

Interactive Model-Based Compilation

A Modeller-Driven Development Approach

Dipl.-Inf. Steven Smyth

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2020

Kiel Computer Science Series (KCSS) 2021/1 dated 2021-04-01

URN:NBN urn:nbn:de:gbv:8:1-zs-00000374-a1

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)

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

The author can be contacted via ssm@rtsys.org

Published by the Department of Computer Science, Kiel University

Real-Time and Embedded Systems Group

Please cite as:

- ▷ Steven Smyth. *Interactive Model-Based Compilation — A Modeller-Driven Development Approach* 2021/1 in Kiel Computer Science Series. Department of Computer Science, 2021. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Smyth21,  
  author    = {Steven Smyth},  
  title     = {Interactive Model-Based Compilation  
             --- A Modeller-Driven Development Approach},  
  publisher = {Department of Computer Science, CAU Kiel},  
  year      = {2021},  
  number    = {2021/1},  
  doi       = {10.21941/kcss/2021/1},  
  series    = {Kiel Computer Science Series},  
  note      = {Dissertation, Faculty of Engineering,  
             Kiel University.}  
}
```

© 2021 by Steven Smyth

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
2. Gutachter: Prof. Dr. Bernhard Steffen
Technische Universität Dortmund
Dortmund

Datum der mündlichen Prüfung: 07.12.2020

Abstract

There is a growing tendency for using domain-specific languages, which help domain experts to stay focussed on abstract problem solutions. It is important to carefully design these languages and tools, which fundamentally perform model-to-model transformations. The quality of both usually decides the effectiveness of the subsequent development and therefore the quality of the final applications. However, as the complexity and safety requirements of modern systems grow, it becomes increasingly burdensome to create highly customized languages and difficult to provide reasonable overviews within these tools.

This thesis introduces a new interactive model-based compilation methodology. Compilations for arbitrary model-to-model transformations are themselves described as models. They can be instantiated for particular inputs, e. g. a program, to create concrete compilation runs, which return the result of that compilation. The compilation instance is interactively observable. Intermediate results serve as new inputs and as documentation. They can be used to create highly customized views and facilitate understandability. This methodology guides modellers from the start of the compilation to the final result so that they can interactively refine their models.

The methodology has been implemented and validated as the KIELER Compiler (KiCo) and is available as part of the KIELER open-source project. It is used to implement the current reference compiler for the SCCharts language, a statecharts dialect designed for specifying safety-critical reactive systems based on a synchronous model of computation. The interactive model-based compilation approach was key to the rapid prototyping of three different compilation strategies, as well as new language extensions, variations and closely related languages. The results are verified with benchmarks, which are again modelled using the same approach and technology. The usability of the SCCharts language and the KiCo tooling is documented with long-term surveys and real-life industrial, academic and teaching examples.

Zusammenfassung

Es gibt eine steigende Tendenz im Einsatz von domänen-spezifischen Sprachen, welche den Experten dieser Domänen helfen sollen, sich auf die eigentlichen abstrakten Problemlösungen zu konzentrieren. Diese Sprachen und Werkzeuge, welche grundlegend Model-zu-Model-Transformationen ausführen, müssen sorgsam entwickelt werden. Für gewöhnlich entscheidet die Qualität beider über die Effektivität des nachfolgenden Entwicklungsprozesses und letztlich über die Qualität der finalen Anwendungen. Da die Komplexität moderner Systeme, zum Beispiel durch erhöhte Sicherheitsanforderungen, stetig steigt, wird es zunehmend mühsam, maßgeschneiderte Sprachen und Werkzeuge mit verständlichen Übersichten zu erstellen.

Diese Arbeit führt eine neue, interaktive, modell-basierte Übersetzungsmethodik ein. Übersetzungen für beliebige Modell-zu-Modell-Transformationen sind ebenfalls als Modelle definiert. Sie werden für bestimmte Eingaben, wie z.B. Programme, initialisiert, um konkrete Übersetzungsläufe zu starten. Die gesamte Instanz ist interaktiv beobachtbar und das finale sowie alle Zwischenergebnisse werden verständlich veranschaulicht. Diese dienen als neue Eingaben und zur Dokumentation. Sinnvolle, dedizierte Ansichten fördern das Verständnis des Modellierers. Diese Methodik leitet den Modellierer von Beginn an bis hin zum Endergebnis der Übersetzung, so dass die Modelle interaktiv verfeinert werden können.

Die Methodik ist als KIELER Compiler (KiCo) im KIELER Open-Source-Projekt implementiert. Sie dient als Grundlage für den aktuellen Referenz-Übersetzer der SCCharts-Sprache, einen Statecharts-Dialekt, welcher für die Spezifikation von sicherheitskritischen Systemen entwickelt wurde und auf einem synchronen Berechnungsmodell basiert. Der interaktive, modell-basierte Übersetzungsansatz spielt eine Schlüsselrolle in der Prototypenentwicklung von drei verschiedenen Übersetzungsstrategien für SCCharts, neuen Spracherweiterungen und verwandten Sprachen. Die Ergebnisse werden mittels Vergleichswerten, modelliert mit dem gleichen Ansatz und der

gleichen Technologie, überprüft. Die Benutzbarkeit der SCCharts-Sprache und von KiCo ist mit Langzeitstudien, industriellen, akademischen und auch Lehrbeispielen aus der Praxis dokumentiert.

Nothing's as good if you don't share it.

— After Life by Ricky Gervais

Contents

1	Introduction	1
1.1	Contributions	6
2	Preliminaries	13
2.1	SCCharts	13
2.2	The KIELER SCCharts Modelling Tool	23
2.3	Related Work	26
I	Interactive Model-Based Compilation	57
3	Model-Based Compilation Systems	59
3.1	A Generic User Story	59
3.2	Modelling Foundations	65
3.3	Interactive Compilation Systems	73
3.4	Derived Compilation Systems	91
4	Interactive Prototyping	95
4.1	Interactive Guidance	96
4.2	Compilation Systems Universes	107
4.3	Transformation Complexity	110
II	Beyond Compiling SCCharts	115
5	Interactive Compilation for SCCharts	117
5.1	The Sequentially Constructive Kernel Language	118
5.2	Netlist-Based Compilation	125
5.3	Priority-Based Compilation	182

Contents

5.4	State-Based Compilation	193
5.5	Evaluation of Compilation Approaches	210
6	Rapid Prototyping	237
6.1	Scheduling Directives	237
6.2	SCCharts Dataflow	247
6.3	Symmetrical Actions	267
6.4	Sequentially Constructive Esterel	270
6.5	Model Extraction of C Code	274
6.6	SCCharts Language Evaluation	277
7	Practical Applications	303
7.1	Specification and Verification	304
7.2	Runtime Enforcement	312
7.3	Teaching	315
8	Conclusion	329
8.1	Summary	329
8.2	Future Work	331
A	Publications	341
B	Related Modelling Tools	359
C	SCCharts iMURD	371
D	Extended SCCharts Features	373
	Bibliography	387
	Lists	413

Introduction

Although few people are likely to build or even maintain a compiler for a major programming language, the reader can profitably apply the ideas and techniques discussed in this book to general software design.
— *Compilers – Principles, Techniques, and Tools*

Some might see the aforementioned book from Aho, Sethi and Ullman [ASU86] as the bible of compiler construction. Even so the general statement of the quote remains untouched as many ideas and techniques are great additions to the repertoire of a software engineer, modern tools kits make it increasingly easier to create compilers for various areas of interest. Especially Domain-Specific Languages (DSLs) profit from this development. However, designing, understanding and maintaining these complex programs are different stories. Modern general purpose language compilers for example perform a lot of analyses and optimizations. Although the user has some degree of control over the compilation, compilers are usually *black boxes* or even a combination of these. Therefore, it is relatively hard to grasp beforehand how the final result of such a compilation chain will look like. It might be even harder to predict intermediate results which in addition are not necessarily executable models or comprehensible in a human-readable way. Maybe, it is not without reason that the infamous book is unofficially dubbed *The Dragon Book*.

1. Introduction

