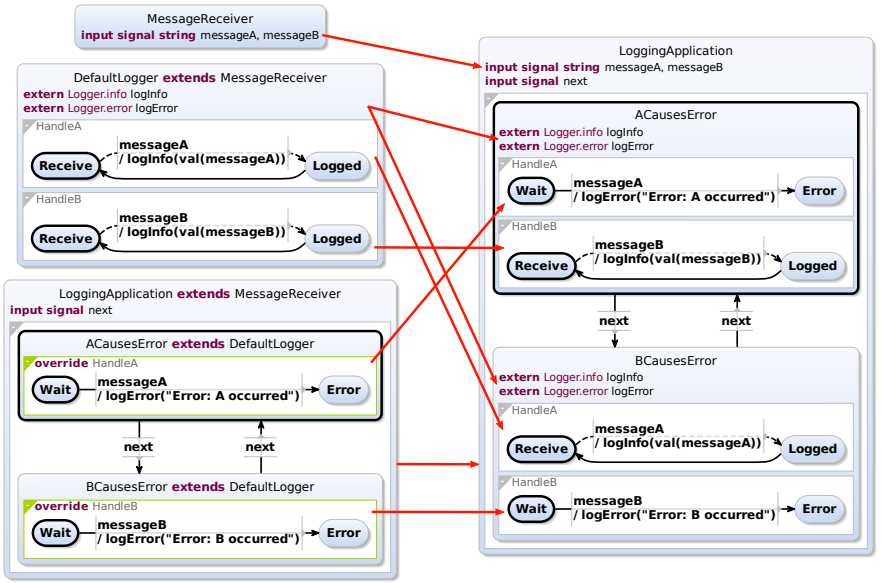


Figure 5.8. Example for usage of inheritance and overriding (left) and the result after inheritance is statically expanded by the compiler (right). Red arrows indicate where the parts of the model are expanded into. (Publ. in [SSM19; SSM21] ©2019 IEEE)

High-Level Transformation Inheritance in SCCharts can be treated as an extended feature and removed by a macro expansion mechanism. A model-to-model transformation copies all variables, methods, and regions into their extending states. Overriding is handled by static dispatching of overridden behavior. The right-hand side of Figure 5.8 shows the result for the LoggingApplication. The red arrows indicate the relation between the use of inheritance with overriding and the resulting model. A special case are the input signals in MessageReceiver that need to be bound when extended in ACausesError and BCausesError, because only root states can have an input output interface. The syntax enables an explicit binding, but in this case it is optional because the LoggingApplication shares the same common ancestor interface (MessageReceiver) and the signals are implicitly bound.

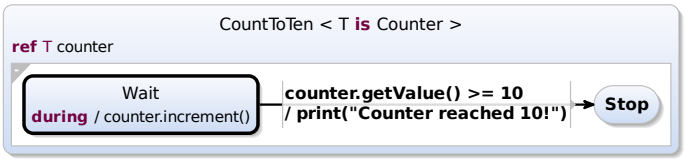


Figure 5.9. An SCChart with a type parameter for a Counter classes that is instantiated and used to count to ten.

5.3.3 Type Parametrization and Subtyping

Parametric polymorphism is the foundation of generic programming and enables the definition of abstract behavior applicable to multiple types only concretized upon actual use. While this concept is not specific or limited to OO, it can be combined with subtyping. C++ implements type parametrization in templates [ISO20], while Java uses generics [GJS+15]. Similarly, SCCharts-based classes can declare type parameters.

Figure 5.9 illustrates the generic CountToTen class that declares a type parameter T after its name. The type T is then used to declare and instantiate the variable counter. In order to enable any sensible interaction with the object, the type T is restricted to the Counter type. Hence, a concrete argument must be of type Counter or any valid subtype. With this constraint, the Wait state can invoke the increment method in a during action and transition to Stop when the value reaches at least 10.

Now, this class could be instantiated via **ref** CountToTen<Counter> ctt, which will result in the message printed after ten ticks. However, if the CountingCounter from Figure 5.4 is passed as type argument for T, the message already appears after half the ticks because this class additionally increments the value by itself.

Subtyping The CountToTen example illustrates that SCCharts-based classes can be used to substitute each other if they are in a subtype relation. The same applies if an SCChart declares a ref declaration as input. As discussed in Section 5.2, it is relevant to restrict the acceptance of subtypes beyond the inheritance relations. For subtyping in SCCharts, there are three aspects that can be considered:

1. interface compatibility,
2. scheduling admissibilit, and
3. behavioral conformance.

Subtypes must provide a compatible interface in terms of operations and their types (1). For example, this is expressed in model refinement [LS17] and OO type checking [Car88; WNS+06; KN06]. In Figure 5.9 inheritance ensures the availability of the used methods, while removing methods is not permitted. Validation rules for overriding will reject changes to incompatible types, e. g., in the return type of a method.

Determinism is a crucial aspect to synchronous languages. The conservative approach presented in this section relies on the established concept of a global white-box causality analysis. In this context, subtype objects can be analyzed and scheduled individually. However, in future extensions, a subtype restriction in terms of scheduling admissibility (2) could enable objects that are interchangeable at runtime. The basic idea is that a subtype must provide the same scheduling interface as its supertype. The program can then establish a deterministic schedule based on the supertype and retain this determinism for any subtype, as Section 6.3.4 will further discuss.

The next level of subtype restrictions is behavioral subtyping (3). There are model checking capabilities in SCCharts [Sta19] that could be extended in this regard. For example, if `Counter` carries some form of invariant that prevents a change of the value without external interaction and must be fulfilled by subtypes, this could be used to prevent the `CountingCounter` from becoming a valid subtype of `Counter`. Given the scope of this thesis, the integration of behavioral subtyping remains future work, see Section 6.3.4.

High-Level Transformation Generic type parameters in SCCharts are statically expanded at compile time. A subtype check validates whether the given argument is an admissible type and then replaces all occurrences of the parameter with the given argument. For the example in Figure 5.9, the static instantiation of **ref** `CountToTen<Counter>` `ctt` follows the procedure described in Section 5.3.1 but expands **ref** `T counter` into **ref** `Counter counter`. Then, the expansion continues recursively.

5.3.4 Modelling Object-Oriented SCCharts

The proposed OO SCCharts combine statecharts modeling with established concepts of OO. However, in order to create a language that provides OO design in a practical and effective way, the actual programming support, or in this case the modeling support, is an important aspect. Stroustrup, the creator of C++, formulates it this way:

> "Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment." [Str87]

In this sense, the OO features presented so far represent language support for user-defined types and expressing inheritance. This is accompanied by a pragmatics-aware modeling environment in the form of the KIELER tool, see Section 2.5. It offers a unique way of combining textual editing with customizable transient views. This also affected the design of OO features in SCCharts, since they are under deliberate influence of popular and especially textual imperative OO languages.

KIELER Figure 5.10 shows a screenshot of the ControlledCountingCounter model from Figure 5.7b in the KIELER tool. On the left side is the editor with a textual source code. The extends syntax in line 5, the overriding of regions as a method-like unit in line 15, and the reference to the super implementation in line 21 are designed to create a Java-like programming experience. This is accompanied by common editor features, such as content-assist.

Located on the right is the Diagram view that displays the graphical representation of the SCChart. The sidebar left of the diagram shows some configuration options for SCCharts and its OO features. In contrast to the diagram shown in Figure 5.7b, this view is configured to provide a preview of inherited elements. Hence, the declarations at the top of the SCCharts also list the variable and methods defined in the derived SCCharts. This

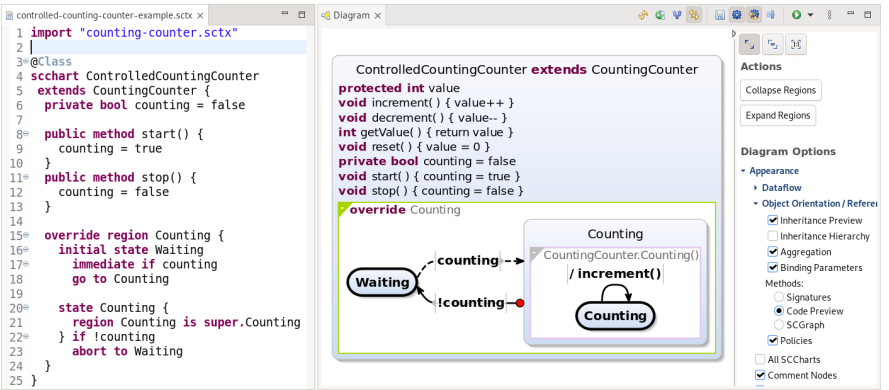


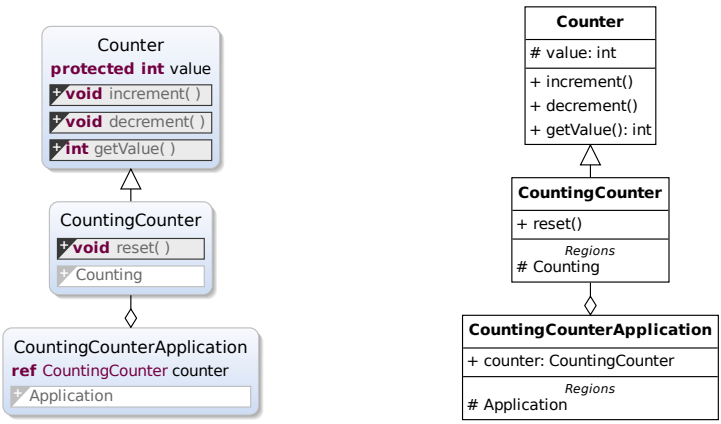
Figure 5.10. Screenshot of the ControlledCountingCounter in the KIELER tool.

gives the modeler an impression of the final properties in the designed class. Additionally, methods are configured to appear as textual code snippets, rather than graphical SCGs.

UML Class Diagrams Another new visualization option that comes with the OO extension is the arrangement of SCCharts in a UML class diagram notation. Figure 5.11a illustrates such a view for the CountingCounterApplication from Figure 5.4b. A generalization edge indicates the inheritance relation between Counter and CountingCounter. The instantiation of the CountingCounter in the CountingCounterApplication SCChart is reflected by the association as an aggregation.

Figure 5.11a illustrates how SCCharts adapt notational aspects of UML class diagrams to augment the classical SCCharts representation. Figure 5.11b presents the same model in a more classic UML class diagram style. A separate section of properties was introduced to list regions.

This pragmatics-aware modeling approach for OO SCCharts seamlessly combines OO programming with the OO design methodology. Additionally, it addresses the important issue of documentation [HLF+22]. In software development, there is often the problem that the actual implementation starts to diverge from the architecture defined in an earlier stage or the documentation, if not kept in sync. In SCCharts, the model acts as the source for



(a) SCCharts notation **(b)** Classical UML notation

Figure 5.11. The CountingCounterApplication and the involved classes arranged in a UML class diagram style.

the generated code and the documentation. Using automatically generated graphical views, the tasks of designing, implementing, and documenting a system start to merge, while handling a single model.

Transparent Compilation The KIELER compiler [SSH18c; Smy21] also facilitates working with OO SCCharts. Its modular approach is influenced by the idea of transient views and produces accessible intermediate results for each step of the compilation. Hence, a user can inspect and verify the effect of each transformation presented in this chapter, down to each individual macro expansion.

5.4 Deterministic Objects

A core feature of synchronous languages is their deterministic concurrency. In the classical approach, described in Section 2.1, the program is subject to an analysis that determines data accesses of individual elements (e. g.,

statements), establishes causal relations, and tries to find a deterministic scheduling based on the respective MoC. However, this white-box approach is limited by the availability of information and is often too fine-grained to support separate modular compilation.

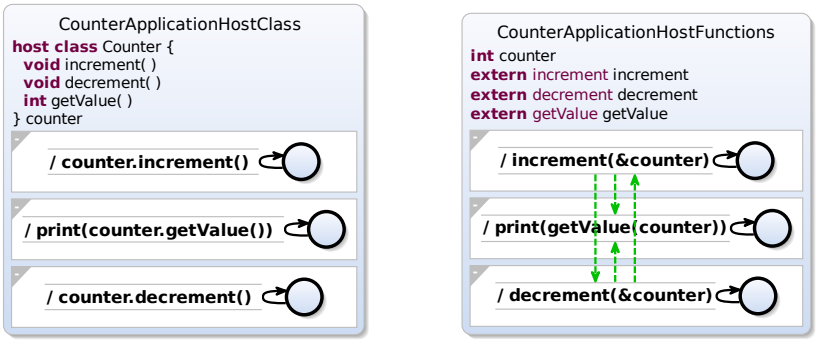
Normally, the analysis ends at the level of the synchronous language and does not reach into the host language. Hence, the host code becomes a black-box and requires some form of causality interface [ZL08] to enable safe interaction from within the synchronous program. This is best exemplified by LF's black-box reactions but also by the host code integration in Esterel and SCCharts.

In separate compilation, the underling rationale is to utilize existing modularity and compile components separately. In turn, this requires some from of causality interface that decouples the compilation process and can act as a placeholder when including a module now reduced to a black-box. Again, LF's reactors are an example for this, same as activities in Blech, which are both designed to be compiled independently. There are also modular compilation approaches for synchronous dataflow languages [BCH+08; TL18; PR10], Esterel [HPB+99], or SCCharts [Lüd21; Smy21].

While these considerations are not new and solutions have been researched, this section revisits this topic in the context of OO in SCCharts. The primary research question is: What are mechanisms to include objects from the host language in synchronous languages, while acknowledging their inherent OO characteristics and providing a flexible way to achieve determinism?

A solution will in turn facilitate modular OO compilation, if SCCharts supports interacting with a black-box object that could result from a separately compiled SCChart. Yet, such an application remains future work, as discussed in Section 6.3.5.

The solution proposed in this section will advocate a contract-like interface for objects that expresses scheduling instructions, rather than data access, to establish internal determinism. Specifically, SCCharts will utilize the existing Scheduling Directives (SDs) [SSH19] and Scheduling Policies (SPs) by Aguado et al. [AMP+18] for this purpose.



(a) The counter defined as a host class with methods. (Publ. in [SSM19; SSM21] ©2019 IEEE)

(b) The counter defined as a simple integer with external functions for manipulation.

Figure 5.12. Two variants of the CounterApplication using host code.

5.4.1 Black-Box Scheduling

The high-level transformations for objects in SCCharts, presented in Section 5.3, provide determinism by statically exposing all data accesses to the white-box analysis in the SCCharts compiler. For the CounterApplication in Figure 5.3, this means that the analysis detects the three concurrent method invocations and looks into their implementation, if not already exposed due to inlining. This yields the result that increment and decrement perform a relative write on the value variable of the counter object, while getValue reads this value. The IURP then prescribes that both the increment and decrement method call (or the actual increment/decrement statement, if inlined) must happen before the reader in getValue, to form a valid SC admissible schedule. Since the two write accesses are commuting, they can be scheduled in any order without jeopardizing determinism.

Figure 5.12a shows the same program but with the Counter class defined in the host language. The host class syntax is a variant of the native class notation used in the compilation of SCCharts-based classes, discussed in Section 5.3.1. The declaration makes the class and the signatures of its methods known to the SCChart. Like extern declarations, the actual

implementation is only linked into the program at compile time. A more detailed description of OO host language integration in SCCharts is available in the corresponding journal on OO in SCCharts [SSM21].

In this model, the methods are now black-boxes and the analysis can no longer determine internal data accesses. Unless the synchronous compiler parses the host language, which, however, would be contrary to the idea of black-box host code. The absence of information about data accesses renders the program vulnerable to non-determinism, since there is no restriction on how to schedule the regions, and the method calls could appear in any order, potentially leading to data races. Hence, the return value of `getValue` is different depending on whether it is executed before or after an `increment/decrement`. Note that for non-parallel SCCharts execution and in the absence of data dependencies between regions, the scheduling falls back to the syntactical order of the regions in the source code. Yet, this still yields a different behavior in this example, since the value would be printed before it is decremented.

Causality Interfaces in Functions The issue of non-determinism with black-box function calls is well-known to synchronous languages. It is usually avoided by demanding that external functions must not have any side effects through shared memory. Hence, shared data has to be passed between these functions through the synchronous language. Figure 5.12b presents a variant of the `CounterApplication` that uses the classical approach of host code functions. The three methods are now external host functions and the `counter` that was previously an internal private member of the `Counter` class is a variable in the SCChart. The invocations then pass the shared variable to the functions. The parameters now act as a causality interface, where call-by-value arguments are considered read and call-by-reference arguments are read-write accesses on the respective variable. Other languages, such as Blech or Esterel, determine the mutating behavior of function arguments via separate parameter lists.

This approach raises the internal data relations into the external input output interface of each function. However, such an interface only allows limited scheduling decisions. For example, this model still has to be rejected because the concurrent calls `increment(&value)` and `decrement(&value)`

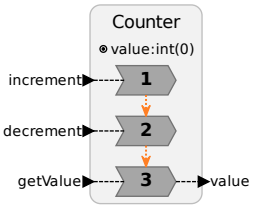


Figure 5.13. An attempt to model the Counter class as a reactor in LF.

both read and write the variable. This constitutes a causality cycle under a write-before-read scheduling. The additional write-read dependencies in Figure 5.12b illustrate this fact. The SC semantics can only accept this program if the analysis is able to detect commuting relative writes, i. e., updates. Hence, a more expressive form of scheduling information is required to enable scheduling beyond a write-before-read protocol.

A more fundamental problem with the aforementioned approach is its incompatibility with the OO principle of encapsulation. In the example in Figure 5.12b, the notation for the counter changed from a class to a set of functions, in order to illustrate the extraction of internal members. This clearly violates the idea of information hiding in objects. If one tries to keep all object members in one data structure to retain encapsulation, it further complicates scheduling. This "self" struct would need to be passed to every method associated with the object, to provide read-write access to its members, creating bidirectional dependencies, i. e., causality cycles, between all of them. After all, the basic purpose of an object's method is to access and manipulate the object's internal state. In other words, methods have side effects on their object by nature, which makes them incompatible with existing mechanisms and assumptions about host functions in synchronous languages.

Causality and Ordering in Reactors When interacting with the host language while permitting side effects, a causality interface for inputs and outputs is necessary but not sufficient, additionally there has to be some ordering that establishes determinism internally. The same principle can be observed in LF, which provides OO encapsulation with black-box code.