```
1 int rec_sum(int* arr, int size) {
2     if (size == 0) {
3         return 0;
4     } else {
5         return arr[0] + rec_sum(arr +
6             1, size - 1);
7     }
}
```

(a) sum_rec function in C

```
1 .LFB0:
2     pushl %ebx
3     movl 12(%esp), %edx
4     xorl %eax, %eax
5     movl 8(%esp), %ecx
6     testl %edx, %edx
7     je .L2
8 .L3:
9     movl (%ecx), %ebx
10    addl $4, %ecx
11    addl %ebx, %eax
12    subl $1, %edx
13    jne .L3
14 .L2:
15    popl %ebx
16    ret
```

(b) Generated assembler code of sum_rec with O2 optimization setting

```
1 rec_sum:
2 .LFB0:
3     pushl %ebp
4     movl %esp, %ebp
5     pushl %ebx
6     subl $20, %esp
7     cmpl $0, 12(%ebp)
8     jne .L2
9     movl $0, %eax
10    jmp .L3
11 .L2:
12    movl 8(%ebp), %eax
13    movl (%eax), %ebx
14    movl 12(%ebp), %eax
15    leal -1(%eax), %edx
16    movl 8(%ebp), %eax
17    addl $4, %eax
18    movl %edx, 4(%esp)
19    movl %eax, (%esp)
20    call rec_sum
21    addl %ebx, %eax
22 .L3:
23    addl $20, %esp
24    popl %ebx
25    popl %ebp
26    ret
```

(c) Generated assembler code of sum_rec with O0 optimization setting

Listing 1.0.1. Classical results of different compilation optimizations

An example of a classical compiler optimization can be seen in Listing 1.0.1. Listing 1.0.1a shows a C program that calculates the sum of a given array recursively. When compiling this code with a traditional compiler, such as the GCC¹, the user can usually choose between different levels of optimizations. In the example, Listing 1.0.1c shows an intermediate result of the compilation with no optimization whereas Listing 1.0.1b depicts a optimization level of two (-O2). The intermediate results are assembler codes and readable for a programmer trained in assembler. However, they

¹<https://gcc.gnu.org/>

<pre> 1 x = 3; 2 if (x > 0) { 3 x = x * 7; 4 } 5 y = x * 2; </pre>	<pre> 1 x_3 = 3; 2 if (x_3 > 0) 3 goto <bb 3>; [0.00%] 4 else 5 goto <bb 4>; [0.00%] 6 </pre>	<pre> 7 <bb 3> [0.00%]: 8 x_4 = x_3 * 7; 9 10 <bb 4> [0.00%]: 11 # x_1 = PHI <x_3(2), x_4(3)> 12 y_5 = x_1 * 2; </pre>
---	--	--

(a) A small C program (b) Code excerpt of intermediate representation from the GCC, generated with the `-fdump-tree-ssa` option.

Listing 1.0.2. Illustration of intermediate compilation result in the GCC (from [SSH18d])

are no longer in the source language and there is no information on how these assembler snippets were created. Even for a trained developer, understanding large pieces of assembler becomes difficult. Moreover, intermediate optimization steps may not be obtained; at least not in a human-readable form. Also, `-O2` itself is an abstract level which does indicate the kind of optimization (although choosing different distinct optimizations is of course possible with the GCC). Without a look into the documentation, it simply states *optimize a lot*.

Nonetheless, the optimization is salient. Without going into the details of assembler, one can see the recursive call of the `sum_rec` function in the unoptimized variant in Listing 1.0.1c in Line 20. Besides only needing 16 lines of code instead of 26, the optimized version in Listing 1.0.1c also does not include the call. The optimization completely resolved the recursion. It would be fantastic to see these optimizations instantaneously in a comprehensible way step by step.

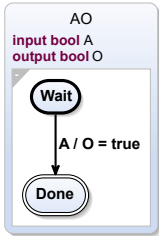
As a second example, Listing 1.0.2 illustrates an intermediate representation of the C compilation of the GCC. In compiler construction, a program commonly gets divided in Basic Blocks (BBs). Basically, a BB represents an atomic sequence of program instructions without any branches [All70]. The partitioning of programs in their basic blocks usually makes subsequent

1. Introduction

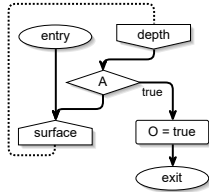
analyses of compilers more efficient. Listing 1.0.2a shows an excerpt of an C program. It first sets a variable x to 3. Then, x is tested and if greater than zero, the assigned value gets multiplied by 7. In the end, variable y is set to x multiplied by 2. When compiled with the GCC, the excerpt gets partitioned into its BBs, which we consider as an Intermediate Representation (IR) of the overall compilation. Using the GCC, the user can look at this result with the `-fdump-tree-ssa` command line option. The tree is shown in textual form as depicted in Listing 1.0.2b. Without going into too much detail, one can see different incarnations of the variables x and y and goto instructions to BBs, represented by the angle brackets and the bb identifier. While the information of the IR is present, the presentation and possibilities to navigate inside this intermediate result could be improved.

Figure 1.0.1 shows an excerpt of the compilation of an SCChart model within the KIELER tool, which will both be discussed in the following sections. The program waits for an input A to occur and then emits an output O before terminating. Figure 1.0.1a – Figure 1.0.1f show some of the intermediate steps of the compilation, each in an IR fitting for the *abstract syntax*. Figure 1.0.1b uses the same abstract syntax, or *metamodel*, as the source model, whereas Figure 1.0.1c – Figure 1.0.1f use a different one. Even without previous knowledge of compiler construction, a user can spot relationships between the individual IRs, which facilitates the overall understanding on how the steps are connected and lead to the final result.

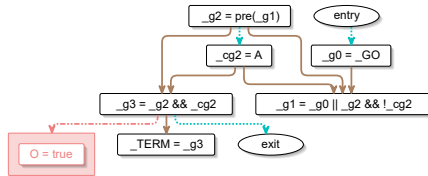
This thesis presents a methodical approach for these interactive compilation models and exemplifies their use and usefulness with the help of the synchronous language SCCharts, as the IRs of visual languages facilitates the modeller’s understanding of the overall workflow and single steps. The approach, however, is not restricted to visual languages or a single language.



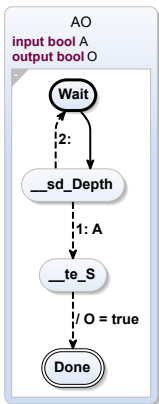
(a) Source SC-Chart



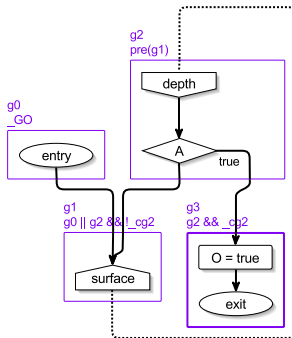
(c) Control-flow graph



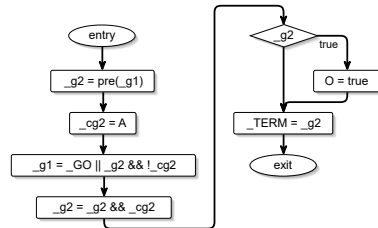
(e) Generated net-list



(b) Normalized version



(d) Basic blocks



(f) Sequentialized net-list

Figure 1.0.1. Model-driven software development example with interactive compilation from the KIELER SCCharts project: In this example, a) a minimal SCCharts program AO gets compiled. The modeler can inspect every aspect of the compilation as every intermediate result is itself a model that can be visualized instantaneously and in a comprehensible way. This example compilation proceeds along the depicted figures: b) normalization, c) control flow graph creation, d) basic block analysis, e) net-list generation, and f) sequentialization. Each step is explained in detail in Part II of this thesis.

1. Introduction

1.1 Contributions

The contributions of this thesis are divided into two parts. My [relevant publications](#) and [supervised theses](#) are cited within the enumeration. Closely related work follows afterwards. A comprehensive list of all publications and supervised theses is given in Appendix A. More related work follows in Chapter 2. In Part I,

1. Chapter 3 presents, a new methodology of **interactive model-based compilation**. The concept builds upon the previously established Single-Pass Language-driven Incremental Compilation (SLIC) approach and enables a developer to define complete compilation processes interactively [[SSH18c](#)]. It systematically establishes a notion for model-based compilation systems. These compilation systems are formed by loosely coupled modular model processor to enable arbitrary tasks, such as compilation, simulation and deployment. They are full models themselves and are mutable just as the models they work on during their lifetime.
2. Building upon the interactive model-based compilation process, I propose a **modeller-driven development process** in Chapter 4. This results in a refinement feedback loop which supports the modeller in their work, which I will refer to as Interactive Model-Understanding-Refinement-Documentation feedback loop (iMURD) loop [[SSH18a](#)]. I investigate unaugmented and augmented model representations, which guide the modeller during the interactive model-based development process. While the concept is not restricted to any particular views, I present six practical examples from the KIELER open-source project, namely, data dependencies, causal dataflow, scheduling propagation, transformation snapshots, automatic element tracing and built-in code mapping. Tool developers can use similar dedicated views to get an overview of the complete compiler infrastructure and universe.

The concepts proposed in Part I are implemented as generic compiler infrastructure, the KiCo. KiCo’s highly modular nature turns the infrastructure into a *meta-tool engine*, which allows arbitrary languages to utilize already established compilation chains.

Part II contributes in the practical domain:

3. I illustrate three different low-level compilation approaches for the synchronous language SCCharts to demonstrate the capabilities of the interactive model-based compilation concept in Chapter 5. The **netlist-based compilation** creates code for software and hardware [SMH15; RSM+16; SLH16][Wei15; Bus16]. While many compilation steps are based on standard compilation techniques, the interactive model-based approach makes a comprehensible observation of all IRs possible. Here, optimizations and synchronous peculiarities, such as schizophrenia, are also addressed [SSH18d][Bus16]. The **priority-based compilation** approach [Pei17] does not synthesize hardware but scales better w.r.t. to model-size. The **state-based compilation** approach, coming in three variants, facilitates understandability and readability [SMH18; SDH19]. All approaches are evaluated against each other in terms of execution times and user experience. Since capabilities of a language and readability and simplicity are often a question of balance, I formulate **ten language requirements** which help decision makers in designing similar languages.
4. Chapter 6 presents how the concepts from Part I also facilitate the rapid development of various extensions to the SCCharts language and its underlying Model of Computation (MoC). Specifically, I present novel **Scheduling Directives** (SDs) as a first class citizen of synchronous languages which directly influence the scheduling of the used MoC [SSH19; SSH18b]. Furthermore, a **dataflow extension** is added to SCCharts, as control-flow-oriented language example, which allows dataflow and control-flow regions to co-exist in the sequential constructive world of SCCharts [GSS+20; SSS+19; MS15][Um15]. The Sequentially Constructive Model of Computation (SCMoC) is implemented into the Esterel language using the interactive model-based compilation approach forming a new language dialect, **SCEst** [SMR+17; RSM+15][Rat15; Rah17]. Further, I show how the interactive model-based approach can be used to turn **legacy general purpose languages** automatically into comprehensible models that already have a working and tested compilation chain [SLH16][Ols16; Len16; And19]. The SCCharts language and the KiCo tooling are evaluated thoroughly through experimental data and surveys which were gathered during the span of more than five years [SMS+15; MSS+16; SMS+19a].

1. Introduction

5. To demonstrate further possibilities of the presented approach, I present several **practical applications** in Chapter 7, namely *transient statecharts* from specifications verified by model checking [Sta19], a real-life cyber-physical pacemaker system [PRS+17b; PRS+17a] and applications in teaching, such as Lego Mindstorms [SMS+19a] and student projects involving a railway installation [SMS+15][Eum17], a quadcopter [Pei15; And15; Mac15] and the Formula Student [SSS+19].

Closely Related Work 1) and 2) are closely related to general purpose compiler infrastructures, such as LLVM [Lat02], which serve as modular frameworks for arbitrary compilers by providing IRs and strictly separate language front-ends from the compiler back-end. The concepts presented here, however, stay as long as possible in the meta-model of the developer to present intermediate compilation steps in a comprehensible way. This facilitates understanding and results in a refined modelling process.

The approach presented here is a generic refinement of the work done w.r.t. SLIC by Motika [Mot17]. SLIC is specifically tailored to the features to the language in question. *Produces* and *can-not-handle* dependencies define the order of model-to-model transformations invocations. The interactive model-based compilation approach creates a generic framework for models, transformations and meters, which influence and evaluate model-based transformation chains. In fact, the approach presented here can be used to re-define and re-implement the previous SLIC approach generically, as demonstrated in Section 3.4.

A closely related modelling process is provided by *meta-tool generators*, such as CINCO [NLK+18]. CINCO is designed for domain experts and can be used to create highly customized Visual Domain-Specific Languages (VDSLs) on demand. While the concept targets a different meta level in creating self-sustained modelling tools, a paramount goal of both concepts is simplicity. They differ in the fact that the interactive model-based compilation approach is not a concept for a generator but for an engine that drives domain-specific requirements.

The processor categorization, orchestration and compilation universe presented in thesis are related to ETI's concept of loose coupling of ser-

vices [SMB97]. ETI differentiates between two communities, the tool builders and users, to increase simplicity for the latter. The approach presented here goes one step further in creating a feedback loop that facilitates the development process by interactively providing the modeller with comprehensible compilation information. In turn, the tool developer can also profit from the process since the same information is used to refine the underlying transformations.