Figure 5.13 shows a reactor that mimics the Counter class in Figure 5.2b. It has input ports for triggering an increment or decrement and getValue that will issue an output at the value port. Other than methods in SCCharts, the associated reactions can only be triggered once per tick, further discussed in Section 5.5.3. The counter value is an internal state variable. Note that this variable is not exposed in the causality interface of reactions nor the input output interface of the reactor. Instead, each reaction has an implicit read and write access (side effect) on state variables and determinism is established by the fixed lexical ordering of reactions. In this diagram, the orange arrows indicate this relation in addition to the numbering.

This sequential ordering is an effective way to establish determinism inside an object. Yet, given the context of shared objects in SCCharts, such a strict order would reduce the extent of utilizable concurrency, since increment and decrement are actually commuting in this example.

Objects with Scheduling Contracts The proposed solution for deterministic black-box objects in SCCharts is similar to the concept of LF. Classes can encode a contract that prescribes the scheduling order for its methods in a concurrent context. This augments the causality imposed by the input output interface of methods and abstractly expresses inter-method dependencies imposed by hidden data dependencies and side effects. The contract is enforced upon concurrent invocation on a per-object basis. The contract could also be used beyond establishing determinism in the presence of concurrency and govern the general interaction with objects, i. e., by prescribing an order even for sequential accesses, but this is not in the focus of this thesis.

For host classes, which are always treated as black-boxes, there is a default contract. This conservative version imposes an invocation order corresponding to the lexical order of declaration in the class notation of SCCharts. It prohibits multiple invocations of the same method, since methods cannot be assumed to be commuting to themself. Normal SCCharts-based classes do not have a default contract because they are scheduled under white-box SC. However, in both cases, a user-defined contract can be specified that supersedes the default ordering.

The following sections will present two existing concepts for implementing such contracts and will adapt them to objects in SCCharts. The first approach are *Scheduling Directives (SDs)* by Smyth et al. [SSH19] that are already implemented in the SCCharts compiler and are designed for the same purpose: adjusting the default scheduling in synchronous languages. Section 4.2.2 already illustrated their utilization in prepending a clock update phase for concurrent clock use. Second, *Scheduling Policies (SPs)* by Aguado et al. [AMP+18] inspired the idea of object-specific contracts. SPs augment objects with a precedence interface based on access methods that subject its scheduling to the specified precedences. With their automaton formalization, SPs are more powerful than SDs and fit well into the modeling approach of SCCharts.

To illustrate the approaches, the CounterApplicationHostClass SCCharts from Figure 5.12a is continued as an example. Its default contract is replaced by a custom ordering that permits clients to concurrently invoke increment and decrement multiple times and in any order, but strictly before any calls to getValue. This results in a deterministic value read from a counter object in every instant and corresponds to the semantics under a white-box SC scheduling.

5.4.2 Scheduling Directives

SDs are designed to facilitate resolution of causality problems in a user-defined way. The idea is to augment the mechanism of casual data-related dependencies imposed by the MoC, e. g., the IURP, with precise rules that aid the compiler in accepting programs that would otherwise be rejected. Therefore, SDs associate a *scheduling unit*, such as a single assignment or a region, with a *named schedule* and an *index*.[12] All SDs associated with the same named schedule must be scheduled according to their index, lowest index first. This induces a new schedule that overrides the pre-defined synchronization protocol of the synchronous language.

As it turns out, this approach is also well suited to introduce scheduling rules for objects, especially since SDs can express non-trivial causal relation.

[12]The notation of SDs avoids the term "priority" to prevent confusion with the priorities of priority-based scheduling [HDM+14], where the highest value priority is executed first.

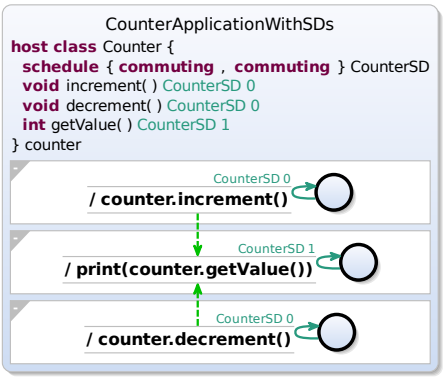


Figure 5.14. The CounterApplication with a host class and a custom schedule. The resulting scheduling instructions are visualized as green arrows. (Publ. in [SSM19; SSM21] ©2019 IEEE)

In an SD-defined schedule each index can be declared *conflicting* or *commuting*, with the former as default. If multiple scheduling units are associated with the same index, a commuting group permits these units to occur in any order, while concurrent execution of conflicting units is rejected by the compiler.

Classes with Scheduling Directives Figure 5.14 shows the CounterApplication from Figure 5.12a using SDs. The definition of the host code class is augmented by a named schedule CounterSD that has two phases, both commutative in their group. The index 0 is assigned to increment and decrement, meaning that their invocations can be ordered arbitrarily but must occur before any calls to getValue, which has assigned the higher index 1. Index 1 is also commuting because reading has no causal implications in this case. As a novelty to classical SDs, the actual scheduling units, i. e., the method invocations, draw their SD from their declaration. The diagram also shows these implicit SDs on the transitions and additionally illustrates the imposed ordering. Furthermore, the current SCCharts compilation ensures atomicity of black-box method calls, which, therefore, can be treated as single scheduling units.

SDs were implemented in the SCCharts compiler by Smyth [Smy21] and are extended here, to support their application in host objects and SCCharts-based classes. The fact that SDs are compiler instructions leads to their complete consumption during static scheduling and imposes no overhead on the generated code or runtime.

5.4.3 Scheduling Policies

A more general form of a scheduling contract is an SP. It augments an object or memory cell with a *policy* that specifies the admissibility and precedence of its access methods in a concurrent context. SPs use an automaton formalism to encode a *precedence graph*, which facilitates a high level of abstraction and enables more advanced stateful scheduling regimes than SDs. They are capable of generalizing scheduling protocols of synchronous MoC for individual shared ADTs. In contrast to the policy model by Caspi et al. [CCG+09], SPs by Aguado et al. [AMP+18] permit destructive updates and sequentiality, which is particularly relevant in the context of SCCharts. SPs act as a contract between objects and the program, and they require the enforcement of a *policy-conformant* scheduling at run-time or statically at compile time, where any inability to do so implies a constructiveness problem. At the same time, a policy also requires a *policy-coherenent* implementation of the objects itself, such that the object will behave deterministically if accessed in accordance to the given SP. Ensuring or checking this property is not in the focus of this thesis, and the following concept will assume that users will provide correct policies w.r.t. the implementation.

Classes with Scheduling Policies Figure 5.15 shows the CounterApplication from Figure 5.12a using an SP. The CounterPolicy is specified inside the Counter class and visualized as a policy region. The syntactic elements are based on SCCharts' states and transitions but adjusted to the SP formalism. The automaton has two states, count and read, which capture the two different scheduling modes, before and after the first read access to the counter. Initially, in state count, invocation of all three methods increment, decrement, and getValue are admissible. This is expressed by the availability of an instantaneous transition labeled with the name of the method and a *blocking*

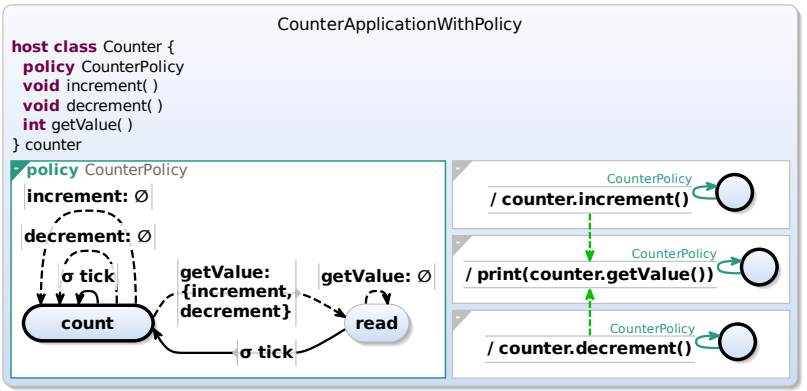


Figure 5.15. The CounterApplication with a host class and a policy automaton. The resulting scheduling instructions are visualized as green arrows. (Publ. in [SSM19; SSM21] ©2019 IEEE)

set, separated by a colon. In the case of getValue, it states that any admissible invocation of getValue must wait for all concurrent calls to increment or decrement, which take precedence. On the other hand, the blocking sets of increment and decrement are empty and thus are neither blocked by getValue nor each other. This corresponds to the commuting property inside the first scheduling group, when expressed by SD in Figure 5.14. In the given policy automaton, the first concurrent invocation of getValue will switch the state to read. There, calls to increment or decrement are no longer admissible, only invocations of getValue, which are again commuting as this transition leads back to read and has an empty blocking set. The solid (non-instantaneous) transitions labeled σ tick represent the synchronous clock that starts a new instant and resets the SP to the initial count mode.

The policy imposes the same precedences onto the method invocations in the CounterApplication as in Figure 5.14. This is again illustrated by the additional arrows in the diagram.

Partial Implementation SPs provide a powerful formalism for prescribing schedules. However, this includes schedules that may require runtime-dependent changes in the ordering of methods, for example, if a policy

does not return to its initial state in each tick. Such policies are not supported by the current static compilation approach in SCCharts and either require a policy-conformant scheduling at runtime or an exploration of the program's state space at compile time, e. g., as illustrated by Aguado and Duenas [AD21] utilizing model checking. Hence, SPs in SCCharts only support a subset of policy automata at this point, specifically those that can be statically transformed into SDs.

SDs are a special case of stateless SP automata that always return to their initial state after a tick and specify precedences which methods can be uniquely assigned to scheduling groups (indices) and classified as conflicting (listing themselfs in their blocking set) or commuting. The transformation of SPs in SCCharts analyzes the automaton and generates SDs based on a topological sort of precedences. If the structural requirements are not met, the program is rejected. Hence, the provided implementation of SPs in SCCharts is only partial and acts as a proof of concept. Note that the SP in Figure 5.15 fulfills the requirements for transformation and will result in the SD shown in Figure 5.14.

A Foundation for Stronger Decoupling SPs for classes, same as SDs, act as contract between the object and the scheduler, as well as the class and its implementation. As such, they can be utilized beyond host objects and formalize the safe interaction between shared objects in SCCharts in general. This can also facilitate future development of modular OO code generation for SCCharts. If the compiler automatically synthesized policies for SCCharts, to specify the precedences between, e. g., regions, this could enable instantiating these SCCharts as black-box modules in other SCCharts. This would facilitate the separate compilation of such modules in a finer granularity, further discussed in Section 6.3.5.

As mentioned in Section 5.3.3, a fixed scheduling contract could also ease the handling of subtyping in the face of runtime mutable object references. If all subtypes must be policy-coherenent to their supertype's policy, a policy-conformant scheduling of an object reference could be established independent of the actual subtype.

5.5 Evaluation

The features proposed in this chapter introduce core capabilities of OO design and programming for SCCharts. They are based on a careful assessment of the OO methodology in Section 5.2 and continue some existing concepts present in Blech, LF, or O-Charts.

After a reflection on goals and the extent of the introduced features, Section 5.5.1 will briefly assess implications of the high-level transformation approach. Next, Section 5.5.2 will evaluate the class modeling capabilities of the OO extension by turning SCCharts' built-in signal types into classes. Section 5.5.3 will follow up with a discussion on the relation of methods to the classical concept of signal interfaces in synchronous languages. Finally, Section 5.5.4 presents a case study on a steam boiler controller modeled in SCCharts using OO features.

Reflection on Goals As a first informal evaluation, the initially set goals are examined for their fulfillment in the proposed concepts (cf. page 167).

Conservative Based on the assessment in Section 5.2, the OO features for SCCharts are carefully selected. Restrictions are only introduced to conservatively enforce determinism, facilitate static analysis, or to limit the implementation extent for this thesis. Additionally, these restrictions are only functional and do not affect the investigation of the language design itself. Furthermore, the presented high level transformations illustrate that the new features can be conservatively grounded in the existing language core and semantics of SCCharts.

High-level All introduced OO modeling capabilities can be handled as extended features in SCCharts and transformed by the proposed high-level approach, which seamlessly integrates into the existing compiler. As a result, the OO features do not require an OO host language and can, for example, be compiled into C code. Yet, the design is not restricted by this approach, since the new language constructs are in many ways inspired by OO general purpose languages, which facilitates utilizing their capabilities downstream. Section 6.3.5 will discuss a sketch for a code generator that carries OO constructs down to the host language

level. Another benefit of the high-level approach is that it facilitates inspecting and verifying the results on the source level.

For host code objects, scheduling contracts provide a high abstraction level, enabling a black-box approach that is not bound to specific OO host languages or relies on parsing external code. With SPs, the specification of custom scheduling regimes becomes a high-level modeling task.

Pragmatics-aware As discussed in Section 5.3.4 and illustrated by the various diagrams and graphical views presented in this chapter, the design of OO SCCharts embodies the principles of pragmatics-aware modeling and tooling.

Concurrent Objects in SCCharts are treated as concurrent entities, similar to LF reactors or actors in dataflow SCCharts. With the support of regions in SCCharts-based classes, the proposed concept augments classical class design with a statecharts notation capable of expressing concurrency. In addition to signals and shared variables, method calls offer a new way of communication in SCCharts, further discussed in Section 5.5.3.

Scheduling contracts are designed to enabled concurrency for shared objects under a black-box abstraction. While fixed ordering or classical write-before-read protocols may limit utilizable concurrency, SDs and SPs facilitate fine-grained scheduling classifications and are able to express confluence in the form of commuting access groups.

Deterministic The high-level transformation approach illustrates that the proposed concept is grounded in Core SCCharts and the deterministic SC MoC. The conservative restriction to immutable references enables this static approach and rules out a source of runtime non-determinism.

For host objects that may have no causality interface due to data hiding, SCCharts provides a deterministic default scheduling based on the lexical ordering of methods. Replacing this default regime by an SD or SP puts the responsibility of establishing determinism in the hands of the users.

Supported Functionality Table 5.1 gives an overview of the characteristics of the OO features and their support in the SCCharts implementation.

Table 5.1. Characteristics of OO features in the current implementation of SCCharts.

Feature	Current Support / Restrictions
Objects	see Section 5.3.1
definition	class-based, either in a classical syntax or modelled as an SCChart, which permits using regions
creation/destruction	static instantiation with static expansion for regions
references	constant object pointers
encapsulation	private, protected, and public visibility
methods	instantaneous imperative bodies; method inlining based on static dispatch
determinism	provided by white-box SC semantics, SDs, or a subset of SPs
Inheritance	see Section 5.3.2
relations	multiple inheritance with unique behavior definition; static expansion at compile-time
applicability	SCCharts-based classes and states (anonymous classes)
overriding	supported for methods and regions
Type parametrization	see Section 5.3.3
definition	declaration of generic types on SCCharts-based classes; static expansion at compile-time
subtyping	admissibility based on interface compatibility and indirectly schedulability by the causality analysis

As initially described, the dynamic runtime aspects of OO are challenging to adapt to a safety-critical and embedded domain, see Section 5.2. These are conservatively restricted in favor of static analyzability, memory boundedness, and good predictability for execution time. As a result, object references are constant and there is only static instantiation for objects, which rules out runtime polymorphism and dynamic dispatching. Some restrictions can be partially lifted by advancing analysis mechanisms, as Section 6.3.4 will describe. Others, such as the support of inheritance in states and SCCharts but not in class declarations, are not part of the proof

of concept implementation provided by this thesis. They do not require an additional concept, as the proposed approach can be easily transferred, and they do not directly contribute to the investigated research questions. In conclusion, the proposed concepts cover core OO constructs, such as a class-based designs and inheritance with overriding, and integrate them into synchronous statecharts modeling.

5.5.1 Assessment of the High-Level Transformations

The proposed high-level transformations for handling OO features in SCCharts have two main benefits. First, they ground the semantics in the existing model of Core SCCharts. Second, extended features can be seamlessly integrated into the existing compiler and enable target non-OO host languages, such as C, which are relevant for many embedded platforms. Yet, the proposed language extension is not designed to be limited by this approach. Instead, OO constructs could be transferred to an OO capable host language, as Section 6.3.5 will discusses in the proposal for a future OO code generation approach. This would reduce the code size of programs that instantiate (reference) the same SCCharts multiple times. Such results were observed by Lüdemann [Lüd21], when testing SCCharts with a modular compilation.

Treating OO modeling capabilities as extended features and removing them for the final program comes at a price. Clearly there is a cost in performing the transformations at compile-time. However, tests show that it is relatively negligible and less relevant than a potential runtime impact. In order to assess any implication of the high-level approach, all OO models presented in Section 5.3 and some additional test cases were also modeled with the classical module approach, to enable a direct comparison.

The Non-OO Logger Figure 5.16 shows such a re-modelled variant of the `LoggingApplication` example in Figure 5.8. It utilizes referenced SCCharts, shown as state with a purple background (see Listing 2.3 as an example in textual syntax), to instantiate the `LogMessage` and `LogError` module. The general design of the SCChart is different from the OO variant because the modules in their given form do not allow expressing a common interface

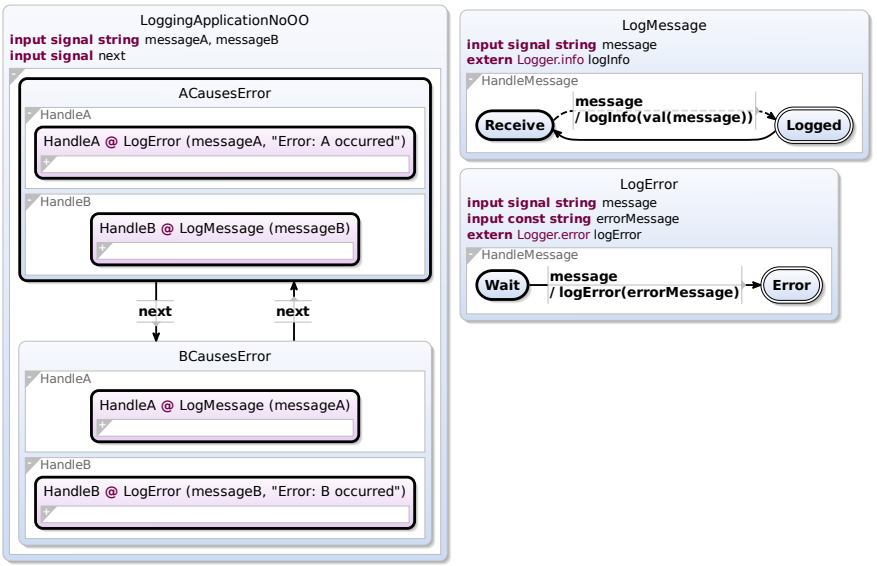


Figure 5.16. Logger example modelled using the classical module approach.

or adjustable default behavior.[13] Instead, the modules individually provide the different logging behaviors, in case of this simplified example only two. The LoggingApplicationNoOO SCChart composes them in the two states to create the desired behavior. The inputs are used to create some degree of reusability given the different context of ACausesError and BCausesError.

Results Executing these models in a benchmark with fixed input traces indicated no drawbacks in runtime performance when using an OO design instead of modules. Inspecting the generated code of OO and non-OO variants reveals nearly the exact same structures and instructions, which underpins the runtime observation.

[13]SCCharts' modules could be extended by a form of *procedure parameters* with default values, in this case for regions, to provide more flexibility and adjustability. The result would be quite similar to type parameters in Section 5.3.3 and would turn SCCharts into higher-order functions. However, in my opinion the OO paradigm is a more natural and well-established way to approach this.

One minor difference is an additional level of hierarchy that is introduced when using modules. In Figure 5.16, state ACausesError has a region HandleA that has a state HandleA in order expand the SCChart LogMessage into this state. Only this state will have the region HandleMessage with the actual business logic. In contrast to that, the OO variant using inheritance puts the regions with the message handling logic directly in the ACausesError state, see Figure 5.8. It is a consequence of the module design in SCCharts that requires a dedicated state to instantiate an SCChart module. Similarly, the instantiation of SCCharts as objects via references, as described in Section 5.3.1, does not introduce this additional state either. Assume that the modules in the FurutaPendulum SCChart were instantiated using this new OO approach. This would eliminate the need for regions such as Controller in Figure 1.3 and directly integrate the variable and region of the PendulumController into the FurutaPendulum SCChart. Alternatively, the additional hierarchy level needed for modules could be optimized by the compiler or addressed by introducing a more compact syntax.

Independent of this aspect, the benchmarks with OO and non-OO models did not indicate that the OO approach is more costly than classical modeling. However, in general purpose OO languages, this is sometimes the case. One reason are lookup tables for polymorphic method calls that impact performance. Yet, this aspect is excluded in SCCharts as discussed in Section 5.2.

Design Methodology The comparison between the OO LoggingApplication in Figure 5.8 and the non-OO variant in Figure 5.16 illustrates that the proposed OO features in SCCharts provide the user with alternative ways of creating reusable, adjustable, and modular SCCharts. While this can be expected from a successful integration of the OO design methodology, this evaluation will refrain from attempting to generally quantify a benefit of OO over other design principles. As initially mentioned, this is a controversial topic and this thesis assumes the relevance of the OO design paradigm, referring to existing surveys on the effect of different design methodologies [Wie98; PLR95].

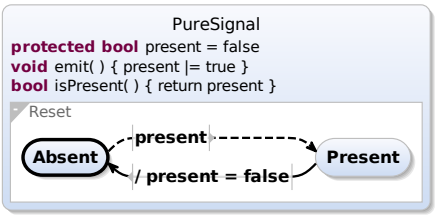


Figure 5.17. The PureSignal class modelled in SCCharts.

5.5.2 Modelling Signals as Classes

Signals are the backbone of classical synchronous programming [BCE+03]. In SCCharts, they are subsumed by SC-variables. Nonetheless, SCCharts provide built-in signals, as for example illustrated in Figure 5.8. Signals in SCCharts are an extended feature and encoded as boolean variables during compilation [HDM+14; Mot17; SMR+17]. The following case study investigates an alternative approach to the hard-coded transformation and uses the new SCCharts-based class modeling capabilities to create user-defined class for signals, evaluating the proposed OO concepts in the process.

The Pure Signal Class A signal in its pure form is reset to absent in each tick and can be emitted, resulting in a present state. The SCChart in Figure 5.17 represents a class with this behavior. It encapsulates the signal's state in the internal variable present and provides the method emit for emissions and isPresent to retrieve the state. The emit method sets the present state by using a relative write. This classifies it as an update and, under the IURP, permits confluent concurrent emissions, while ordering it after the initialization to absent at the beginning of a tick. The reset behavior is modelled in the Reset region of the class, since it is inherently active during the lifetime of an instance of this class. In any tick the signal is emitted, this region will immediately switch from the Absent state to Present, which causes a reset of the present state in the next tick, due to the delayed transition back to Absent. Alternatively, one could use a during action that resets the present in every tick.

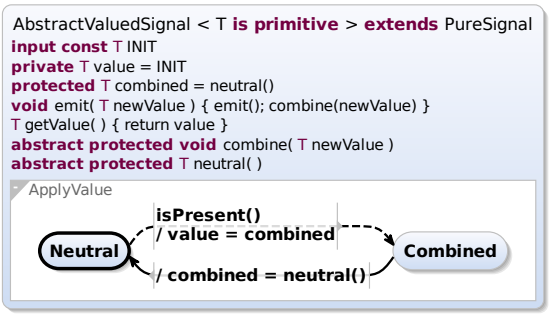


Figure 5.18. The AbstractValuedSignal class that carries a generic value and requires the implementation of a combination function.

To evaluate this design against the built-in signals, existing SCCharts models that use built-in signals were adjusted to use the PureSignal class. The test set included SCCharts that are used in the continuous integration of the SCCharts compiler to check the correct behavior of built-in signals, as well as a variant of the traffic light controller in Section 4.2 by Wechselberg et al. [WSS+18] that uses signals. Comparing the generated code of both approaches revealed very little differences, if the classes were fully expanded and inlined. However, one advantage of the built-in variant is that it consolidates the reset behavior of all signals into a single region, while the class-based approach resulted in one region per signal instance. This more distinct use of concurrency comes in favor of a more modular design.

The Valued Signal Class In addition to the pure signal behavior, valued signals carry a persistent value that is set upon emission. In case of multiple concurrent emissions, a combination function must deterministically merge the different values into one. Figure 5.18 illustrates the SCCharts-based class AbstractValuedSignal, representing such a behavior independent of a concrete value type or combination function.

The AbstractValuedSignal SCChart extends the PureSignal class and declares a type parameter T that is restricted to be primitive. The primitive supertype was introduced to permit only basic variable types, such as int or bool, and

not classes. This enables the compiler to handle these types different to ref declarations, turning them into normal variable declarations, which can be assigned in contrast to object references in SCCharts. Hence, this new supertype is a technical consequence of the design decision to prevent writes on object pointers.

The AbstractValuedSignal class has two variables, value and combined, both of type T. The value represents the persistent value of the signal and is initialized by the constant input INIT. The combined variable is part of a two stage process that first combines the values of all emissions in a tick into one combined value, and then replaces the previous value of the signal. The mechanism corresponds to the built-in implementation of signals in SCCharts [Mot17]. The overloaded emit method invokes the pure emission to set the present state and afterwards calls the abstract combine method, which is responsible for performing the update on the combined value. In order to work correctly, the combined variable must carry the neutral element of the combination function, which is provided by the abstract neutral method. For example, a concrete implementation for integers with a sum combination would return 0 as the neutral element and implement the combine method with combined += newValue in the body. This represents an OO approach to concretizing the combination function, but one could also imagine extending SCCharts further and allowing functional parameters to solve this differently.

Additionally, this class contains a region named ApplyValue that sets the value to the combined value of all emission in a tick. At the beginning of the next tick, it resets combined variable to the neutral element to enable a new round of value combination. As a result, each instance of this class will have two regions running. Alternatively, one could override the Reset region of the PureSignal to jointly handle the reset of the variables in one region. However, the presented design favors encapsulation.

Like in the case of pure signals, the tests confirmed that the implementation of the AbstractValuedSignal class provides the same behavior as the built-in signals and the generated code only showed insignificant differences.

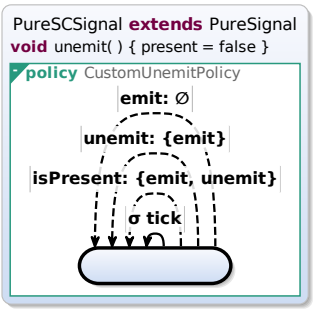


Figure 5.19. The PureSCSignal class extending pure signals by an unemit method with a custom scheduling protocol.

The Customized Pure SC Signal Class A main advantage of having signals as accessible classes instead of a hard-coded transformation is that they can be easily extended by the user. For example, in SCEst [SMR+17], signals were adjusted to the characteristics of the SC semantics. These SC signals feature an *unemit* that allows the user to manually reset the present state and consequently permits multiple states during a tick, as enabled by the SC MoC.

Figure 5.19 illustrates such an extension by adding an unemit method. Additionally, this PureSCSignal is further customized to provide a different scheduling regime than proposed in the SCEst definition. Originally, an unemit takes presence over emissions in a concurrent context [SMR+17]. This naturally fits in to the IURP because an unemit is implemented as an absolute write and emissions are updates. However, this example uses an SP to override this regime and schedules concurrent unemits after emissions, allowing to suppress emissions in a concurrent context.

Final Assessment These examples illustrate the ability of OO SCCharts to express customizable classes for complex data types by combining classical OO programming concepts with modeling SCCharts. At the same time, the object abstraction lends itself well to flexible user-defined scheduling contracts defined at the class level and independent of the specific caller context.

5.5.3 On Methods and Signals

Most examples in this chapter that illustrate OO modeling in SCCharts use methods for interacting with objects. This is a deliberate choice as methods were specifically introduced to provide a common imperative OO programming style, shaped by languages such as C++ or Java. Yet, this raises the question whether it constitutes a fundamental break with the principle of using signals or shared variables for communication between modules, as it is the case in classical synchronous languages.

Input Output Interfaces A first observation is that the new concept of methods does not replace the classical interface of SCCharts, but constitutes an alternative or addition that is tailored to objects. SCCharts-based classes still can use input or output variables that will be bound at instantiation, just like SCCharts modules, see Section 5.3.1.

Moreover, the tick functions interface to the environment remains the same, see Section 2.3.2. Method calls are used for internal interaction with objects and are not supposed to be called from the environment, since this would interfere with the synchronous execution of a tick. Given the inlining approach, they might not even be available for this purpose. At the same time, host class methods enable a more natural handling of host objects, which can be used to communicate with the environment. Yet, this is not a conceptual novelty given the previous concept of host functions.

Multiple Invocation Basically, methods and variables or signals in an input output interface serve the same purpose. They pass control messages and data to and from the object or module to affect its behavior.

Figure 5.20 illustrates an attempt to model the `Counter` class from Figure 5.2b as a classical SCCharts module. Two boolean inputs can be set to trigger an increment or decrement of the counter. The `value` variable is provided as an output and two during actions process the inputs and update the value accordingly.

The most striking difference to the design with methods is the inability of this SCChart to perform multiple increments or decrements in one tick. Of course this is a consequence of using a single boolean input. However, even if one uses an integer input variable to indicate the number of operations

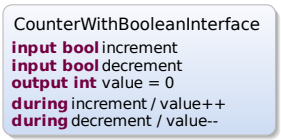


Figure 5.20. The Counter class modelled as a classical SCCharts module with boolean variables for interaction.

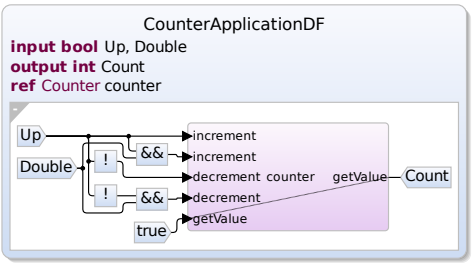


Figure 5.21. The CounterApplicationDF SCChart using the Counter class with methods in a dataflow region.

(increments or decrements) that should be performed, the counter itself is not affected multiple times but once. The environment of this module has to combine all "operation invocations" into a single input value that is passed to the counter and read in the during actions. Consequently, this counter module cannot be used as a class in a sequential context, where instantaneous writing and reading is permissible in any number, order, and combination under the SC MoC. In other words, only with methods supporting multiple invocations, SCCharts can model objects that can harness the benefits provided by the SC semantics in terms of sequentiality.

Methods and Dataflow Methods are beneficial for the imperative programming capabilities in SCCharts. They and their invocations represent a control-flow perspective that matches the SC semantics in which sequentiality can be utilized. On the other hand, classical input output variables rather correspond to a dataflow view, where there is no control-flow invocation but a signal or boolean value indicating the presence of an operation and its accompanied data.

Technically, there is very little difference between these concepts in SCCharts. Considering the fact that methods can be inlined by the compiler, this places the method's body and internal data access in the caller's context, similar to writing to an input variable. However, objects use encapsulation

to permit only regulated operations under abstract aliases (method names), e. g., emit and unemit in Section 5.5.2.

While inlining illustrates a tight integration into an imperative program, the message passing nature of methods also enables a dataflow view. This is best represented by an actor-oriented design [Lee03; LLN09]. It establishes the objects as concurrent components and explicitly models their communication. As discussed in Section 2.2, LF embodies this nature. Considering the Counter reactor in Figure 5.13, input ports receive the operation events handled by reactions, similar to methods. However, reactions can only execute once per tag. This fits well into the dataflow nature of actors because there usually are no control-flow elements that could express the sequential relations of multiple instantaneous events. Instead, one would use LF's superdense time model and spread out the events across separate sequentially ordered microsteps, in order to enable separate reactions. This differs from the concept of performing multiple instantaneous method invocations. The work of Rentz et al. on representing C code in actor models also illustrates the challenges of expressing instantaneous control-flow aspects in a dataflow notations [RSA+21].

Methods in Dataflow SCCharts With dataflow regions, there is also an actor notation in SCCharts, as discussed in Section 2.3.1. While most of the proposed OO features, such as inheritance, easily adapt to this notation, just like in LF, methods require special consideration.

Figure 5.21 illustrates the CounterApplicationDF SCChart that uses the Counter class with methods from Figure 5.2b in a dataflow region. The counter is incremented in each tick when the Up input is true and otherwise is decremented. The Double input doubles the number of steps per tick and Count conveys the counter value as an output. The connections to the counter actor are inspired by the LF mockup for the counter in Figure 5.13. A boolean typed wire indicates the invocation of a method. It is represented by a special port figure that should illustrate the more event-driven nature of methods. The input wire could be considered an individual clock signal for the method. Multiple invocations are visualized as different port instances of the same method. Return values, as in the case of getValue(), are produced at output ports. An additional internal edge, the dotted line, associates the

method invocation port with the corresponding return value port. Similarly, parameters would be received as separate input ports, also associated with the method's triggering port.

This approach does not constitute a fleshed out proposal for methods in actors, but instead should act as a proof of concept to illustrate that the introduction of methods did not result in an incompatibility to SCCharts' dataflow notation. Note that in the compilation of dataflow regions, methods do not impose a challenge either, because the dataflow is transformed into normal control-flow SCCharts. A future refinement of the visualization approach in Figure 5.21 may also be used to create views for regular OO code, continuing in the direction by Rentz et al. [RSA+21]. It could further act as an interesting use case for investigating sequentiality in SC dataflow, as started by Grimm et al. [GSS+22].

5.5.4 System Design Aspects of an Object-Oriented Steam Boiler Controller in SCCharts

The OO design principles become most effective when structuring larger software systems that can be modularized and provide opportunities for reusing code. The Furuta pendulum or the examples presented in this chapter, which are tailored to briefly illustrate specific aspects of the OO extension, offer only limited opportunity to evaluate this aspect. Hence, this section investigates the system design capabilities of the proposed OO concepts in a case study modeling a more complex steam boiler controller.

The Steam Boiler The steam boiler by Abrial [Abr96] is a well-known model for control software specification and represents a CPS. In the provided scenario, the program has to maintain a safe water level in a steam boiler, e. g., in a power plant. This involves interacting with its physical devices, such as multiple pumps and sensors for throughput of water and steam. The program has to be able to detect different device and operation failures and react to that by switching into different operation modes, until the devices are repaired, or an emergency stop is required because the situation in the boiler becomes critical.

Over the years, steam boiler controllers have been implemented in several languages. For this evaluation, the implementation by Büssow and Weber [BW96] is particularly interesting because they use an OO approach to decompose the problem and derive an *architectural view*. Yet, the goal of this case study for OO SCCharts is not to recreate the design of Büssow and Weber but to use it as a point of reference in terms of system design. For the actual implementation of their program, Büssow and Weber use classical statecharts and the specification language Z [Spi89].

The main goal in modeling the steam boiler in SCCharts is to use the new OO features to the best of their ability, in order to evaluate the resulting structure. While the steam boiler example is usually used in combination with formal methods, this aspect is not in the focus of this case study.

The entire controller is too large to be presented in full. The model takes over 600 lines of code and consists of 24 individual SCCharts. Hereinafter, characteristic OO design aspects of the controller are presented and discussed.

Composition Figure 5.22 illustrates main components of the steam boiler controller and their compositions into the Controller SCChart. This view focuses on the classes that represent the real-world devices the controller interacts with, and omits several other components that are used internally to provide modular behavior, such as failure detection. The diagram further hides all declarations and methods in these SCCharts to provide an abstracted overview. The Controller hosts instances of the Water Sensor, the SteamSensor, the Valve, and the pumps. Each physical pump has a driver, implemented in Pump and a monitoring device, represented by PumpMonitor. Both components are joined into a MonitoredPump. The Controller interacts with the MonitoredPumpsControl SCChart that provides an interface to all pumps in the steam boiler, here indicated by the multiplicity Config.NUM_PUMPS. In the underlying specification, four pumps are proposed, but this design can handle any number.