The graphical syntax and, therefore, the ways of presenting information in this thesis is similar to other modelling tools and Statechart [Har87; Dou99] dialects in general, particularly SyncCharts [And96b] in the case of SCCharts. The diagrams are extended by new information, such as dependency or scheduling information. To display these diagrams interactively in IRs, automatic layout [Fuh11; Rüe18; Sch19] and transient view technologies [SSH13] are used.

The compilation approaches in 3) use several well-established compilation techniques, such as the Program Dependency Graph (PDG) [FOW87] and copy propagation [ASU86]. They use common statecharts generation patterns [PM03; Sam02]. However, they are adapted to the interactive model-based compilation approach to facilitate understandability. Although possible, the goal is not to reach the most efficient code quickly but to provide the modeller with information about what is happening. The code generations and extensions 4) for synchronous languages are based on established procedures w.r.t. the peculiarities of synchronous MoCs, such as synchronizers and schizophrenia [PEB07; EZ07]. Here, understandability is also the goal in this thesis. Especially the presented state-based approaches are designed for readability and tailored to the topology of SCCharts and statechart dialects in general.

Outline Chapter 2 introduces the primary example language, SCCharts and its underlying MoC. It follows with detailed related work and related modelling tools. The chapter further includes the complete lists of publications and supervised theses.

The remainder of this thesis is structured according to the list of contributions. In Part I, Chapter 3 gives a motivation for the interactive model-based compilation approach and explains it in detail. The iMURD methodology and

1. Introduction

details on the framework usability from the perspectives of a tool developer and of a modeller are discussed in Chapter 4. In Part II, the methodology established in Part I is used to create the reference compiler for SCCharts implemented in the KIELER SCCharts tools. While SCCharts serve as primary example in Chapter 5, the concepts can be applied to any statechart variation or even completely distinct languages. Chapter 6 explains additions to the SCMoC. Practical use-cases, which have been successfully developed using the technologies presented in this thesis, are demonstrated in Chapter 7. I conclude and give ideas for future work in Chapter 8.

On a higher abstraction level, the relationships between the contents of the different chapters form another refinement cycle, which is illustrated in Figure 1.1.1. Tool developers use the concepts presented in Chapter 3. Following iMURD, they can refine and document prototypes, Chapter 4, until a satisfactory result is achieved. These are used in production, Chapter 5, to create whole compilation universes, e. g. for statechart dialects. The generated meta-tool engine can be extended to the actual needs of the domain experts, Chapter 6. Practical applications, Chapter 7, give new insights on the current workflow, which can then be adjusted to increase efficiency. The lessons learned can be used to iteratively improve the transformations that drive the used meta-tool engine.

1.1. Contributions

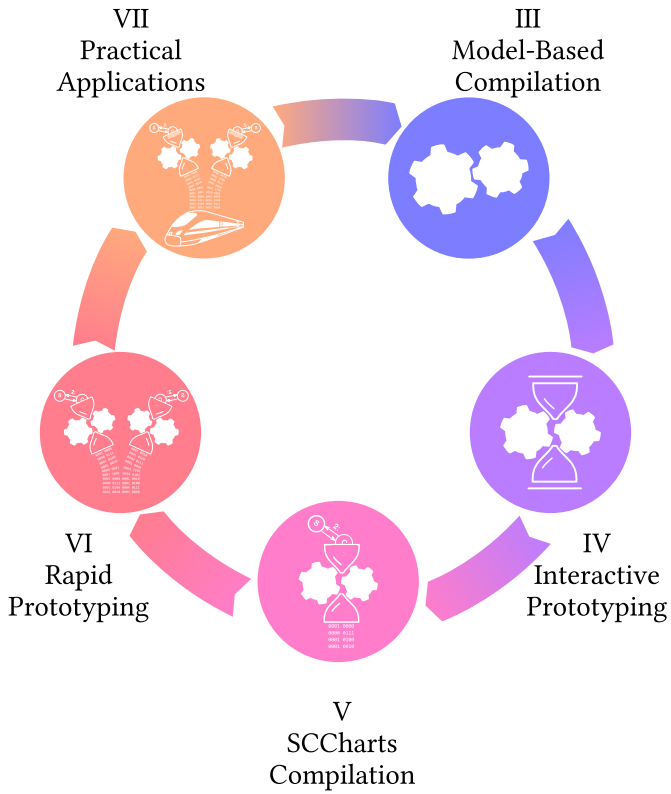


Figure 1.1.1. Outline of the development ecosystem

Preliminaries

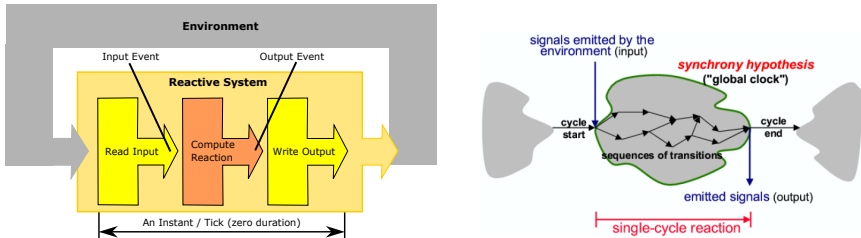
*Antoine de Saint-Exupéry, the French writer and aircraft designer, said that
“A designer knows he has arrived at perfection
not when there is no longer anything to add,
but when there is no longer anything to take away”.*
*More programmers should judge their work by this criterion.
Simple programs are usually more reliable, secure, robust and efficient
than their complex cousins, and easier to build and maintain.*
— *Programming Pearls*

This chapter introduces preliminaries about SCCharts, the primary example language, and synchronous languages in Section 2.1. Section 2.2 presents the open-source project KIELER, which comprises the reference implementations of the interactive model-based compiler infrastructure and SCCharts. Section 2.3 discusses further related work and modelling tools in detail.

2.1 SCCharts

SCCharts is a synchronous language developed for safety-critical applications [HDM+14]. Chronologically, it can be seen as a successor of SyncCharts [And95], which will be discussed further in Section 2.3.5, because SCCharts conservatively extends its MoC. This means that every SyncCharts program is also valid in SCCharts. However, the SCCharts SCMoc also includes sequentiality and, hence, accepts a broader class of programs. SyncCharts, using the statecharts formalism proposed by Harel [Har87], can

2. Preliminaries



(a) Typical schematic of an embedded, reactive system encapsulated in its surrounding environment [MHH13]

(b) Macro and micro ticks in synchronous languages (G. Luetzgen, 2001)

Figure 2.1.1. Reactions in synchronous languages

be seen as a graphical form of Esterel [BC84], one of the most prominent synchronous languages.

2.1.1 Synchronous Languages

Deterministic behaviour becomes difficult when it comes to concurrency. In traditional programming languages, concurrent execution is prone to *race conditions* [Lee06]. Synchronous languages are designed to ensure determinism [BCE+03]. Due to their predictability, synchronous languages are well-suited to model reactive and safety-critical systems. Figure 2.1.1a depicts a typical schematic of an embedded, reactive system. A reactive system constantly reacts to stimuli from its surrounding environment, also known as *inputs*. The system then computes a reaction and sends the *outputs*, e. g. some actuator control, back to the environment. The environment specifies the pace of this cyclic execution, which continues potentially indefinitely.

The classical synchronous MoC states that time is divided in discrete chunks, called *instants* or *macro ticks*. They are often also simply referred to as *ticks*. A macro tick, as depicted in Figure 2.1.1b, represents one reaction of the system, including reading inputs, computation, and writing the outputs. While a macro tick may consist of finitely many single calculations, *micro ticks*, the overall tick is considered to consume no time. Hence, the outputs are generated at the same time when the inputs are read. This is also

known as the *synchrony hypothesis* and is generally used in all synchronous languages. Esterel, Lustre, and Signal are prominent representatives of these languages.

Such languages use *signals* to communicate with their environment and also across concurrent regions of a program. Because a tick represents a discrete point of time, signals traditionally must have a coherent status within the same tick. The *signal coherence law* states that within a tick a signal can only be absent or present but not both at the same time. A signal S is absent by default and only present in a tick if and only if S is emitted in this tick.