This view displays the composition relations in a flat association graph. Comparing this structure to the architectural view by Büssow and Weber [BW96] reveals many similarities. They use the same separation for interacting with devices, including a MonitoredPump, but do not introduce

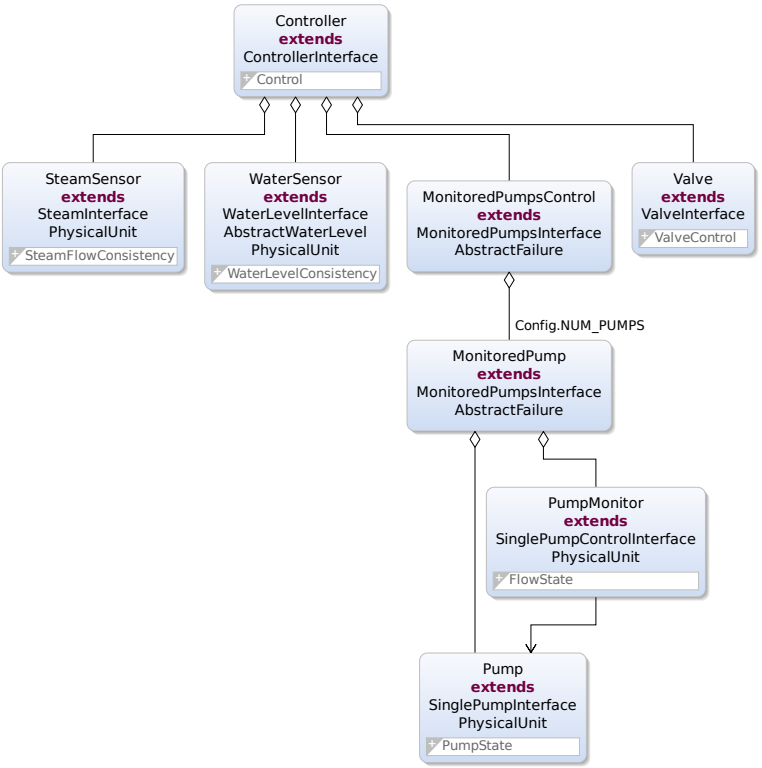


Figure 5.22. Composition of the main components in the steam boiler controller.

a MonitoredPumpsControl. Instead, they interact directly with the array of pumps, while in SCCharts MonitoredPumpsControl provides a method to activate a parameterized number of pumps. Moreover, Büssow and Weber use a UnitManager component to handle the repair protocol and failure detection of all devices. In contrast to that, the SCCharts implementation expresses this common behavior in the PhysicalUnit SCChart inherited by each individual device component, which will be discussed later. Büssow and Weber also note that a more fine-grained decomposition could further improve modularity, which seems to be the case in this implementation.

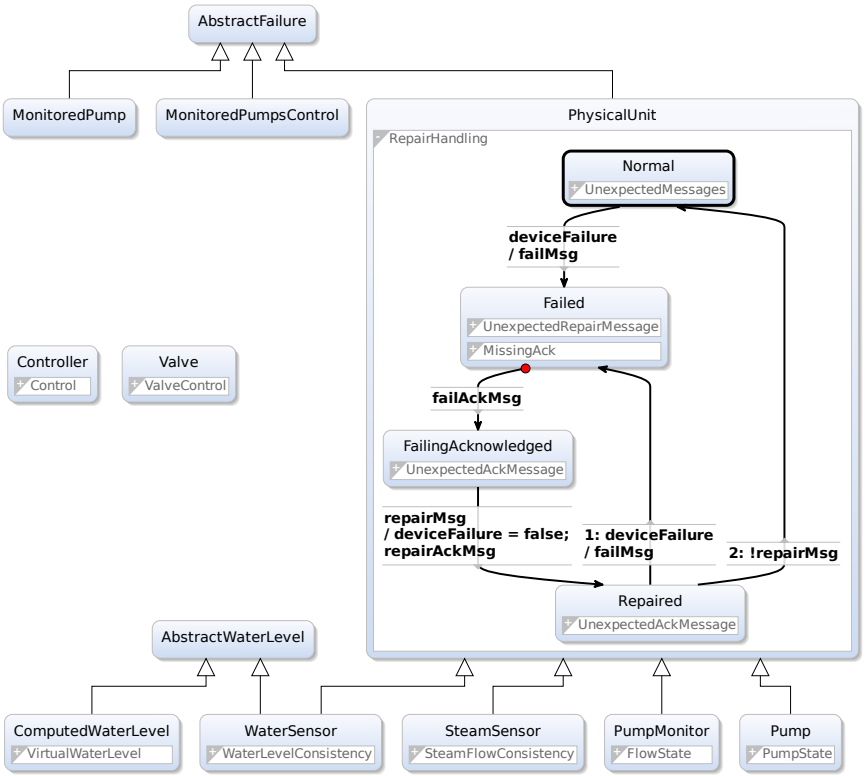


Figure 5.23. Inheritance relations between main components in the steam boiler controller.

Inheritance Figure 5.22 lists the SCCharts inherited by the main components. This includes interfaces named after devices, such as `ValveInterface`. Their only purpose is to declare the communication signals associated with the specific device, similar to the `MessageReceiver` interface in the logger example in Figure 5.8. However, more interesting are the other inheritance relations that are illustrated in Figure 5.23. This view shows the main components from the previous figure plus relevant SCCharts that they derive from. It omits the less relevant interfaces and again hides all details in the SCCharts, except their regions.

Starting at the top, the AbstractFailure is an SCChart that provides an interface and abstract behavior for transmission and device failures. It is implemented via the PhysicalUnit by each device, but also by the MonitoredPump and the MonitoredPumpsControl, since they collect and forward failure information from their contained components, see Figure 5.22. The controller will finally use the interface defined by AbstractFailure to access failure information of devices.

The PhysicalUnit SCChart is an abstract class that implements the common repair protocol for physical devices. While the actual failure must be detected by the device implementation, e. g., a negative value for the water level in WaterSensor, the communication protocol for reporting and resolving the problem is the same for all devices. As illustrated in the expanded RepairHandling region of the PhysicalUnit SCChart, a failure message (failMsg) is sent and must be acknowledged by the environment. Then, the program waits for a repair signal, acknowledges its reception, and returns with normal operation as soon as the environment stops sending the repair message. The derived classes will bind these message placeholders, such as failMsg, with concrete communication signals from their interface, e. g., LEVEL_FAILURE_DETECTION. Interestingly, the Valve has no failure and repair protocol in the specification by Abrial [Abr96] and thus does not extend PhysicalUnit despite being a physical device. Likewise, the Controller has no inheritance relations to any of these classes.

Subtyping Another relevant inheritance relation can be found in the form of the AbstractWaterLevel in Figure 5.23. This SCChart implements the classification of the water level, e. g., as *low* or *critical* according to the specification. However, it does not retrieve an actual water level value. This is implemented in the WaterSensor, which reads from the physical device, and in the ComputedWaterLevel SCChart, which is used only in the Rescue mode of the Controller.

Figure 5.24 illustrates the Controller SCChart with its Modes region in focus. Without going in to too much detail about the steam boiler behavior, the specification describes different modes of operation that a controller must provide. In case that only the WaterSensor has a failure and needs to be repaired, the controller has to enter a Rescue mode, modelled by a state

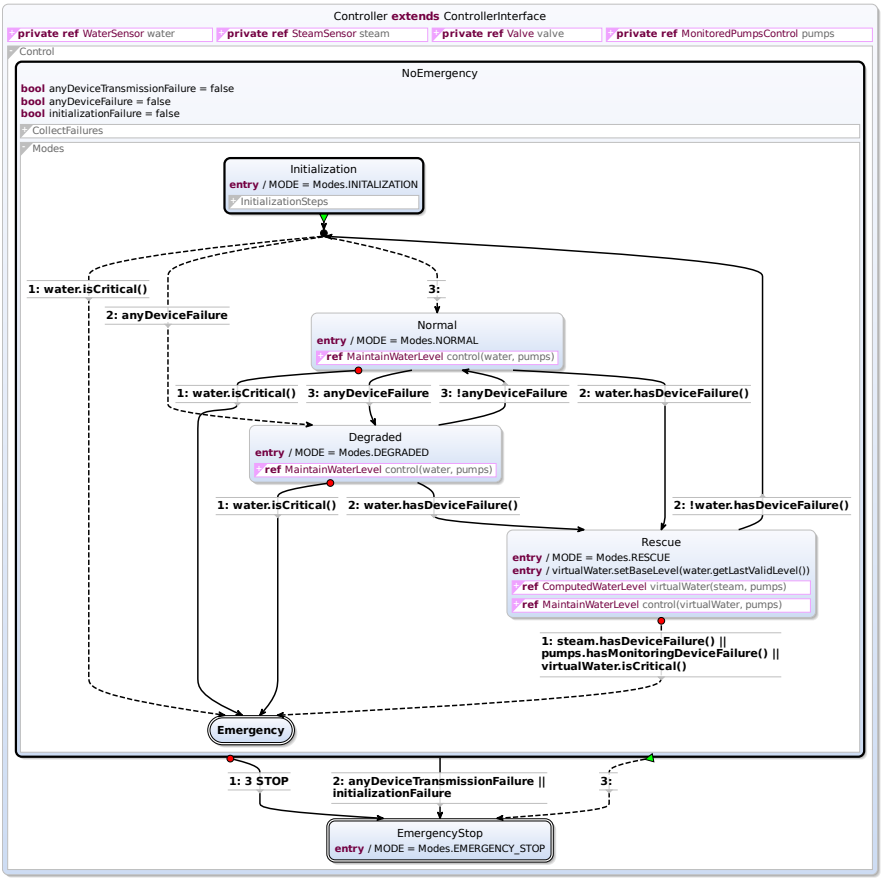


Figure 5.24. The different control modes in the Controller SCChart for the steam boiler.

with the same name in Figure 5.24. Now that there is no water sensor, the controller derives a water level based on the steam output and the throughput of the pumps. This is modelled by the ComputedWaterLevel SCChart. Since both the WaterSensor and the ComputedWaterLevel share the same super class, they are admissible subtypes. This enables implementing the control logic for the pumps in the MaintainWaterLevel SCChart independent of the source of the water level information. Both the Normal state and the Rescue state in the Controller instantiate this SCChart. In this view, ref declarations are visualized as region-like containers, which allows interactively expanding them to inspect the instantiated SCChart. In state Normal (likewise in Degraded), the WaterSensor (water variable) is provided as source for the water level. In contrast to that, the Rescue state first creates an instance of the ComputedWaterLevel (virtualWater variable) and binds the control logic to this source, which in turn draws from steam and pumps. The bindings are expressed as argument of the reference variable, which correspond to constructor arguments in a shortened notation.

Final Remarks While modelling the steam boiler in SCCharts, there were certain aspects in which SCCharts' dataflow notation would have been a likewise suitable alternative. In my opinion, this illustrates the tight relation of OO and actors, also expressed by Lee et al. [Lee03; LLN09]. However, these were only notational aspects when interacting with objects. For example, with a dataflow notation in the Rescue state, the individual input and outputs port connections between control, virtualWater, steam, and pumps would need to be explicitly specified, whereas the current notation hands over entire objects. Of course, the actor notation would expose the communication more clearly, but this aspect could also be derived automatically, as illustrated by SCCharts' induced dataflow[14], see Section 2.5.

Another aspect in the relation between dataflow SCCharts and OO SCCharts could be efficiency. With the current use of methods, a return value, e. g., of water.isCritical(), is computed only on demand, whereas in a classical dataflow region, each output is computed in each tick disregarding its use.

[14]The current implementation of induced dataflow in SCCharts does not yet support OO SCCharts and requires an extension in the future to illustrate these relations.

On the other hand, multiple invocations of the same method might lead to computing the same result multiple times. This relation between OO and dataflow in SCCharts is an interesting topic for future investigation.

The specification of the steam boiler problem itself does not assume any specific design paradigm, and this evaluation does not aim at establishing the superiority of any notation. Instead, modeling this complex scenario illustrates that the new OO SCCharts are capable of expressing an OO architecture and its implementation in one model, while various derived views can visualize different structural aspects.

Chapter 6

Conclusions

This thesis investigated various research questions in the context of language design for reactive systems. LF and SCCharts acted as examples to study modal models in a dataflow environment, real-time modeling in statecharts, dynamic execution of ticks, and OO programming and design in synchronous statecharts. Both languages embody important characteristics and principles of reactive system design, as well as the latest generation of pragmatics-aware MDE. This allowed for new approaches but also required crafting lean and seamless language extensions that carefully align with the fundamental principles of the underlying language.

The proposed concepts were successfully implemented and evaluated. All new features were warmly welcomed by the LF and respectively the SCCharts community, and received positive feedback.

Outline Section 6.1 will present a short summary of the results in this thesis. Since not every follow-up idea could be pursued, this leaves room for future continuation and improvement of some topics, addressed in Sections 6.2 and 6.3. Section 6.4 will close with a few personal remarks.

6.1 Summary of Results

Chapter 2 started with introducing and comparing the fundamental principles of LF and SCCharts. The discussion revealed important characteristics shared throughout languages for reactive systems, such as concurrency, causality, and notions of logical time. It also illustrated the degrees of freedom that exist in language design, including the different approaches of

dataflow and statecharts modelling, language features and their compilation, or practical aspects of interfacing with the environment. Many of these principles are reflected in the goals that drive the design of the language extensions in the subsequent chapters. The chapter further showed the idea of pragmatics-aware modeling in these languages.

Modal Models Chapter 3 investigated the design of a modal coordination layer in a reactor-oriented dataflow environment. The proposed solution is a seamless integration of modes into LF reactors, diagrams, and tooling. It provides a lean, polyglot, concurrent, timed, and deterministic language extension that retains the crucial black-box abstraction of reactions. The design carefully considers trade-offs, for example in terms of limitations in analyzability due to transition triggering in reactions or omitting some language features known from SCCharts or synchronous languages in favor of a leaner compilation.

The proposed concept extends the modeling capabilities of LF with modal models without breaking with existing principles. In terms of timed behavior, the notion of mode-local time is a powerful tool that enables designs otherwise tedious to achieve in LF.

Time In Chapter 4, the research focus was on modelling with time and efficient execution strategies for ticks in SCCharts, and synchronous languages in general. With timed automata, SCCharts received a well-established notation for timed modeling that seamlessly integrates into the existing SCCharts language, compilation approach, and semantics.

An important research question for timed SCCharts was the efficient execution of ticks, following the desired eager semantics for timed automata. Dynamic ticks provide the required flexibility in combination with a lean interface to the environment and the option to use physical time as input. In turn, this facilitates resilient designs that can deal with imperfections in relation to physical time by using strategies, such as soft bounds, soft resets, and logical clocks. A consequence of the proposed approach is that the model itself provides deterministic behavior independent of potential non-determinism in the physical environment.

While timed automata and dynamic ticks already existed, their combination and integration into SCCharts with a strong focus on practicability resulted in a capable timed modeling concept. The evaluation in a software experiment with LF and a real-world hardware demonstrator for dynamic ticks confirmed this. Dynamic ticks in SCCharts are capable of achieving real-time constraints, given an appropriate platform, while only performing relevant ticks. Moreover, at this level SCCharts are on par with LF, in terms of performance and expressiveness. Even if there are some conceptual differences between the two languages, the evaluation shows that many of them can be bridged by appropriate designs.

Object Orientation Chapter 5 investigated concepts for OO programming and design in the context of SCCharts. The proposed language extension provides a conservative set of features for OO modeling in SCCharts that can be handled by high-level transformations. Inheritance offers new and powerful opportunities for reusing code and enables expressing commonalities between SCCharts-based classes. These features enable the application of an OO methodology in SCCharts modelling, as illustrated in the evaluation. In combination with the pragmatics-aware design of SCCharts, this enables the integration of UML notations in to the modelling process and facilitates the automatic generation of model documentation.

Based on existing techniques for user-defined scheduling contracts, objects in SCCharts can be customized to provide deterministic behavior even if concurrently shared. As it turns out, the philosophy of the SC semantics is a natural match to objects communication via methods. This concept is crucial for integrating objects from the host language, since scheduling contracts enable a black-box treatment but also facilitate resolving causality issues in a classical white-box compilation.

6.2 Future Work on Lingua Franca

At the time of writing, LF is a vibrant project, whose future development is driven by the many contributors onto various avenues. This work contributes some more directions, primarily in terms to modes.

6.2.1 Extending the Implementation of Modes

The design of modal reactors in Chapter 3 conforms with the polyglot idea, but currently only C and Python are supported as target languages. One future task is to extend the support of modal reactors into the remaining target languages.

Modal Federations In the proposed design, a federated reactor cannot contain modes, only its federates. It might be worthwhile to investigate potential semantics and use cases for modes on a federation level. They could be used to coordinate the distribution of federates at runtime or represent a mode of operation shared with the entire federation.

Additionally, modes could contribute to the synchronization behavior in a decentralized federation. If a federate reactor has a mode that has no reactions to certain input ports, it does not need to synchronize with (wait for) reactors that supply these inputs as long a that mode is active. This may allow the reactor to respond faster and increases its availability.

Startup and Shutdown Another improvement could emerge from revisiting the behavior of `startup` and `shutdown` in modes and the general way of managing resources in reactors. One idea to address the issues discussed in Section 3.3.3 is to introduce dedicated constructors and destructors for reactors. This would separate the memory management aspect from the event processing behavior of `startup` and `shutdown`. However, this would be a major change to the way reactors are written, whereas the current solution offers a backward-compatible way of dealing with user resources in the presence of modes.

Inheritance Reactors in LF support inheritance, see Section 2.2. Yet, in the proposed syntax for modes, modal reactors rely on syntactic containment to associate reactions and other elements with a mode. This inhibits adding new reactions to an inherited mode when extending reactors because the inherited mode is only implicitly present. Hence, the mode syntax in LF requires further extension, either to explicitly associate reactions with inherited modes (e. g., **reaction mode** `M` (t) {=...=}) or to extend inherited modes by new reactions (e. g., **extend mode** `M` { **reaction** (t) {=...=} }). Adding new

reactions to modes of derived reactors would also enable changing transitions because these reactions would be ordered after previous ones and can override transition effects. Additionally, they could be used to connect new modes into the inherited modal model.

Aside from modes, LF can also benefit from OO concepts developed for SCCharts in Chapter 5, especially in terms of visualization. The UML-inspired view for OO class relations can be directly transferred to reactors, to illustrate these relations in LF models and improve LF's documentation capabilities.

6.2.2 Formal Analysis of Modes

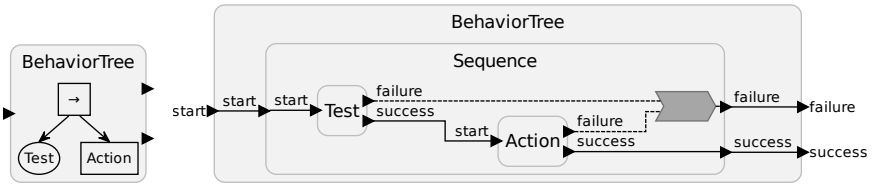
An important topic in the future development of LF is the application of formal methods. Modes facilitate such processes by statically expressing structural properties about runtime relations between reactor elements. For example, mutual exclusion of modes can help to reduce the state space that needs to be explored.

However, as mentioned in Section 3.4, there is a drawback of the current design, caused by placing the transition triggering inside the reaction code. The black-box treatment of reactions only allows a conservative approximation of effects in a static context. Actual transitions and changes in mode activity can be only determined at runtime. However, this is generally the case for any state-dependent verification in LF.

There is already a debugging tool by Deantoni et al. [DCB+21] that hooks into the LF runtime system to monitor effects of reactions. With an exhaustive simulation, it enables static verification of temporal properties. However, it does not yet consider modes.

Another approach could be the annotation of the LF reactions with additional assumptions and input output relations that are passed on to a model checker by a special compilation. Lin et al. [LML+23] outline such an approach, which is also similar to model checking in SCCharts [Sta19].

Alternatively, the program could be written in a special target language that allows a white-box analysis for verification but can also be compiled into executable code. A setup could be similar to the use of SCCharts as a target language in Section 3.4.4.



(a) Tree **(b)** Reactor network

Figure 6.1. A mockup of a behavior tree in LF. The two variants illustrate different options for representing the same behavior.

6.2.3 Behavior Trees

In recent years, *behavior trees* [CÖ17] gained attention for describing reactive system behavior. They were initially developed in the context of gaming applications but also become more popular other fields, such as robotics or artificial intelligence [CÖ14]. A key advantage is their modularity, as they describe combinations of actions in a tree structure, with each node either *running*, *failing*, or *succeeding*. Even if this notation differs from classical hierarchical state machines or actor-oriented dataflow, there are many behavioral commonalities. An interesting future research direction is the combination of these notations and computation models.

For example, Colledanchise and Ögren propose a pattern-based structural translation of behavior trees into a dataflow-like structure[1] [CÖ17]. Such a structure can easily be created in LF. Figure 6.1 illustrates a model based on this pattern. Figure 6.1a shows the BehavriorTree reactor with the contained behavior tree rendered in its classical notation. It consists of a sequence node (→), a condition Test, and an Action task. If Test reports success, Action will be executed afterwards, otherwise the Sequence node will abort and yield a failure. Figure 6.1b illustrates a reactor network with the same behavior. Each node is represented by a reactor and connections model the triggering relations. Note that this model follows a reactive semantics in

[1] In section 2.2.2, Colledanchise and Ögren call the result an FSM. However, given the fact that model does not have actual states and the "transitions" are used to convey triggering information, a dataflow network is in my opinion a more fitting description.

which tests and actions run instantaneously, rendering the running output of nodes superfluous. In order to provide the sketched modeling capabilities for behavior trees to users, LF could be extended by a dedicated syntax and an automated transformation, or it could be provided in the form of a library with generic reactors for the different node patterns.

The modal reactors introduced in Chapter 3 additionally facilitate modeling stateful aspects of behavior tree nodes and enable the investigation of alternative transformation approaches. Furthermore, the topic of behavior trees is equally interesting from the perspective of synchronous languages, since the synchronous reactive model fits well into the notion of behavior trees.

6.3 Future Work on SCCharts

SCCharts have seen active development over the years and established themself as a reliable synchronous modeling language. With the introduction of timed and OO modeling features there are new directions for further research and some proposed concepts leave room for improvement. In combination with LF, there are new opportunities to explore in the future.

6.3.1 Distributed SCCharts using Lingua Franca

At the moment, the code generation capabilities of SCCharts do not provide a distributed deployment option. Hence, including aspects of LF could broaden the usability of SCCharts to this area. While it would be possible to adapt and reimplement the distributed infrastructure of LF for SCCharts, a simpler and more robust way could be the integration of LF into the downstream compilation of SCCharts. LF would act as a dedicated distributed coordination layer for concurrent SCCharts. In this regard, the dataflow notation in SCCharts, presented in Section 2.3.1, could act as a natural bridge between the two language.

Section 3.4.4 already illustrated a concept to embed SCCharts in LF. A similar approach could be used to deploy distributed SCCharts. An intermediate transformation in SCCharts would synthesize a federated LF

model from a dataflow SCChart that models a distributed system. Each SCCharts actor would be compiled into a separate tick function (cf. [Lüd21; Smy21]) that is invoked by a reaction in corresponding federates. The LF runtime infrastructure would then take care of the distributed deployment, communication, and execution of the SCCharts components.

6.3.2 Multiclocked SCCharts

The introduction of clocks to SCCharts in Chapter 4 primarily focuses on modeling real-time aspects. With period directives there is also a feature that is inspired by synchronous multiclocking. In this context, CCSL could be utilized to provide a formalization of program parts that are subject to a periodic execution. While this could cover real-time considerations, CCSL also offers means for logical relations between clock-bound entities, e. g., regions. One such example is the intended 3-to-1 ratio for the region pacing in Figure 4.15. In a preliminary experiment, an SCChart was automatically synthesized from a CCSL specification for a similar relation and could act as a watchdog for the pacing of two regions [SHM+18; SHM+20]. In this regard, one future direction for SCCharts could be the extension of discrete logical pacing mechanisms toward multiclocking. This would enable the binding of the periodic activation of certain program parts not only to physical time but to other subsystems.

6.3.3 Sleeping Programs

In the dynamic ticks experiment with the Furuta pendulum in Section 4.5.1, the response time analyses revealed that a notable delay was caused by the wake-up procedure. Subtracting the tick's execution time, the system's sleep function (`clock_nanosleep`) increased the response time by 172 usecs on average, with a minimum at 7 usecs and a maximum of 308 usecs. In contrast to that, the busy waiting implementation for the DS demo in Section 4.5.2 caused a smaller overhead. Admittedly, the used operating system had no real-time capabilities and imprecisions can be expected in such an environment. Real-time kernels with core isolation can yield average delays of about 1 usec for the same sleep function [AJH+23]. Yet,

such environments may not be always available, and the delays on the non-real-time system already impacted the sound signal of the pendulum by variations of up to one note, despite a relatively low octave.

While the busy waiting approach cancels out the efficiency benefits of dynamic ticks, it could be used to improve the implementation in both SCCharts and LF on non-real-time platforms. Anticipating a lag in the wake-up process, one could invoke the sleep function with a reduced sleep time, based on a fixed value or some learning strategy. If the sleep function would then return before the intended wake-up time, the remaining time could be bridged by a different mechanism, such as busy waiting. This may reduce the wake-up lag on regular systems, but this requires further investigation.

Additionally, one could investigate padding strategies that dynamically delay the outputs to compensate varying lags and execution times. This would facilitate producing outputs with specific interval, such as the sound signal. Precision timed architectures [EL07] could further facilitate such strategies.

The experiments with dynamic ticks also illustrate that a sparse execution with explicit sleep or idle periods can drastically reduce the computational load of a program. It might be worthwhile to additionally investigate the resulting energy consumption. This especially, since some platforms provide software controlled power saving modes, such as the `LowPower.deepSleep`[2] functionality on some Arduino boards. However, this might intensify the problem of wake-up lag.

6.3.4 Refining Object Orientation

As discussed in Section 5.5, some OO feature for SCCharts were conservatively restricted to focus on the investigation of language design aspects and to create a safe subset for the deterministic semantics of SCCharts. The proposed approach lays a foundation for future development and further refinement of OO concepts in SCCharts and synchronous languages.

References One limitation in the proposed OO language extension for SCCharts is the restriction to constant references for objects. This facilitates

[2] `https://www.arduino.cc/reference/en/libraries/arduino-low-power/lowpower.deepsleep/`

static analysis and compilation but also prevents passing around objects at runtime, which is quite common in general purpose OO languages. One direction for future work is to lift this restriction. Rust, discussed in Section 5.1.1, illustrates a model that uses reference lifetimes and ownership *borrowing* [Pea21] to address this issue statically. The same mechanism is used in Blech to restrict mutable references.

With subtyping there is another dimension to this problem, since references can hold subtype objects that have different behavior, and might access data differently, affecting causality. Here SPs could be used to ensure a schedule independent of subtypes. A subtype object that is policy-coherenent to its supertype's policy could be used as a replacement for its supertype without the need to adjust the policy-conformant scheduling. A detailed sketch of this approach can be found in the journal publication on OO in SCCharts [SSM21].

An alternative approach can be found in Céu, mentioned in Section 5.1.2, where concurrent intra-instant communication is prohibited and threads are scheduled in lexical order. While this can be considered a harsh restriction, the idea of adding constraints on concurrency can help in reducing the complexity of the problem of causality. For example, reactors have no shared state variables, only instantaneous events between reactors. The *lean* state-based compilation approach for SCCharts by Smyth [Smy21] uses a region-based scheduling granularity that limits interleaving and drops support for instantaneous back and forth communication, similar to LF.

Behavioral Subtyping Section 5.3.3 presented different levels of subtype admissibility. While the proposed implementation with its white-box scheduling does not require extensive subtyping restrictions and explicitly refrains from creating built-in limitations for SCCharts that go beyond minimal type safety, it is certainly relevant to further investigate means to express advanced aspects of subtyping. As already mentioned, SPs can be used to retain a scheduling interface in the face of type inheritance. A more extensive approach comes in the form of behavioral subtyping. There is already research in the context of Harel's OO statecharts [HK02; SSL19] that could be transferred to SCCharts. Lee and Xiong present a behavioral type system for component-based designs that uses extended interface

automata [LX04]. Furthermore, the existing model checking capabilities in SCCharts [Sta19] could be utilized to introduce pre- and post-conditions for methods and regions that must be fulfilled by subtype implementations.

Formal Semantics While the focus of this thesis lies on language design and pragmatics-aware modeling with a proof of concept implementation, future research for OO in SCCharts may include formal models for the proposed concepts. This could involve a dedicated formalization of SCCharts themself because their current semantics is grounded via model transformations [Mot17] in the SC semantics of the SCG/SCL. Alternatively, the SC kernel language could be extended to capture aspects of OO. With the Sequentially Constructive Procedural Language (SCPL), Gretz et al. made a first step in this direction when they created an extension to formalize the semantics of Blech [GGM+20; GGM+22]. Furthermore, there is a detailed theory on type systems and subtyping [Pie02; Car88], as well as machine-checked proofs of type soundness in languages such as Java [KN06] and C++ [WNS+06], that could be adapted to express aspects of type inheritance in SCCharts.

6.3.5 Object-Oriented State-Based Code Generation

Section 5.3 used a high-level transformation approach for the proposed OO features. In addition to that, the new OO features provide an opportunity to create dedicated OO code generation approaches that do not remove aspects of OO but transfer them into the host language, if supported. The state-based code generation approach for SCCharts [SMH18; Smy21] is particularly well suited for such a concept. It is designed to directly represent the stateful structure of SCCharts in code, e. g., by synthesizing regions and states into separate functions, and enumerations and switch statements to encode state machines.

An OO state-based approach could modularize the code into separate classes and express instantiation and inheritance directly at this level. Listing 6.1 illustrates a mockup in Java using the `CountingCounterApplication` in Figure 5.4b and the `CountingCounter` in Figure 5.7a. Note that the code omits some aspects for brevity, such as the handling of region termination, which is irrelevant in this example, the transitions of the `Wait` and `Reset` states, and

```
1  public class CountingCounterApplication {
2   enum RegionApplicationStates { Wait, Reset }
3
4   private CountingCounter counter;
5   private RegionApplicationStates
          regionApplicationActiveState;
6
7   public CountingCounterApplication() {
8    reset();
9   }
10
11  public void reset() {
12   counter = new CountingCounter();
13   regionApplicationActiveState =
          RegionApplicationStates.Wait;
14  }
15  public void tick() {
16   counter.regionCounting();
17   regionApplication();
18  }
19
20  private void regionApplication() {
21   switch (regionApplicationActiveState) {
22    case Wait: regionApplication_Wait(); break;
23    case Reset: regionApplication_Reset(); break;
24   }
25  }
26  private void regionApplication_Wait() {
27   // Check transition and set next state
28  }
29  private void regionApplication_Reset() {
30   // Check transition and set next state
31  }
32 }
```

(a) CountingCounterApplication

```
1  public class CountingCounter
          extends Counter {
2   private boolean
          regionCountingDelayEnabled = false;
3
4   public void reset() {
5    value = 0;
6    regionCountingDelayEnabled = false;
7   }
8   public void regionCounting() {
9    regionCounting_Counting();
10  }
11  private void regionCounting_Counting() {
12   if (regionCountingDelayEnabled) {
13    increment();
14   }
15   regionCountingDelayEnabled = true;
16  }
17 }
```

(b) CountingCounter

```
1  public class Counter {
2   protected int value = 0;
3
4   public void increment() {
5    value++;
6   }
7   public void decrement() {
8    value--;
9   }
10  public int getValue() {
11   return value;
12  }
13 }
```

(c) Counter

Listing 6.1. A mockup of the code structure generated by an OO state-based approach in Java for the CountingCounterApplication in Figure 5.4b and CountingCounter with inheritance.

optimizes the Counting region, which only has one state. In contrast to the object instantiation via macro expansion, discussed in Section 5.3.1, this approach keeps SCCharts-based classes as classes including their regions. This is the case for the CountingCounter in Listing 6.1b that, in this example, also utilizes Java's inheritance mechanism to extend the Counter class in Listing 6.1c. The CountingCounter is then kept as an object reference in the CountingCounterApplication, see lines 4 and 12 of Listing 6.1a.

The CountingCounterApplication as the main program provides a tick function interface that invokes the Counting regions of the counter in line 16 and the local Application region in line 17. This design follows the lean state-based approach that treats regions as atomic scheduling units. Additionally, it represents a refinement of the tick function modularity approach, proposed by Smyth [Smy21; Lüd21]. Assuming scheduling information about each region in an SCChart, e. g., in the form of an SP or a causality interface, the container SCChart could invoke regions individually and schedule its own regions in between. This would permit accepting more programs, since this approach is more fine-grained than the one invoking tick functions of submodules.

Still, with regions as atomic scheduling units, the approach does not support interleaving of regions, which is required in the presence of instantaneous back and forth communication. While this is a justified design decision for the lean state-based approach, it means that the given CountingCounterApplication SCChart cannot be compiled with this code generator. The IURP prescribes that reset must be scheduled before the increment in region Counting, while getValue must be ordered after this update, which is impossible if region Application is not split up into separate scheduling units. The code in Listing 6.1 actually assumes a CountingCounter with a custom update-before-reset scheduling, discussed in the context of clocks in Section 4.2.2. This allows and prescribes the scheduling of the Counting region (line 16) before the Application region (line 17). To mitigate such a limitation, future development could involve techniques to automatically divide regions into subunits that enable interleaved scheduling, e. g., by Pouzet and Raymond [PR10]. This would also benefit methods in SCCharts, which are limited in the same way. Without inlining, they need to scheduled atomically, which rules out interleaving at a statement level.

6.4 Closing Remarks

This thesis discussed various aspects of language design for reactive systems and illustrated the proposed solutions directly in SCCharts and LF. Now that the contributions of my work have been presented and discussed, I would like to close this thesis with some personal remarks and observations adjacent to these topics.

Section 2.5 already sketched the idea of pragmatics-aware modeling and automatically generated views. However, it deserves another mentioning that this approach is a significant factor in working with SCCharts and LF. Especially the interactivity plays a crucial role, and although the many figures in this thesis are a result of these diagrams, a written document cannot adequately convey the live experience. Moreover, the fact that this approach represents the model in both a textual notation, for editing, and a graphical notation, for perception with an adjustable degree of abstraction, is (for now) a rare characteristic by itself. This thesis capitalizes on the availability of this approach and uses the unique opportunity to set different emphases in the notations for defining and visualizing certain language constructs in SCCharts and LF. Fortunately, my work can build upon years of development from previous Ph.D. students that laid the foundations and provided the frameworks for automatic layout, diagram synthesis, and interactive visualization. Likewise, open source solutions, such as Xtext, are a cornerstone of tools such as KIELER or the LF editor. The combination of these technologies not only enables crafting useful tools but also allows rapid prototyping, which has proven itself valuable to me many times. When meeting with members of the LF team for the first time, we were able to quickly create the first diagrams and try out different graphical styles.

Another aspect that I would like to emphasize is the fact that both LF and SCCharts (and KIELER in general) are developed and maintained as feature-rich and lasting tools, instead of being just disposable prototypes demonstrating a single concept. They are consistently reevaluated in terms of their user experience and are actively integrated in teaching and industrial collaboration. From my perspective, this has the benefit of providing a sense of lasting purpose, which motivates creating high quality and usable solutions. At the same time, it requires a lot of effort to orga-

nize and maintain such projects. When I joined the KIELER team in 2013 in the context of my Bachelor's thesis, it was already several years in use and had a large codebase. It always took a team effort to maintain and advance this project. And while such an endeavor provides valuable experiences in software engineering and project management, the time spent on maintaining and sometimes restructuring the codebase itself rarely directly results in academic publications on that particular topic. In my experience, it requires a careful balance between innovative research and maintenance of the surrounded tooling. Additionally, it relies on the endorsement and support from the academic advisory side, which I am glad to have received plentifully. While LF is a younger project compared to KIELER, it likewise carries the same aspiration for creating a lasting usable tool, maybe also in the spirit of its predecessor Ptolemy.

Finally, I hope the concepts presented in this thesis provide a valuable perspective on the topic of language design and maybe inspire new or further refined approaches in the future. Equally, I hope that, also in the long run, the extensions to the LF and the SCCharts language will support users in modeling reactive systems.

Acknowledgments

I want to thank my advisor, Reinhard von Hanxleden, for giving me the opportunity to write my thesis at his group and under his experienced supervision. Working with him on the subjects of this thesis and various topics beyond was a great and invaluable experience to me. I am very grateful for his support, his patience, and the freedom and trust he has given me. He has been a steady source of motivation and opportunities.

My thanks also go to Michael Mendler for the productive and inspiring collaboration on various topics and papers. His knowledge and experience, especially in the field of synchronous semantics, always provided a valuable perspective and a great source of advice.

I thank Edward A. Lee for welcoming me so warmly into the LF team. Working with him and such an international and global team was an honor. His expertise and openness helped me in learning the subtleties of LF and in contributing to it myself, ultimately resulting in this thesis covering research questions regarding both SCCharts and LF.

In the LF team, my thanks go to Marten Lohstroh in particular. After our initial meeting at the FDL in Southampton, which was the birthplace of the first LF diagrams, he cordially integrated me into the team, resulting in many productive and interesting meetings, as well as a joint effort for the LF product.

Likewise, I want to thank my colleagues and predecessors in the KIELER team. Their software and research on layout, pragmatics, and SCCharts laid the foundation for much of my work in this thesis. It was a great pleasure to be part of such a dedicated team.

Special thanks go to my colleague, Steven Smyth, for the countless hours we spent together discussing ideas in front of whiteboards, collaborating on research, and programming for the KIELER project. Working with him will always remain a core memory of my time at the group. I would also like to thank Steven and Marten for their very valuable proofreading of my thesis.