```
1 if (!x) {
2   // do something
3   x = true;
4 }
```

Listing 2.1.1. Typical sequential programming pattern forbidden in classical synchronous languages: set flag x to true after some work has been done.

Depending on the concrete semantics, this law sometimes varies slightly, but the consequence for classical synchronous languages is overarching: A signal must have a unique state within a tick. Hence, deterministic behaviour and predictability come at a price. The class of programs that are considered valid, or *constructive*, is restricted, because even classical sequential programming pattern, as depicted in Listing 2.1.1, are forbidden.

Here, a boolean variable x is checked. If its value is false, a specific task *Example* should be executed. Afterwards, x is set to true to indicate the completion of the task. If x were a synchronous signal, this program would not be constructive, because x would not have a coherent status as it would be false and true within the same tick.

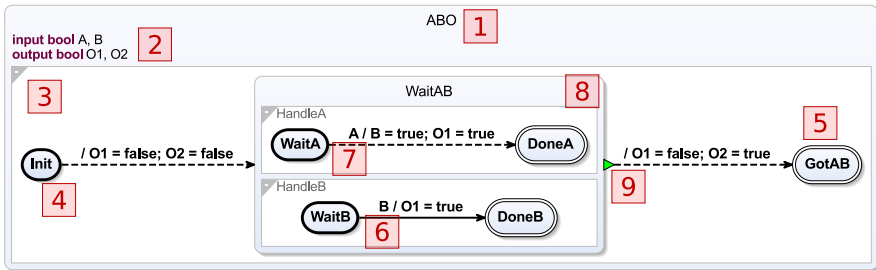
The SCM_{oC} relaxes this restriction. Instead of using signals which are uniquely defined per tick, the SCM_{oC} directly uses variables which can change their values in every micro tick. All non-concurrent variable access are scheduled in sequential order. Hence, deterministic behaviour and predictability remain untouched while enabling the modeller to create programs that use well-known imperative programming patterns. It is a conservative extension to the classical synchronous MoC. Every program that is constructive under the classical MoC is also valid under the SCM_{oC}.

2. Preliminaries

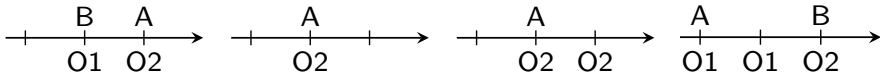
2.1.2 Core SCCharts

SCCharts is divided into two conceptual parts, *Core* and *Extended* SCCharts. Core SCCharts includes all essential language elements, while Extended SCCharts adds further features on top of Core SCCharts for the modeller's convenience. These features make it more convenient for a modeller to express certain functionality even though it is possible to express the same functionality with more basic features. These models often appear to be more compact because their complexity is hidden. However, all complex features must be translated into the *kernel language* which the compiler understands and uses to generate the output.

Example The essentials of SCCharts are exemplified in Figure 2.1.2a. The figure shows ABO, the *Hello World!* of sequential constructive statecharts. Every SCChart begins with a header area which contains the name of the chart in the middle **1** and an optional *declaration* part **2**. This outermost structure is called the *root state*. The declaration part in Core SCCharts can include *input*, *output*, and *local variables*. Input and output variables in the declaration area of the chart form the *interface* of the program, because they specify the ways the program can communicate with its surrounding environment. Note that a program that does not specify an interface has no observable behaviour and is semantically equivalent to `NULL` (if side-effects are excluded). The chart further contains one or more *regions* that contain *states* and *transitions* between states **3**. Regions are units of potentially concurrent control-flow and can be seen as threads. If active, they have their own control flow and run concurrently to other regions at the same nesting level. States are locations inside a region where the control of a region rests. They are connected via transitions, which are used to transfer control to another state. If a region becomes active, the control starts at the *initial* state of the region. It is visualized with a thick border **4**. The region terminates, if a *final* state, displayed with a double-border **5**, is reached. Every region must have exactly one initial state and can include arbitrarily many final states. A transitions connects a *source* state with a *target* state. A transition is taken, if its source state is active and the trigger of the transition evaluates to true. An omitted trigger implicitly states true. If multiple transitions exist, a transition priority determines in which order the transitions are checked.



(a) The “Hello World!” of sequential constructive statecharts: ABO



(b) Tick line #1: Output O is true as soon as both inputs A and B were true.

(c) Tick line #2: Input A implicitly also sets input B.

(d) Tick line #3: Behavior of O if GotAB was not a final state.

(e) Tick line #4: HandleB does not react in the first tick due to the delayed transition.

Figure 2.1.2. Syntax and behaviour of the *Hello World!* of sequential constructive statecharts: ABO

Only the first transition which is eligible to activate is considered. However, transitions have two different *delay types*, *delayed*, displayed as a solid edge [6], and *immediate*, drawn dashed [7]. A delayed transition cannot activate, if its source state activated in the current tick. Hence, a delayed transition can only become active, if the state was already active at the start of this tick and has not been re-entered in the meantime. When a transition is activated, the control is transferred to the target state and optional effects can be executed. States can be *simple states* or *superstates* [8]. Besides marking a point in the control-flow of the region, a superstate also has inner behaviour. All regions inside the superstate are activated when the superstate itself becomes active. In Core SCCharts, a superstate is exited with a *termination* transition [9], also known as *join*. A termination transition, visualized with a green triangle at the start of the transition, usually has no trigger. It immediately becomes active when all regions of the superstate have terminated. A superstate can also declare new local variables which are scoped hierarchically. The

2. Preliminaries