Furthermore, I thank my colleagues, friends, and family who accompanied me on my academic journey and helped me in keeping my personal life at balance.

Finally, I would like to express my deepest thanks to my parents, Gudrun and Hans-Werner Schulz-Rosengarten. I will be forever grateful for their unlimited patience and unwavering support that made my academic endeavor and ultimately this thesis possible.

Bibliography

[ABP+97] Charles André, Frédéric Boulanger, Marie-Agnès Péraldi, Jean-Paul Rigault, and Guy Vidal-Naquet. "Objects and synchronous programming". In: *RAIRO-APII-JESA-Journal Europeen des Systemes Automatises* 31.3 (1997), pp. 417–432.

[Abr96] Jean-Raymond Abrial. "Steam-boiler control specification problem". In: *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Berlin, Heidelberg: Springer, 1996, pp. 500–509. ISBN: 978-3-540-49566-6. DOI: 10.1007/BFb0027252.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "The algorithmic analysis of hybrid systems". In: *Theoretical Computer Science* 138.1 (1995), pp. 3–34. DOI: 10.1016/0304-3975(94)00202-T.

[AD21] Joaquín Aguado and Alejandra Duenas. "Synchronised shared memory and model checking: a proof of concept". In: *24th Forum on specification and Design Languages, FDL'21*. Antibes, France: IEEE, Sept. 2021, pp. 01–08. DOI: 10.1109/FDL53530.2021.9568373.

[AD94] Rajeev Alur and David L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[Ada16] AdaCore. *High-integrity object-oriented programming in Ada, v1.4*. Oct. 2016. URL: https://www.adacore.com/uploads/techPapers/HighIntegrityAda.pdf.

[Agh86] Gul Agha. "An overview of actor languages". In: *Proceedings of the 1986 SIGPLAN Workshop on Object-Oriented Programming,*

OOPWORK 1986. Yorktown Heights, New York, USA: ACM, 1986, pp. 58–67. DOI: 10.1145/323779.323743.

[AH01] Luca de Alfaro and Thomas A. Henzinger. "Interface automata". In: *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001*. Vienna, Austria: ACM, 2001, pp. 109–120. DOI: 10.1145/503209.503226.

[AJH+23] Henrik Austad, Erling Rennemo Jellum, Sverre Hendseth, Geir Mathisen, Torleiv Håland Bryne, Kristoffer Nyborg Gregertsen, Sigurd Mørkved Albrektsen, and Bjarne Emil Helvik. "Composable distributed real-time systems with deterministic network channels". In: *Journal of Systems Architecture* 137.C (Apr. 2023). DOI: 10.1016/j.sysarc.2023.102853.

[Ald13] Jonathan Aldrich. "The power of interoperability: why objects are inevitable". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 101–116. ISBN: 9781450324724. DOI: 10.1145/2509578.2514738.

[All70] Frances E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479.

[AM09] Charles André and Frédéric Mallet. "Specification and verification of time requirements with CCSL and Esterel". In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '09. Dublin, Ireland: ACM, 2009, pp. 167–176. ISBN: 9781605583563. DOI: 10.1145/1542452.1542475.

[AMP+18] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. "Deterministic concurrency: A clock-synchronised shared memory approach". In: *27th European Symposium on Programming, ESOP'18*. Thessa-

loniki, Greece, Apr. 2018, pp. 86–113. DOI: 10.1007/978-3-319-89884-1_4.

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

[And09] Charles André. *Syntax and semantics of the clock constraint specification language (CCSL)*. Tech. rep. RR-6925. INRIA, 2009, p. 37. URL: https://hal.inria.fr/inria-00384077.

[And19] Lewe Andersen. "Dataflow and statemachine extraction from C/C++ code". Master's thesis. Kiel University, Department of Computer Science, Dec. 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-mt.pdf.

[Asp19] Aspencore. *2019 embedded markets study*. EETimes (embedded.com). Mar. 2019. URL: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[ASU07] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — principles, techniques, and tools*. Addison-Wesley, 2007, p. 1009. ISBN: 0-321-48681-1.

[AT05] Karine Altisen and Stavros Tripakis. "Implementation of timed automata: An issue of semantics or modeling?" In: *Formal Modeling and Analysis of Timed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 273–288. DOI: 10.1007/11603009_21.

[BBB+10] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. "Statistical abstraction and model-checking of large heterogeneous systems". In: *Formal Techniques for Distributed Systems*. Vol. 6117. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 32–46. ISBN: 978-3-642-13464-7. DOI: 10.1007/978-3-642-13464-7_4.

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "A formally verified compiler for Lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 586–601. ISBN: 9781450349888. DOI: 10.1145/3062341.3062358.

[BCC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. "A survey on reactive programming". In: *ACM Computing Surveys* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[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. DOI: 10.1109/JPROC.2002.805826.

[BCH+08] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '08. Tucson, AZ, USA: ACM, 2008, pp. 121–130. DOI: 10.1145/1375657.1375674.

[BCP+15] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "A synchronous-based code generator for explicit hybrid systems languages". In: *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015*. Vol. 9031. Lecture Notes in Computer Science. London, UK: Springer, 2015, pp. 69–88. DOI: 10.1007/978-3-662-46663-6_4.

[BCR12] Albert Benveniste, Benoît Caillaud, and Jean-Baptiste Raclet. "Application of interface theories to the separate compilation of synchronous programs". In: *Proc. of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA*. IEEE, Dec. 2012, pp. 7252–7258. DOI: 10.1109/CDC.2012.6426437.

[BDS96] Frédéric Boussinot, Guillaume Doumenc, and Jean-Bernard Stefani. "Reactive objects". In: *Annales Des Télécommunications* 51.9 (Sept. 1996), pp. 459–473. DOI: 10.1007/BF02997708.

[Ber00] Gérard Berry. "The foundations of Esterel". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5. DOI: 10.7551/mitpress/5641.003.0021.

[Ber93] Gérard Berry. "Preemption in concurrent systems". In: *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings*. Vol. 761. Lecture Notes in Computer Science. Springer, 1993, pp. 72–93. DOI: 10.1007/3-540-57529-4_44.

[Ber99] Gérard Berry. *The Esterel v5 language primer*. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps. 1999.

[Bou06] Frédéric Boussinot. "Fairthreads: mixing cooperative and preemptive threads in C". In: *Concurrency and Computation: Practice and Experience* 18.5 (Apr. 2006), pp. 445–469. DOI: 10.1002/cpe.919.

[Bou09] Timothy Bourke. "Modelling and programming embedded controllers with timed automata and synchronous languages". PhD thesis. University of NSW, Sydney, 2009.

[BP13] Timothy Bourke and Marc Pouzet. "Zélus: a synchronous language with ODEs". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013*. Philadelphia, PA, USA, Apr. 2013, pp. 113–118. DOI: 10.1145/2461328.2461348.

[BPS04] Reinhard Budde, Axel Poigné, and Karl-Heinz Sylla. "synERJY an object-oriented synchronous language". In: Electronic Notes in Theoretical Computer Science 153.4 (2004). Ed. by Florence Maraninchi, Alain Girault, and Marc Pouzet, pp. 99–115. DOI: 10.1016/j.entcs.2006.02.026.

[BS01] Gérard Berry and Ellen Sentovich. "Multiclock Esterel". In: *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001*. Vol. 2144. Lecture Notes in Computer Science. Livingston, Scotland, UK: Springer, 2001, pp. 110–125. DOI: 10.1007/3-540-44798-9_10.

[BS09] Timothy Bourke and Arcot Sowmya. "Delays in Esterel". In: *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*. Dagstuhl Seminar Proceedings 09481. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 11 2009.

[BSH20a] Andreas Boysen, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "A hard real time demo for dynamic ticks and timed SCCharts". In: *MBMV 2020 — Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, GMM/ITG/GI-Workshop, GMM-Fachbericht 96*. Stuttgart, Germany, Mar. 2020, pp. 61–64.

[BSH20b] Andreas Boysen, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "A hard real time demonstrator for dynamic ticks and timed SCCharts". In: *Proc. Forum on Specification and Design Languages (FDL '20)*. Kiel, Germany: IEEE, Sept. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232943.

[BSH20c] Andreas Boysen, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *An FPGA-based Demonstrator for Dynamic Ticks*. Technical Report 2001. ISSN 2192-6247. Kiel University, Department of Computer Science, July 2020.

[BW96] Robert Büssow and Matthias Weber. "A steam-boiler control specification with Statecharts and Z". In: *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 109–128. ISBN: 3540619291. DOI: 10.1007/BFb0027233.

[Cap03] Luiz Fernando Capretz. "A brief history of the object-oriented approach". In: *SIGSOFT Software Engineering Notes* 28.2 (Mar. 2003), p. 6. ISSN: 0163-5948. DOI: 10.1145/638750.638778.

[Car88] Luca Cardelli. "A semantics of multiple inheritance". In: *Information and Computation* 76.2/3 (Feb. 1988), pp. 138–164. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90007-7.

[CCG+09] Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. "Synchronous objects with scheduling policies: introducing safe shared memory in Lustre". In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES'09)*. Dublin, Ireland: ACM, June 2009, pp. 11–20. DOI: 10.1145/1542452.1542455.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. "From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications". In: *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*. LCTES'03. San Diego, California, USA: ACM, 2003, pp. 153–162. ISBN: 1581136471. DOI: 10.1145/780732.780754.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. "Type inference for late binding: the SmallEiffel compiler". In: *Modular Programming Languages*. Vol. 1204. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 67–81. ISBN: 978-3-540-68328-5. DOI: 10.1007/3-540-62599-2_31.

[Cha84] Daniel M. Chapiro. "Globally-asynchronous locally-synchronous systems". PhD thesis. Stanford University, CA, Dept. of Computer Science, 1984.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. "Introducing Objectcharts or how to use Statecharts in object-oriented design". In: *IEEE Transactions on Software Engineering* 18.1 (Jan. 1992), pp. 8–18. ISSN: 0098-5589. DOI: 10.1109/32.120312.

[CHC89] William R. Cook, Walter Hill, and Peter S. Canning. "Inheritance is not subtyping". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-*

guages. POPL'90. San Francisco, California, USA: ACM, 1989, pp. 125–135. ISBN: 0897913434. DOI: 10.1145/96709.96721.

[CHP06] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "Mixing signals and modes in synchronous data-flow systems". In: *Proc. 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006*. Seoul, Korea: ACM, 2006, pp. 73–82. DOI: 10.1145/1176887.1176899.

[CLB+19] Fabio Cremona, Marten Lohstroh, David Broman, Edward A Lee, Michael Masin, and Stavros Tripakis. "Hybrid co-simulation: it's about time". In: *Software & Systems Modeling* 18.3 (2019), pp. 1655–1679. DOI: 10.1007/s10270-017-0633-6.

[CÖ14] Michele Colledanchise and Petter Ögren. "How behavior trees modularize robustness and safety in hybrid systems". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*. IEEE, 2014, pp. 1482–1488. DOI: 10.1109/IROS.2014.6942752.

[CÖ17] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: an introduction*. CRC Press, 2017. ISBN: 978-1-138-59373-2. DOI: 10.1201/9780429489105.

[Coo09] William R. Cook. "On understanding data abstraction, revisited". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 557–572. DOI: 10.1145/1640089.1640133.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *Proc. of the 5th ACM International Conference On Embedded Software, EMSOFT 2005*. Jersey City, NJ, USA: ACM, 2005, pp. 173–182. DOI: 10.1145/1086228.1086261.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "SCADE 6: A formal language for embedded critical software development (invited paper)". In: *11th International Symposium on*

Theoretical Aspects of Software Engineering TASE. Sophia Antipolis, France, Sept. 2017, pp. 1–11. DOI: 10.1109/TASE.2017.8285623.

[Cra07] Iain D. Craig. *Object-oriented programming languages: interpretation*. London: Springer-Verlag London, 2007. ISBN: 978-1-84628-773-2. DOI: 10.1007/978-1-84628-774-9.

[CW85] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys* 17.4 (Dec. 1985), pp. 471–523. DOI: 10.1145/6041.6042.

[DCB+21] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin, and Marten Lohstroh. "Debugging and verification tools for Lingua Franca in Gemoc studio". In: *Proc. Forum on Specification and Design Languages, FDL'21, Antibes, France, September 8-10, 2021*. IEEE, Sept. 2021, pp. 1–8. DOI: 10.1109/FDL53530.2021.9568383.

[DDR04] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. "Almost ASAP semantics: from timed models to timed implementations". In: *Hybrid Systems: Computation and Control*. Berlin, Heidelberg: Springer, 2004, pp. 296–310. DOI: 10.1007/978-3-540-24743-2_20.

[DO-11] DO-332. *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*. Standard. Washington, D.C., USA: Radio Technical Commission for Aeronautics, Dec. 2011.

[DO-12] DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. Standard. Washington, D.C., USA: Radio Technical Commission for Aeronautics, Jan. 2012.

[Dom18] Sören Domrös. "Moving model-driven engineering from Eclipse to web technologies". Master's thesis. Kiel University, Department of Computer Science, Nov. 2018. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf.

[Dvo09] Daniel Dvorak. "NASA Study on Flight Software Complexity". In: *AIAA Infotech@Aerospace Conference*. 2009.

[DZK+02] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. "Event-driven programming for robust software". In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 186–189. ISBN: 9781450378062. DOI: 10.1145/1133373.1133410.

[EB10] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA, 2010, pp. 307–309. ISBN: 9781450302401. DOI: 10.1145/1869542.1869625.

[EH20] Stephen A. Edwards and John Hui. "The sparse synchronous model". In: *Proc. Forum on Specification and Design Languages (FDL'20)*. Kiel, Germany, Sept. 2020. DOI: 10.1109/FDL50818.2020.9232938.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. "Taming heterogeneity—the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[EL07] S. A. Edwards and E. A. Lee. "The case for the Precision Timed (PRET) machine". In: *Proceedings of the 44th Design Automation Conference*. DAC '07. San Diego, CA, USA: ACM, June 2007, pp. 264–265. DOI: 10.1145/1278480.1278545.

[ELM+12] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit Seshia, and Jia Zou. "Distributed real-time software for cyber-physical systems". In: *Proceedings of the IEEE* 100.1 (Jan. 2012), pp. 45–59. DOI: 10.1109/JPROC.2011.2161237.

[Ess96] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. Dissertation. Eidgenössische Technische Hochschule [ETH] Zürich, 1996.

[Eum20] Philip Eumann. "Model-based debugging". Master's thesis. Kiel University, Department of Computer Science, June 2020. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-mt.pdf.

[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. Lecture Notes in Computer Science. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[FYK92] Katsuhisa Furuta, M. Yamakita, and S. Kobayashi. "Swing-up control of inverted pendulum using pseudo-state feedback". In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 206.4 (1992), pp. 263–269. DOI: 10.1243/PIME_PROC_1992_206_341_02.

[GG10] Abdoulaye Gamatié and Thierry Gautier. "The signal synchronous multiclock approach to the design of distributed embedded systems". In: *IEEE Trans. Parallel Distributed Syst.* 21.5 (2010), pp. 641–657. DOI: 10.1109/TPDS.2009.125.

[GG18] Friedrich Gretz and Franz-Josef Grosch. "Blech, imperative synchronous programming!" In: *Proc. Forum on Specification Design Languages (FDL'18)*. Sept. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524036.

[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. DOI: 10.1109/5.97301.

[GGM+20] Friedrich Gretz, Franz-Josef Grosch, Michael Mendler, and Stephan Scheele. "Synchronized shared memory and procedural abstraction: towards a formal semantics of Blech". In: *Proc. Forum on Specification and Design Languages (FDL'20)*. Kiel, Germany, Sept. 2020. DOI: 10.1109/FDL50818.2020.9232942.

[GGM+22] Friedrich Gretz, Franz-Josef Grosch, Michael Mendler, and Stephan Scheele. “Synchronized shared memory and black-box procedural abstraction: towards a formal semantics of Blech”. In: *ACM Transactions on Embedded Computing Systems* (Nov. 2022). Just Accepted. DOI: 10.1145/3571585.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 8 Edition*. Addison-Wesley Professional, 2015.

[GM10] Maurizio Gabbrielli and Simone Martini. *Programming languages: principles and paradigms*. London: Springer-Verlag London, 2010. ISBN: 978-1-84882-913-8. DOI: 10.1007/978-1-84882-914-5.

[Gri19] Lena Grimm. “From Lustre to graphical dataflow programs”. Master’s thesis. Kiel University, Department of Computer Science, May 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf.

[GSS+20] Lena Grimm, Steven Smyth, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, and Marc Pouzet. “From Lustre to graphical models and SCCharts”. In: *Proc. Forum on Specification and Design Languages (FDL ’20)*. Kiel, Germany, Sept. 2020. DOI: 10.1109/FDL50818.2020.9232944.

[GSS+22] Lena Grimm, Steven Smyth, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, and Marc Pouzet. “From Lustre to graphical models and SCCharts”. In: *ACM Transactions on Embedded Computing Systems* (July 2022). Just Accepted. DOI: 10.1145/3544973.

[GTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. “POLYCHRONY for system design”. In: *Journal of Circuits, Systems, and Computers* 12.3 (2003), pp. 261–304. DOI: 10.1142/S0218126603000763.

[Gur99] Corin A. Gurr. "Effective diagrammatic communication: syntactic, semantic and pragmatic issues". In: *Journal of Visual Languages & Computing* 10.4 (1999), pp. 317–342. DOI: 10.1006/jvlc.1999.0130.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993. DOI: 10.1007/978-1-4757-2231-4.

[Han09] Reinhard von Hanxleden. "SyncCharts in C—a proposal for light-weight, deterministic concurrency". In: *Proceedings of the 9th ACM & IEEE International conference on Embedded software (EMSOFT'09)*. Grenoble, France: ACM, Oct. 2009, pp. 225–234. DOI: 10.1145/1629335.1629366.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.

[HBG17] Reinhard von Hanxleden, Timothy Bourke, and Alain Girault. "Real-time ticks for synchronous programming". In: *Proc. Forum on Specification and Design Languages (FDL '17)*. Verona, Italy: IEEE, Sept. 2017, pp. 1–8. DOI: 10.1109/FDL.2017.8303893.

[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. DOI: 10.1109/5.97300.

[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. Kiel University, 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. DOI: 10.1145/2594291.2594310.

[HE22] John Hui and Stephen A. Edwards. "The sparse synchronous model on real hardware". In: *ACM Transactions on Embedded Computing Systems* (Dec. 2022). Just Accepted. DOI: 10.1145/3572920.

[Hew77] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial Intelligence* 8.3 (1977), pp. 323–364. DOI: 10.1016/0004-3702(77)90033-9.

[HG96] David Harel and Eran Gery. "Executable object modeling with statecharts". In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE '96. Berlin, Germany: IEEE Computer Society, 1996, pp. 246–257. ISBN: 0-8186-7246-3. DOI: 10.1109/2.596624.

[HHK03] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. "Giotto: a time-triggered language for embedded programming". In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99. DOI: 10.1109/JPROC.2002.805825.

[HHN01] Michael Hanus, Frank Huch, and Philipp Niederau. "ObjectCurry: an object-oriented extension of the declarative multi-paradigm language Curry". In: *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 89–106. DOI: 10.1007/3-540-45361-X_6.

[HK02] David Harel and Orna Kupferman. "On object systems and behavioral inheritance". In: *IEEE Transactions on Software Engineering* 28.9 (2002), pp. 889–903. DOI: 10.1109/TSE.2002.1033228.

[HK04] David Harel and Hillel Kugler. "The Rhapsody semantics of Statecharts (or, on the executable core of the UML)". In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Vol. 3147. Lecture Notes in

Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 325–354. ISBN: 978-3-540-27863-4. DOI: 10.1007/978-3-540-27863-4_19.

[HLF+22] Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard. "Pragmatics twelve years later: a report on Lingua Franca". In: *11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Vol. 13702. Lecture Notes in Computer Science. Springer. Rhodes, Greece, Oct. 2022, pp. 60–89. DOI: 10.1007/978-3-031-19756-7_5.

[HN96] David Harel and Amnon Naamad. "The STATEMATE semantics of statecharts". In: *ACM Transactions on Software Engineering and Methodology* 5.4 (Oct. 1996), pp. 293–333. DOI: 10.1145/235321.235322.

[HNS+94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. "Symbolic model checking for real-time systems". In: *Information and Computation* 111.2 (1994), pp. 193–244. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1045.

[HP85] David Harel and Amir Pnueli. "On the development of reactive systems". In: *Logics and models of concurrent systems*. NATO ASI Series 13 (1985), pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17.

[HPB+99] Olivier Hainque, Laurent Pautet, Yann Le Biannic, and Eric Nassor. "Cronos: a separate compilation toolset for modular Esterel applications". In: *World Congress on Formal Methods*. Vol. 1709. Lecture Notes in Computer Science. Springer, Sept. 1999, pp. 1836–1853. DOI: 10.1007/3-540-48118-4_47.

[HR04] Grégoire Hamon and John Rushby. "An operational semantics for Stateflow". In: *Fundamental Approaches to Software Engineering (FASE)*. Vol. 2984. Lecture Notes in Computer Science. Barcelona, Spain: Springer, Apr. 2004, pp. 229–243. DOI: 10.1007/978-3-540-24721-0_17.

[ISO12] ISO/IEC 8652:2012. *Information technology — Programming languages — Ada*. Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 2012.

[ISO18] ISO 26262-6:2018. *Road vehicles — Functional safety — Part 6: Product development at the software level*. Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 2018.

[ISO20] ISO/IEC 14882:2020. *Programming languages — C++*. Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 2020.

[JJK+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages (POPL'18)* (Dec. 2017). DOI: 10.1145/3158154.

[JMO93] Martin Jourdan, Florence Maraninchi, and Alfredo Olivero. "Verifying quantitative real-time properties of synchronous programs". In: *Proc. Computer Aided Verification, 5th International Conference, CAV '93*. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 347–358. DOI: 10.1007/3-540-56922-7_29.

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

[KHA17] Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive design patterns*. Manning Publications Company, 2017.

[KN06] Gerwin Klein and Tobias Nipkow. "A machine-checked model for a Java-like language, virtual machine, and compiler". In: *ACM Transactions on Programming Languages and Systems* 28.4 (July 2006), pp. 619–695. ISSN: 0164-0925. DOI: 10.1145/1146809.1146811.

[Kod20] Jeffrey Kodosky. "Labview". In: *Proceedings of the ACM on Programming Languages* (June 2020). DOI: 10.1145/3386328.

[KS08] Hannes Kegel and Friedrich Steimann. "Systematically refactoring inheritance to delegation in Java". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. New York, NY, USA: ACM, 2008, pp. 138–146.

[KS12] Christoph M. Kirsch and Ana Sokolova. "The logical execution time paradigm". In: *Advances in Real-Time Systems*. Ed. by Samarjit Chakraborty and Jörg Eberspächer. Berlin, Heidelberg: Springer, 2012, pp. 103–120. ISBN: 978-3-642-24349-3. DOI: 10.1007/978-3-642-24349-3_5.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[LBL+21] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "Quantifying and generalizing the CAP theorem". In: *Computing Research Repository (CoRR)* (Sept. 2021). DOI: 10.48550/arXiv.2109.07771. arXiv: 2109.07771.

[LBL+23] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "Trading off consistency and availability in tiered heterogeneous distributed systems". In: *Intelligent Computing* 2 (2023). DOI: 10.34133/icomputing.0013.

[LBM+23] Marten Lohstroh, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A. Lee. "Deterministic coordination across multiple timelines". In: *ACM Transactions on Embedded Computing Systems* (Oct. 2023). Just Accepted. DOI: 10.1145/3615357.

[Lee03] Edward A. Lee. *Model-driven development – From object-oriented design to actor-oriented design*. Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop), Chicago. Sept. 2003.

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

[Lee08] Edward A. Lee. "Cyber physical systems: design challenges". In: *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[LEJ+02] Jie Liu, Johan Eker, Jörn W. Janneck, and Edward A. Lee. "Realistic simulations of embedded control systems". In: *IFAC Proceedings Volumes, 15th IFAC World Congress*. Vol. 35. 1. Elsevier, 2002, pp. 391–396. DOI: 10.3182/20020721-6-ES-1901.00553.

[LÍG+19] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. "Reactors: a deterministic model for composable reactive systems". In: *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*. Vol. 11971. Lecture Notes in Computer Science. Springer, 2019, pp. 59–85. DOI: 10.1007/978-3-030-41131-2_4.

[LL21] Edward A. Lee and Marten Lohstroh. "Time for all programs, not just real-time programs". In: *Proc. of 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021*. Vol. 13036. Lecture Notes in Computer Science. Springer, 2021, pp. 213–232. ISBN: 978-3-030-89159-6. DOI: 10.1007/978-3-030-89159-6_15.

[LLN09] Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. "Classes and inheritance in actor-oriented design". In: *ACM Transactions on Embedded Computing Systems* 8.4 (July 2009). ISSN: 1539-9087. DOI: 10.1145/1550987.1550992.

[LMB+21] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. "Toward a lingua franca for deterministic concurrent systems". In: *ACM Transactions on Embedded Computing Systems* 20.4 (May 2021), Article 36. DOI: 10.1145/3448128.

[LML+23] Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. "Towards building verifiable CPS using Lin-

gua Franca". In: *ACM Transactions on Embedded Computing Systems* 22 (Sept. 2023). DOI: 10.1145/3609134.

[LMS+20] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. "A language for deterministic coordination across multiple timelines". In: *Proc. Forum on Specification and Design Languages (FDL '20)*. Kiel, Germany, Sept. 2020. DOI: 10.1109/FDL50818.2020.9232939.

[Loh20] Marten Lohstroh. "Reactors: a deterministic model of concurrent computation for reactive systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1 (1997), pp. 134–152. DOI: 10.1007/s100090050010.

[LS17] Edward A. Lee and Sanjit A. Seshia. *Introduction to embedded systems, a cyber-physical systems approach, second edition*. MIT Press, 2017. ISBN: 978-0-262-53381-2. URL: http://LeeSeshia.org.

[LSH+21] Daniel Lucas, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Friedrich Gretz, and Franz-Josef Grosch. "Extracting mode diagrams from Blech code". In: *Proc. Forum on Specification and Design Languages (FDL '21)*. Antibes, France, Sept. 2021. DOI: 10.1109/FDL53530.2021.9568375.

[LT10] Edward A. Lee and Stavros Tripakis. "Modal models in Ptolemy". In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*. Vol. 47. Linköping University Electronic Press, Linköping University, 2010, pp. 11–21.

[Luc20] Daniel Lucas. "Extraction of mode diagrams from Blech". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dalu-mt.pdf. Master's thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Apr. 2020.

[Lüd21] Gavin Lüdemann. "Modular code generation for SCCharts". Bachelor's thesis. Kiel University, Department of Computer Science, Sept. 2021. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/glu-bt.pdf.

[LW94] Barbara H. Liskov and Jeannette M. Wing. "A behavioral notion of subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.

[LX04] Edward A. Lee and Yuhong Xiong. "A behavioral type system and its application in Ptolemy II". In: *Formal Aspects of Computing* 16.3 (2004), pp. 210–237. DOI: 10.1007/s00165-004-0043-8.

[Mar92] Florence Maraninchi. "Operational and compositional semantics of synchronous automaton compositions". In: *Third International Conference on Concurrency Theory (CONCUR'92)*. Vol. 630. Lecture Notes in Computer Science. Springer, 1992, pp. 550–564. DOI: 10.1007/BFb0084815.

[MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. "Programming deterministic reactive systems with Synchronous Java". In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Invited Paper. Paderborn, Germany, June 2013. DOI: 10.1109/ISORC.2013.6913222.

[Mil89] Robin Milner. *Communication and concurrency*. USA: Prentice Hall, 1989, p. 260. ISBN: 978-0-13-114984-7.

[MIS08] MISRA. *Guidelines for the Use of the C++ Language in Critical Systems*. Standard. ISBN 978-906400-03-3. Warwickshire, United Kingdom: Motor Industry Software Reliability Association (MISRA), 2008.

[MLB+23] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jerónimo Castrillón, and Edward A. Lee. "High-performance deterministic concurrency using Lingua Franca". In: *Computing*

Research Repository (CoRR) (Jan. 2023). DOI: 10.48550/arXiv.2301.02444. arXiv: 2301.02444.

[Mot17] Christian Motika. *SCCharts—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation. Kiel University, Faculty of Engineering. Department of Computer Science, 2017.

[MP93] Zohar Manna and Amir Pnueli. "Verifying hybrid systems". In: *Hybrid Systems*. Vol. 736. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1993, pp. 4–35. ISBN: 3540573186. DOI: 10.1007/3-540-57318-6_22.

[MR98] Florence Maraninchi and Yann Rémond. "Mode-automata: about modes and states for reactive systems". In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98*. Vol. 1381. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 185–199. DOI: 10.1007/BFb0053571.

[MS98] Leonid Mikhajlov and Emil Sekerinski. "A study of the fragile base class problem". In: *Proc. 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Vol. 1445. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 355–382. DOI: 10.1007/BFb0054099.

[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. Lecture Notes in Computer Science. Corfu, Greece, Oct. 2014, pp. 461–480. DOI: 10.1007/978-3-662-45234-9.

[Obj11] Object Management Group. *OMG unified modeling language (OMG UML), superstructure. version 2.4.1*. https://www.omg.org/spec/UML/2.4.1/Superstructure/PDF. Aug. 2011.

[OSY94] Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "Using abstractions for the verification of linear hybrid systems". In: *Proceedings of the 6th Annual Conference on Computer-Aided Verification*. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 81–94. DOI: 10.1007/3-540-58179-0_45.

[PCO19] André Pinho, Luis Couto, and José Oliveira. "Towards Rust for critical systems". In: *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036.

[Pea21] David J. Pearce. "A lightweight formalism for reference lifetimes and borrowing in Rust". In: *ACM Transactions on Programming Languages and Systems* 43.1 (Apr. 2021). DOI: 10.1145/3443420.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007. DOI: 10.1007/978-0-387-70628-3.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Jan. 2002.

[PLR95] Nancy Pennington, Adrienne Y. Lee, and Bob Rehder. "Cognitive activities and levels of abstraction in procedural and object-oriented design". In: *Human-Computer Interaction* 10.2-3 (1995), pp. 171–226. DOI: 10.1080/07370024.1995.9667217.

[PR10] Marc Pouzet and Pascal Raymond. "Modular static scheduling of synchronous data-flow networks—an efficient symbolic representation". In: *Design Automation for Embedded Systems* 14.3 (2010), pp. 165–192. DOI: 10.1007/s10617-010-9053-3.

[PRS+17] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. "Runtime enforcement of cyber-physical systems". In: *ACM Transactions on Embedded Computing Systems, Special Issue for ESWEEK/EMSOFT '17* 16.5s (2017), 178:1–178:25. DOI: 10.1145/3126500.

[PS91] Jens Palsberg and Michael I. Schwartzbach. "Object-oriented type inference". In: *Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. Phoenix, Arizona, USA: ACM, 1991, pp. 146–161. DOI: 10.1145/117954.117965.

[PTH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. "Synthesizing Safe State Machines from Esterel". In: *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '06)*. Ottawa, Canada: ACM, June 2006, pp. 113–124. DOI: 10.1145/1134650.1134667.

[Pto14] Claudius Ptolemaeus, ed. *System design, modeling, and simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[Ras21] Philip Raschkowski. "SCCharts for game development". Bachelor's thesis. Kiel University, Department of Computer Science, Mar. 2021. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/phr-bt.pdf.

[Ree79] Trygve Reenskaug. *Models – Views – Controllers*. Xerox PARC technical note. Dec. 1979.

[Ren18] Niklas Rentz. "Moving transient views from Eclipse to web technologies". Master's thesis. Kiel University, Department of Computer Science, Nov. 2018. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf.

[Rog11] James S. Rogers. "Language choice for safety critical applications". In: *Proceedings of the 2011 ACM Annual International Conference on Special Interest Group on the Ada Programming Language (SIGAda '11)*. Denver, Colorado, USA: ACM, 2011, pp. 81–90. DOI: 10.1145/2070337.2070363.

[RSA+21] Niklas Rentz, Steven Smyth, Lewe Andersen, and Reinhard von Hanxleden. "Extracting interactive actor-based dataflow models from legacy C code". In: *12th International Conference on Diagrammatic Representation and Inference (DIAGRAMS'21)*. Vol. 12909. Lecture Notes in Computer Science. Springer, Sept. 2021, pp. 361–377. DOI: 10.1007/978-3-030-86062-2_37.

[RSM+15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. "SCEst: Sequentially Constructive Esterel". In: *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '15)*. Austin, TX, USA, Sept. 2015. DOI: 10.1109/MEMCOD.2015.7340462.

[RV98] Didier Rémy and Jérôme Vouillon. "Objective ML: an effective object-oriented extension to ML". In: *Theory and Practice of Object Systems* 4.1 (1998), pp. 27–50. DOI: 10.1002/(SICI)1096-9942(1998)4:1%3C27::AID-TAPO3%3E3.0.CO;2-4.

[RWB16] Raymond Roestenburg, Rob Williams, and Robertus Bakker. *Akka in action*. Manning Publications Company, 2016.

[Sam08] Miro Samek. *Practical UML Statecharts in C/C++: event-driven programming for embedded systems*. CRC Press, 2008.

[San18] Francisco Sant'Anna. "Structured synchronous reactive programming for game development - case study: on rewriting Pingus from C++ to Céu". In: *17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2018)*. IEEE Computer Society, 2018, pp. 240–249. DOI: 10.1109/SBGAMES.2018.00036.

[Sch10] Klaus Schneider. *The synchronous programming language Quartz*. Internal Report. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, 2010.

[Sel93] Bran Selic. "An efficient object-oriented variation of the Statecharts formalism for distributed real-time systems". In: *Proceedings of the 11th IFIP WG10.2 International Conference Sponsored by IFIP WG10.2 and in Cooperation with IEEE COMPSOC on Computer Hardware Description Languages and Their Applications (CHDL '93)*. Vol. A-32. IFIP Transactions. NLD: North-Holland Publishing Co., 1993, pp. 335–344. ISBN: 0444816410. DOI: 10.1016/B978-0-444-81641-2.50030-7.

[SFR97] Manas Saksena, Paul Freedman, and Pawel Rodziewicz. "Guidelines for automated implementation of executable object oriented models for real-time embedded control systems". In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*. IEEE, 1997, pp. 240–251. DOI: 10.1109/REAL.1997.641286.

[SGA+17] Aminata Sabané, Yann-Gaël Guéhéneuc, Venera Arnaoudova, and Giuliano Antoniol. "Fragile base-class problem, problem?" In: *Empirical Software Engineering* 22.5 (Oct. 2017), pp. 2612–2657. DOI: 10.1007/s10664-016-9448-2.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994. ISBN: 0-471-59917-4.

[SHL+23a] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Soroush Bateni, and Edward A. Lee. "Modal reactors". In: *Computing Research Repository (CoRR)* (Jan. 2023). DOI: 10.48550/ARXIV.2301.09597. arXiv: 2301.09597.

[SHL+23b] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Soroush Bateni, and Edward A. Lee. "Polyglot modal models through Lingua Franca". In: *Proc. Design, Automation & Test in Europe Conference & Exhibition, DATE 2023*. Extended Abstract. Antwerp, Belgium: IEEE, 2023, pp. 1–2. DOI: 10.23919/DATE56975.2023.10136890.

[SHL+23c] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Edward A. Lee, and Soroush Bateni. "Polyglot modal models through Lingua Franca". In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week '23. San Antonio, TX, USA: ACM, 2023, pp. 337–342. DOI: 10.1145/3576914.3587498.

[SHM+18] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. "Time in SCCharts". In: *Proc. Forum on Specification and Design Languages (FDL '18)*. Munich, Germany, Sept. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524111.

[SHM+20] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. "Time in SCCharts". In: *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2018*. Ed. by Tom J. Kazmierski, Sebastian Steinhorst, and Daniel Große. Springer, 2020, pp. 1–25. ISBN: 978-3-030-31585-6. DOI: 10.1007/978-3-030-31585-6_1. Reproduced with permission from Springer Nature.

[SIL+17] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. "The design and implementation of the synchronous language CÉU". In: *ACM Transactions on Embedded Computing Systems* 16.4 (July 2017), 98:1–98:26. DOI: 10.1145/3035544.

[SIL15] Francisco Sant'Anna, Roberto Ierusalimschy, and Noemi de La Rocque Rodriguez. "Structured synchronous reactive programming with Céu". In: *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015*. Fort Collins, CO, USA: ACM, 2015, pp. 29–40. DOI: 10.1145/2724525.2724571.

[SL02] Stéphane S. Somé and Timothy C. Lethbridge. "Enhancing program comprehension with recovered state models". In: *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 2002, pp. 85–93. DOI: 10.1109/WPC.2002.1021325.