rootstate of the program is a superstate which declares inputs and outputs and which cannot have any outgoing transitions. The program terminates when the rootstate terminates.

When executed, the control of the program depicted in Figure 2.1.2a starts in the initial state of the root state ABO. It immediately traverses along the only outgoing transition to the superstate WaitAB, because the transition is immediate and the omitted default trigger is always true. Here, the two regions HandleA and HandleB wait for the input signals A and B concurrently. Each region contains an initial Wait state from which a transition points to a final done state. When the transition is taken, O1 is set to true. The transition in HandleA also sets B. After both regions reach their final state, WaitAB terminates and the termination transition is immediately active. Its action sets O1 to false and O2 to true. Once GotAB is reached, the program terminates.

Figure 2.1.2b–Figure 2.1.2e show four example *tick lines*. Tick lines are a typical means in synchronous languages to illustrate the input/output behaviour of the program. Discrete ticks are marked on the axis. Inputs are written above the tick; outputs below. In Figure 2.1.2b the input B is true in the second tick. After A becomes true in the third tick, the program emits O2. In Figure 2.1.2c A is already present in the second tick. B is not true in the surrounding environment. However, when A becomes true, it also sets B to true as can be seen in HandleA. O2 is emitted in the second tick, but O1 is set to false again because of the termination, which is not permitted in classical synchronous MoCs. Since the program terminates as soon as O2 becomes true, there is no more behaviour after the second tick. This would be different if GotAB would not be a final state. Then, the control would rest in GotAB and since SCCharts use persistent variables for communication, the last variable configuration will remain indefinitely as depicted in Figure 2.1.2e. This does not hold for input variables though, as inputs are set by the environment in every tick. In Figure 2.1.2e, A is true in the first tick. This time, O2 stays false due to the delayed transition in HandleB. The transition is not active in the first tick and HandleB cannot terminate. Furthermore, B is an input and set from the environment in every tick. Hence, the B assignment from the first tick is lost. O2 is finally emitted after B becomes true again in tick
_ three.

2.1.3 Concurrent Scheduling

The previous two sections show that concurrency is an essential property of SCCharts and Statechart dialects in general. On the abstract modelling level of synchronous languages, concurrent executions of statements become interesting if they access the same variable in the same tick. It is easy to create concurrent programs that do not behave deterministically. Since deterministic behaviour is an intrinsic feature of all synchronous languages, clear scheduling rules must be established.

A variable can be accessed in two different ways. First, it can be written to, which is referred to as a *write access*. Second, a variable can be read to retrieve a previously assigned value, which is considered a *read access*. Two variable accesses are *commuting* if the order in which they are executed does not matter. If all variable accesses inside a schedule for a concurrent context are commuting, the schedule becomes *confluent*. *Definition*

For instance, two assignments which assign the same value to a variable are commuting because no matter in which order the assignments are executed the variable will still hold the same value. *Example*

Write accesses are further categorized in *absolute writes* and *relative writes* in SCCharts. Relative writes have the form $x = f(x, e)$ where f is so that such assignments are commuting with each other and expression e does not depend on x , and where the evaluation of e does not have any side effects. All non-relative writes are absolute writes. To schedule different types of writes, the sequential constructive approach organizes non-confluent concurrent variable accesses under a strict Initialize-Update-Read Protocol (IURP). In a tick instance, concurrent absolute writes (initializations) must be executed first as long as they are confluent with each other. Secondly, all concurrent relative writes (updates) can run but must be scheduled after the absolute writes. Finally, all reads may proceed. Note that all non-concurrent access are scheduled sequentially and do not have to adhere to the IURP because there is no concurrency to resolve.

A program which contains concurrent non-commuting write accesses in any tick of the execution is not schedulable under the IURP and must be rejected by the SCCharts compiler. *Conclusion*

2. Preliminaries

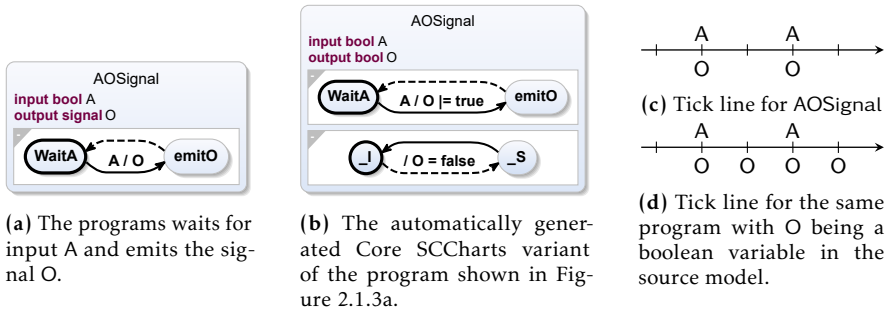


Figure 2.1.3. The Extended and Core SCCharts variants of AO with signals

2.1.4 Extended SCCharts

Extended SCCharts include features which can be expressed in Core SCCharts but provides more convenient ways for the modeller. It is also sometimes referred to as the *syntactic sugar* of SCCharts.

Example Figure 2.1.3 shows two variants of AO, presented in Figure 1.0.1, with signals, which are an extended feature in SCCharts. The first version in Figure 2.1.3a uses the extended SCCharts syntax. Output O can directly be declared as signal. Hence, O is only true in the ticks in which also A is true. Otherwise, O is set to false as can be seen in Figure 2.1.3c, because it is a signal. Figure 2.1.3d visualizes the behaviour of AOSignal if O in the source model would have been declared as standard boolean variable.

Eventually, the Extended SCChart is transformed into a semantically identical Core SCChart during compilation. The result for this particular extended feature is depicted in Figure 2.1.3b. One can see that the output O is now a boolean type. In every tick, O is explicitly set to false and the emit of O got transformed to a relative write assignment. As explained in Section 2.1.3, the IURP takes care of the concurrent scheduling and makes sure that the initialization happens before the relative write access.

SCCharts does not reinvent the wheel when it comes to the features of synchronous languages. Many extended features are borrowed from related languages, such as Esterel [GR83], SyncCharts [And96b], SCADE [CPP17] and Quartz [Sch09]. The complete language overview is depicted in Fig-

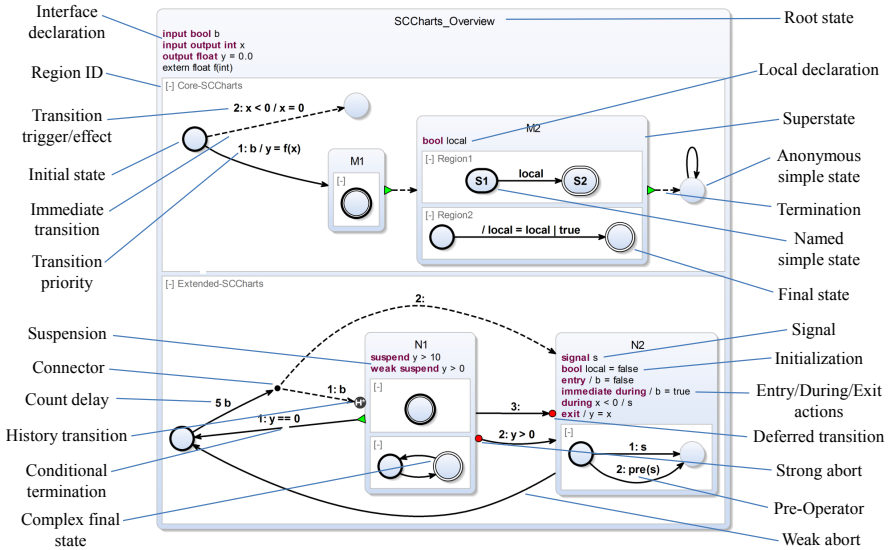


Figure 2.1.4. SCCharts feature overview

ure 2.1.4. Since Motika [Mot17] gives a full overview over all common extended features and their transformations, they are not discussed in detail here. However, the meanings of the most common extended features, which are also used during this thesis, are briefly recapitulated:

Strong Abort A strong abort preempts inner behaviour of a superstate. If a strong abort transition is taken, the inner behaviour of a superstate is not executed. Strong aborts are depicted with an red tail circle at the transition in the graphical syntax.

Weak Abort A weak abort grants a superstate its *last wish*. The superstate is allowed to finish its inner behaviour of the actual tick and is then preempted. Weak aborts do not have a tail decorator in the graphical syntax. Conclusively, a superstate transition can be a termination (green triangle), a strong abort (red circle), or a weak abort (no special decorator). Note that different preemption types are not reasonable on simple states, because the different types only differ in the handling

2. Preliminaries

of the inner behaviour. Therefore, simple states should simply be left with unqualified transitions w.r.t. preemption. Syntactically, they are identical to weak aborts in SCCharts.

Entry Action An entry action is executed immediately when the parent state is entered even if the state is immediately left again. The actions are executed in sequential order in which they appear before any internals continue.

During Action As long as the superstate is active, its during actions are executed. They all run concurrently.

Exit Action An exit action is executed when the parent superstate is left. This includes preemption. Like entry action, exit actions are ordered.

Complex Final State A complex final state is a final state that has inner behaviour or outgoing transitions. Hence, a complex final state can execute inner behaviour and can also be left before the parent superstates terminates. It contributes to the join such that the superstate terminates if the all other regions' control also rests in a final state but continues to operate otherwise.

Pre Operator A pre operator can be used inside expressions. It retrieves the value of a variable of the preceding tick. Pre operators can be nested. Hence, $\text{pre}(\text{pre}(x))$ will return the value of x from two ticks ago.

Count Delay A preceding integer n in front of a transition or action trigger is called a count delay. It states that this trigger must happen n times before the transition, resp. action, enables.

The symmetrical handling of entry and exit actions [Mot17] is a break with the SyncCharts semantics, where entry actions are simply abbreviations of similar actions of incoming transitions which cannot be preempted by an aborting transition because the state has not been entered yet. Contrary, exit actions can be skipped by a preemption. However, adding two unambiguous new actions in Section 6.3 will solve this asymmetry and hence, bring the SCCharts semantics in line again with the semantics of SyncCharts.

Remark A frequently asked question is if there is a difference between “SCCharts” and “SCChart” and when to use which. In general, when talking about SCCharts, the language SCCharts is meant as a whole. Consequently, if one talks about *the* or *an* SCChart, they usually refer to a single SCCharts model.

2.2 The KIELER SCCharts Modelling Tool

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is an academic research project which is mainly divided into two parts, namely *semantics* and *pragmatics*. In this context, the semantics area covers modelling of programming languages and model-based compilation of these with a particular focus on synchronous languages and interactive compilation which are suited for safety-critical applications. The pragmatics teams deals with automatic layout, diagram synthesis and usability questions. Especially the development of layout and transient view frameworks, such as the Eclipse Layout Kernel (ELK) [Rüe18; Sch19] and KLighD [SSH13], are the main foci.

As of July 2020, over 100 committers with over 16000 commits alone in the semantics area of KIELER contributed to the project during its lifetime. The KIELER SCCharts Release 1.0 comprises 35 Eclipse plug-ins, which result in over 2.3M Lines of Code (LoC). However, due to an increasingly streamlined workflow, most of this code is auto-generated nowadays. In fact, only 200k – 300k LoC are written manually in Java or Xtend¹. The implementation of the compiler infrastructure, which is discussed in Part I, is known as KIELER Compiler (KiCo).

KiCo is used to realize the SCCharts reference compiler, which consists of over 80 dedicated compilation systems and 165 modular processing units that can be combined to form new compilation systems. The compiler is released as the *KIELER SCCharts Editor*. It enables a user to model, compile and simulate any SCCharts model.

Figure 2.2.1 shows the default simulation perspective of the KIELER SC- *Example* Charts IDE. The editor is located on the left side **1**. Here, the modeller can edit and save the source model. The modelled program is instantly visualized as layouted statemachine in the Diagram view in the middle **2**. The process of instantly and automatically generated, layouted diagrams from source models is called *synthesis*. This area also highlights running simulations.