[SMC95] Sowmitri Swamy, Arthur Molin, and Burton M. Covnot. "OO-VHDL: object-oriented extensions to VHDL". In: *Computer* 28.10 (1995), pp. 18–26. DOI: 10.1109/2.467587.

[SMH18] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. "Synthesizing manually verifiable code for statecharts". In: *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*. Boston, MA, USA, Nov. 2018. DOI: 10.1145/3281278.3281283.

[SMR+17] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. "SCEst: Sequentially Constructive Esterel". In: *ACM Transactions on Embedded Computing Systems—Special Issue on MEMOCODE 2015* 17.2 (Dec. 2017), 33:1–33:26. DOI: doi.org/10.1145/3063129.

[SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SCCharts: the railway project report*. Technical Report 1510. ISSN 2192-6247. Kiel University, Department of Computer Science, Aug. 2015.

[Smy21] Steven Smyth. *Interactive model-based compilation — a modeller-driven development approach*. Kiel Computer Science Series 20 21/1. Dissertation. Kiel University, Faculty of Engineering. Department of Computer Science, 2021. DOI: 10.21941/kcss/2021/1.

[Spi89] John Michael Spivey. *Understanding Z - a specification language and its formal semantics (reprint)*. Vol. 3. Cambridge tracts in theoretical computer science. Cambridge University Press, 1989. ISBN: 978-0-521-33429-7.

[SQK18] Wasim Said, Jochen Quante, and Rainer Koschke. "On state machine mining from embedded control software". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 138–148. DOI: 10.1109/ICSME.2018.00024.

[SSH+18a] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. *A sequentially constructive circuit semantics for Esterel*. Technical Report 1801. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2018.

[SSH+18b] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. "On reconciling concurrency, sequentiality and determinacy for reactive systems — a sequentially constructive circuit semantics for Esterel". In: *18th International Conference on Application of Concurrency to System*

Design (ACSD'18). June 2018, pp. 95–104. DOI: 10.1109/ACSD.2018.00018.

[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: IEEE, 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.

[SSH18a] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Guidance in model-based compilations". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '18), Doctoral Symposium*. Vol. 78. Electronic Communications of the EASST. Limassol, Cyprus, Nov. 2018. DOI: 10.1007/978-3-030-03418-4_15.

[SSH18b] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Practical causality handling for synchronous languages*. Technical Report 1808. ISSN 2192-6247. Kiel University, Department of Computer Science, Dec. 2018.

[SSH18c] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Towards interactive compilation models". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Vol. 11244. Lecture Notes in Computer Science. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260. DOI: 10.14279/tuj.eceasst.78.1098.

[SSH19] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Practical causality handling for synchronous languages". In: *Proc. Design, Automation and Test in Europe Conference (DATE '19)*. Florence, Italy: IEEE, Mar. 2019. DOI: 10.23919/DATE.2019.8715081.

[SSL19] Eugene Syriani, Vasco Sousa, and Levi Lúcio. "Structure and behavior preserving statecharts refinements". In: *Science of Computer Programming* 170 (2019), pp. 45–79. DOI: 10.1016/j.scico.2018.10.005.

[SSM19] Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. "Towards object-oriented modeling in SCCharts". In: *Proc. Forum on Specification and Design Languages (FDL '19)*. Southampton, UK, Sept. 2019, pp. 1–8. DOI: 10.1145/3453482.

[SSM21] Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. "Toward object-oriented modeling in SCCharts". In: *ACM Transactions on Embedded Computing Systems* 20.4 (May 2021). DOI: 10.1145/3453482.

[Sta19] Andreas Stange. "Model checking for SCCharts". Master's thesis. Kiel University, Department of Computer Science, May 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf.

[STP05] Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. "The synchronous hypothesis and synchronous languages". In: *Embedded Systems Handbook*. Ed. by Richard Zurawski. CRC Press, 2005. DOI: 10.1201/9781420038163.ch8.

[Str87] Bjarne Stroustrup. "What is "Object-Oriented Programming"?" In: *European Conference on Object-Oriented Programming (ECOOP' 87)*. Vol. 276. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 51–70. DOI: 10.1007/3-540-47891-4_6.

[TL18] Stavros Tripakis and Roberto Lublinerman. "Modular code generation from synchronous block diagrams: interfaces, abstraction, compositionality". In: *Principles of Modeling — Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*.

Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Vol. 10760. Lecture Notes in Computer Science. Springer, 2018, pp. 449–477. DOI: 10.1007/978-3-319-95246-8_26.

[TYN13] Ewan Tempero, Hong Yul Yang, and James Noble. "What programmers do with inheritance in java". In: *27th European Conference on Object-Oriented Programming (ECOOP 2013)*. Ed. by Giuseppe Castagna. Vol. 7920. Lecture Notes in Computer Science. Montpellier, France: Springer, 2013, pp. 577–601. DOI: 10.1007/978-3-642-39038-8_24.

[Weg87] Peter Wegner. "Dimensions of object-based language design". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*. Orlando, Florida, USA: ACM, 1987, pp. 168–182. DOI: 10.1145/38765.38823.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. "The state of practice in model-driven engineering". In: *IEEE Software* 31.3 (2014), pp. 79–85. DOI: 10.1109/MS.2013.65.

[Wie98] Roel J. Wieringa. "A survey of structured and object-oriented software specification methods and techniques". In: *ACM Computing Surveys* 30.4 (1998), pp. 459–527. DOI: 10.1145/299917.299919.

[WNS+06] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. "An operational semantics and type safety proof for multiple inheritance in C++". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Portland, Oregon, USA: ACM, 2006, pp. 345–362. DOI: 10.1145/1167473.1167503.

[WSS+18] Nis Wechselberg, Alexander Schulz-Rosengarten, Steven Smyth, and Reinhard von Hanxleden. "Augmenting state models with data flow". In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of his 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Vol. 10760. Lecture Notes in Computer Science. Springer, 2018, pp. 504–523. DOI: 10.1007/978-3-319-95246-8_28.

[WSW+06] Ferdinand Wagner, Ruedi Schmuki, Peter Wolstenholme, and Thomas Wagner Thomas. *Modeling software with finite state machines: a practical approach*. Auerbach Publications, 2006.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. "Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. Atlanta, Georgia, USA: ACM, 1997, pp. 125–141. DOI: 10.1145/263698.263728.

[ZL08] Ye Zhou and Edward A. Lee. "Causality interfaces for actor networks". In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), 29:1–29:35. DOI: 10.1145/1347375.1347382.

[ZLL07] Yang Zhao, Edward A. Lee, and Jie Liu. "A programming model for time-synchronized distributed real-time systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, pp. 259–268. DOI: 10.1109/RTAS.2007.5.

Glossary

ADT Abstract data type
An abstract data type combines a named type with operations to create, combine, and observe values of that type [Coo09].

API Application programming interface
A contract for communication or interaction between software components.

ASAP As soon as possible
A colloquial abbreviation.

CCSL Clock Constraint Specification Language
A notation to handle clocks and to specify logical and chronometric time constraints [And09].

CFG Control-flow Graph
A directed graph notation in which nodes represent instructions or conditions and edges specify control-flow paths [All70].

CPS Cyber Physical System
A system that contains behavior defined in both the cyber and the physical domain [LS17].

ELK Eclipse Layout Kernel
A framework under the Eclipse umbrella that provides several layout algorithms, as well as the surrounding infrastructure.
Available at: `https://www.eclipse.org/elk`

FPGA Field Programmable Gate Array
An integrated circuit that is designed to support configuration by a hardware description language after manufacturing.

FSM Finite state machine
A form of automaton that uses a finite number of states.

GALS Globally Asynchronous Locally Synchronous System
A system design principle consisting of multiple locally synchronized subsystems that use asynchronous communication between these units [Cha84].

IDE Integrated Development Environment
An application that provides advanced features for developing software.

IURP Initialize-Update-Read Protocol
A three-staged intra-instant synchronization protocol defined in the context of sequential constructiveness to order concurrent variable accesses [HDM+14].

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
A development environment for SCCharts and other synchronous languages in combination with pragmatics-aware modeling technologies.
Available at: `https://www.informatik.uni-kiel.de/rtsys/kieler`

KLighD KIELER Lightweight Diagrams
A lightweight and extensible framework for automatic diagram synthesis and pragmatics-aware visualization [SSH13].
Available at: `https://github.com/kieler/KLighD`

LF Lingua Franca
A reactor-oriented polyglot coordination language [LMB+21].
Available at: `https://www.lf-lang.org`

MDE Model-Driven Engineering
A software development methodology that uses specialized domain models to express data or processes in a program.

MISRA Motor Industry Software Reliability Association
A collaboration between manufacturers, component suppliers, and engineering consultancies to promote best practices in developing safety-related electronic systems and software. See: `https://www.misra.org.uk`

MoC Model of Computation
A model that describes the computational process of a function producing outputs for given inputs.

MVC Model-View-Controller
A design pattern that divides software components into a model, views that represent the model, and controllers for interactions between the former [Ree79].

OO Object-Oriented
A programming paradigm that combines data and their operations into classes and object, and supports relations between them.

PTIDES Programming Temporally Integrated Distributed Embedded Systems
A discrete-event model that acts as a programming specification for time-synchronized distributed real-time systems [ZLL07].

ROOM Real-time Object-Oriented Modeling
A modeling concept for event-driven real-time distributed systems that leverages object orientation for modularity and abstraction [SGW94].

SC Sequentially Constructive
A synchronous model of computation that considers sequential relations in the source code, while ensuring deterministic behavior by a constructive analysis approach [HDM+14].

SCADE Safety-Critical Application Development Environment
A graphical dataflow language based on Lustre semantics [CPP17].

SCG Sequentially Constructive Graph
A control-flow graph notation that includes synchronous elements and dependencies for a low-level representation of sequentially constructive programs [HDM+14].

SCL Sequentially Constructive Language
A minimal imperative programming language with sequentially constructive semantics [HDM+14].

SCPL Sequentially Constructive Procedural Language
An imperative synchronous language with sequentially constructive semantics but stronger emphasis on procedural abstraction and shared memory communication [GGM+20; GGM+22].

SP Scheduling Policy
An object-based scheduling regime that uses automata for specifying precedence relations of operations [AMP+18].

SSM Sparse Synchronous Model
A synchronous model of computation that incorporates the sparseness of events into the execution regime [EH20].

UML Unified Modeling Language
A collection of modeling languages for the architectural design and behavior of software systems [Obj11].

VHDL Very High Speed Integrated Circuit Hardware Description Language
A standardized hardware description language supports specifying the structure and behavior of digital circuits on different abstraction levels.

List of Figures

1.1 Schematic of a Furuta pendulum with an LED and speaker. . . 3
1.2 The pendulum control program in LF. 5
1.3 The pendulum control program in SCCharts. 6

2.1 Abstract view on a reactive system embedded in its environment (based on [MHH13]). 17
2.2 LF-specific reactive program components and their interaction with each other and the environment. 24
2.3 The ABO SCChart [HDM+14], consisting only of Core SCCharts elements. 26
2.4 The FurutaPendulum program modeled as a hybrid dataflow SCChart. 33
2.5 SCCharts-specific reactive program components and their interaction with each other and the environment. 34
2.6 Two examples for specialized views in SCCharts and LF. . . . 47

3.1 An excerpt of the led output by the PendulumController reactor in a simulation. It illustrates a timing behavior of a non-aligned and an aligned timer in relation to the catch mode. . 54
3.2 A Furuta pendulum implementation in Ptolemy by Liu et al. [LEJ+02] 59
3.3 The PendulumController reactor using modes. 62
3.4 The ToggleLED reactor that models the alternating LED state with modes. 64
3.5 The TimingExample model for illustrating the different effects of reset and history transitions on timers and actions in modes. 71
3.6 The execution trace with reaction illustration for the TimingExample model in Figure 3.5. 72

3.7 The progression of time in each mode and their respective timer of the TimingExample model in Figure 3.5. 74
3.8 The Connection reactor. 75
3.9 The control components of the Furuta pendulum in a simulation setup connected to a user interface that supports pausing. 84
3.10 Two examples for LF models that can be accepted as deterministic/ causal due to the use of modes. 85
3.11 An example for a strong abort in SCCharts and a similar but shallow implementation in LF. 88
3.12 Conceptual structure of an LF compilation with an embedded SCChart. 92

4.1 Trafficlight controller modeled as timed automaton by Lee and Seshia [LS17]. 108
4.2 Illustration of variation in the timed behavior of timed automata based on the execution strategy. The execution traces show the reactions (vertical strokes) of the traffic light controller under different regimes. 111
4.3 The traffic light controller modeled in SCCharts using the timed automaton notation. 112
4.4 The traffic light controller SCChart after the clock transformation. 113
4.5 The traffic light controller SCChart after the advanced clock transformation using Scheduling Directives. 115
4.6 A concurrent counter reset, illustrating the fundamental problem with clocks managed by a single concurrent during action [SSH19]. 116
4.7 The PendulumSound component modelled as timed SCChart. . 120
4.8 The PendulumController component modelled as timed SCChart. 121
4.9 Example of a multiclocked SCChart that has two LEDs blinking in different frequencies. 123
4.10 Different timing abstractions [HBG17]. 125

4.11 Execution traces of the traffic light controller based on different execution strategies. Vertical strokes denote reactions. . . 127
4.12 The basic idea of dynamic ticks: the wake-up times are controlled by the tick function itself [HBG17]. 130
4.13 A dynamic tick function and its environment. Components in red are new compared to Figure 2.5. 132
4.14 The traffic light controller SCChart after the non-concurrent clock transformation producing dynamic ticks. 135
4.15 Motivating example for using soft bounds in dynamic ticks. . 139
4.16 Variant of FastAndSlowGreedy SCChart using soft reset in both regions. 141
4.17 Example for using a logical clock with dynamic ticks. 142
4.18 Observed pendulum behavior during simulation. 146
4.19 Comparison of a representative execution sample, illustrating the reduction of ticks in SCCharts with dynamic ticks. The sample shows the ticks on their respective timelines. The yellow boxes indicate the wake-up lag and gray ones the tick computation time. 148
4.20 Deviation of the sound signal from the expected angle-dependent note. 150
4.21 DS Demonstrator setup with annotations [BSH20b]. 152
4.22 Technical drawing of the DS motor assembly [BSH20b]. 153
4.23 The timed SCChart handling overcurrent protection in the controller [BSH20b]. 154
4.24 General structure of the dynamic tick environment with multicycle tick execution, implemented in the DS demo [BSH20b]. 155
4.25 Comparison of the FPGA and Raspberry Pi controller in experiments on the DS demo. [BSH20b] 157
4.26 Timed variant of the AO SCChart and its embedding Reactor. 160
4.27 The TimedSignal class in SCCharts extending the basic behavior with a scheduling and deadline option. 162

5.1 Varieties of polymorphism (based on [CW85]) 182
5.2 Visual representations of a Counter class. 187

5.3 The CounterApplication SCChart using Counter class. 188
5.4 Example of an SCChart class with a region. 190
5.5 First step in the high-level transformation of the CountingCounterApplication, replacing the SCCharts-based class by a native notation. 191
5.6 Conceptualized second step in the high-level transformation of the CountingCounterApplication with method inlining and simplification of the class data structure. 192
5.7 Examples of inheritance and overriding in SCCharts-based classes. 193
5.8 Example for usage of inheritance and overriding (left) and the result after inheritance is statically expanded by the compiler (right). Red arrows indicate where the parts of the model are expanded into. 197
5.9 An SCChart with a type parameter for a Counter classes that is instantiated and used to count to ten. 198
5.10 Screenshot of the ControlledCountingCounter in the KIELER tool. 201
5.11 The CountingCounterApplication and the involved classes arranged in a UML class diagram style. 202
5.12 Two variants of the CounterApplication using host code. 204
5.13 An attempt to model the Counter class as a reactor in LF. . . . 206
5.14 The CounterApplication with a host class and a custom schedule. The resulting scheduling instructions are visualized as green arrows. 209
5.15 The CounterApplication with a host class and a policy automaton. The resulting scheduling instructions are visualized as green arrows. 211
5.16 Logger example modelled using the classical module approach. 217
5.17 The PureSignal class modelled in SCCharts. 219
5.18 The AbstractValuedSignal class that carries a generic value and requires the implementation of a combination function. 220
5.19 The PureSCSignal class extending pure signals by an unemit method with a custom scheduling protocol. 222

5.20 The Counter class modelled as a classical SCCharts module with boolean variables for interaction. 224
5.21 The CounterApplicationDF SCChart using the Counter class with methods in a dataflow region. 224
5.22 Composition of the main components in the steam boiler controller. 228
5.23 Inheritance relations between main components in the steam boiler controller. 229
5.24 The different control modes in the Controller SCChart for the steam boiler. 231

6.1 A mockup of a behavior tree in LF. The two variants illustrate different options for representing the same behavior. 240

List of Tables

2.1 A brief overview and comparison of key aspects in LF and SCCharts. It covers the current state of the implementation, plus references to extensions made by this thesis. 16

4.1 Results of the Furuta pendulum simulation. 147

5.1 Characteristics of OO features in the current implementation of SCCharts. 215

List of Code Listings

2.1 Abstract implementation of a tick function loop for the PendulumSound SCChart in C. 36
2.2 Source code of the FurutaPendulum main program in LF. 39
2.3 Source code of the FurutaPendulum main program in SCCharts. 40
2.4 Source code of the PendulumSound component in LF. 41
2.5 Source code of the PendulumSound component in SCCharts. . . 42
2.6 Source code of the PendulumController component in LF. 44
2.7 Source code of the PendulumController component in SCCharts. 46

3.1 Source code of the PendulumController reactor with modes. . . 63
3.2 Source code of the ToggleLED reactor. 64
3.3 Source code of the Connection reactor illustrating manual management of a socket connection. 75
3.4 Source code of the ShallowStrongAbortExample reactor. 88

4.1 Tick loop example for dynamic ticks. 133
4.2 Source code of the TimedAO reactor embedding the SCChart. . 160

5.1 Data structures in Blech and Céu, illustrating a counter with an additional routine that increments the value every tick respectively millisecond. 172
5.2 A simple generic interface for a stack in Java 182
5.3 Textual SCCharts-based Counter class. 187

6.1 A mockup of the code structure generated by an OO state-based approach in Java for the CountingCounterApplication in Figure 5.4b and CountingCounter with inheritance. 246

List of Algorithms

3.1 Processing of mode transitions at the end of each execution cycle. 80