¹<https://www.eclipse.org/xtend>

2. Preliminaries

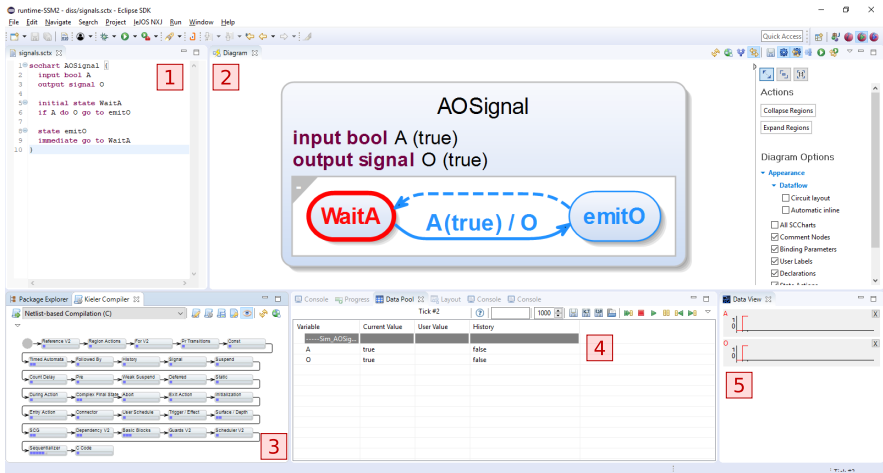


Figure 2.2.1. KIELER SCCharts Editor – Simulation perspective

Synthesis options, which are specialized for the currently active synthesis, can be modified on the left panel in this view. In the lower left area 3, the developer can inspect the actual compilation chain. Intermediate results can be selected and simulated like the source model. The lower middle view 4 hosts controls for the simulation such as value tables for the environment. Here, it is also possible to modify the pace of the simulation or issuing single discrete steps. All components that have been active in this tick are highlighted in light blue. States in which the control rests at the end of the tick are red. On the lower right 5, single values can be inspected. Their history is depicted as a graph appropriate for the corresponding data type.

Remark

It is noteworthy that while being the most complete implementation, the artefact is not a full implementation of all SCCharts possibilities which have been proposed. Additionally, it is an academic project to test and evaluate research. As time goes on, new features will be added and no longer maintained ones will perish. However, the individual release snapshots should remain of course.

2.2. The KIELER SCCharts Modelling Tool

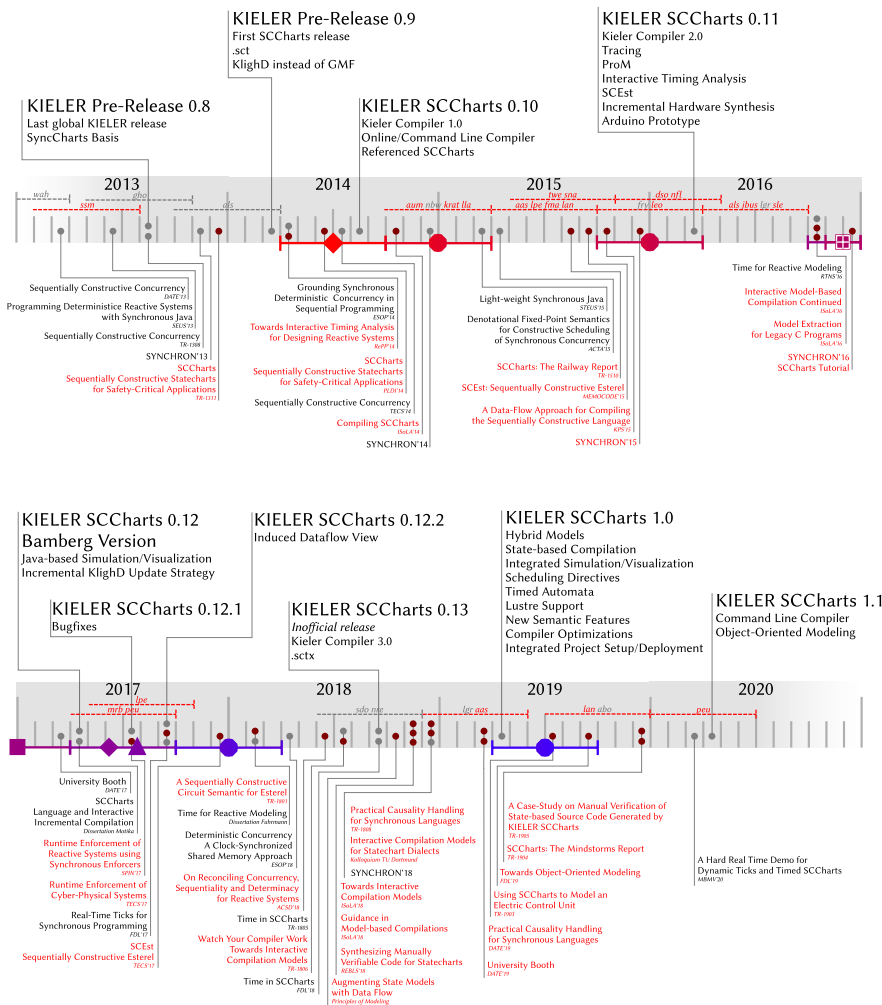


Figure 2.2.2. The KIELER SCCharts development timeline with releases on top, publications on the bottom side, theses inline and conducted surveys on the border. The author's involvement is marked in red.

2. Preliminaries

An overview over the complete SCCharts project is depicted in Figure 2.2.2. The timeline shows the complete SCCharts development from 2013, in which year the last SyncCharts variant of KIELER was released, up until the 2020 with upcoming future releases. My involvement is marked in red. All major releases are shown on the top side of the timeline. They led to the version which was used mainly throughout this book, which is KIELER SCCharts 1.0. The publications related to SCCharts are shown on the bottom side. The theses that are related to the KIELER semantics area are depicted on the timeline and the conducted surveys are shown on the bottom border. A comprehensive list of all my publications and advised theses is listed in Appendix A.

2.3 Related Work

Previous work done by Motika and Fuhrmann is explained in Section 2.3.1. The subsequent sections will cover related topics towards compiler infrastructures, modelling tools, model-based compilation patterns and synchronous languages.

2.3.1 SLIC and Interactive Timing Analysis

This section discusses the results from Motika [Mot17] and separates them from the contributions of this thesis. There has also been joint work on the topic of Interactive Timing Analysis [Fuh17], which can be seen as a demonstrator for the pragmatic and interactive modelling approach.

Relevant ongoing related work towards SCCharts, e. g. research towards object-orientation in statecharts by Schulz-Rosengarten and sequential constructiveness in dataflow languages, such as Lustre, by Grimm, are also not part of this thesis but will be sketched out in future works in Section 8.2.

SCCharts—Language and Interactive Incremental Compilation Motika finished his PhD thesis [Mot17] in 2017 and was the former team leader of the KIELER semantics team and the mentor of this thesis' author. As his list of major publications shows [Mot17, p. 10 ff.], he heavily contributed towards

the design and high-level compilation of SCCharts. This section clearly differentiates the contributions of this thesis to the work done previously. Motika presented first steps of an interactive and incremental compilation approach named Single-Pass Language-driven Incremental Compilation (SLIC). It is a model-based compilation approach which consists out of a series of Model-to-Model Transformations (M2MTs) [MSH14] and is defined by the four letters that form the acronym.

- 1) **Single-Pass** Each transformation in a SLIC schedule is only executed once. Cyclic dependencies between transformations are forbidden.
- 2) **Language-Driven** The transformation invocation order depends on the features of the particular language. They form a dependency graph that describes which transformation *produces* other features and which transformation *cannot handle* other features of the language. Hence, transformations in the SLIC approach are categorized in *produced* and *not-handled-by* transformations. Obeying the first rule 1) of SLIC, the dependencies must form an acyclic graph and there might be more than one valid graph for a specific compilation target.
- 3) **Incremental** In the case of SLIC and due to its model-driven nature, incremental means that every feature of a language gets transformed each by each. Consequently, it is expected that a transformation resolves a specific language feature and that this feature is no longer contained in the model after the transformation is finished.
- 4) **Compilation** SLIC is a compilation approach.

The SLIC approach was an inspiration for the interactive model-based approach presented in this thesis. Chronologically, the SLIC compiler is the predecessor of the KiCo approach. The KiCo development profited greatly from the lessons learned during the SLIC development. Today, KiCo is the predominant approach used in KIELER. Hence, the two main contributions from Motika are not part of this thesis. Particularly,

- SLIC is specifically tailored to the features to the language in question. Produces and can-not-handle dependencies, depicted in Figure 2.3.1a, determine in which order arbitrary model-to-model transformations are executed. The SLIC dependencies between the transformations of the extended SCCharts features from Motika's work are shown in Figure 2.3.1c.

2. Preliminaries

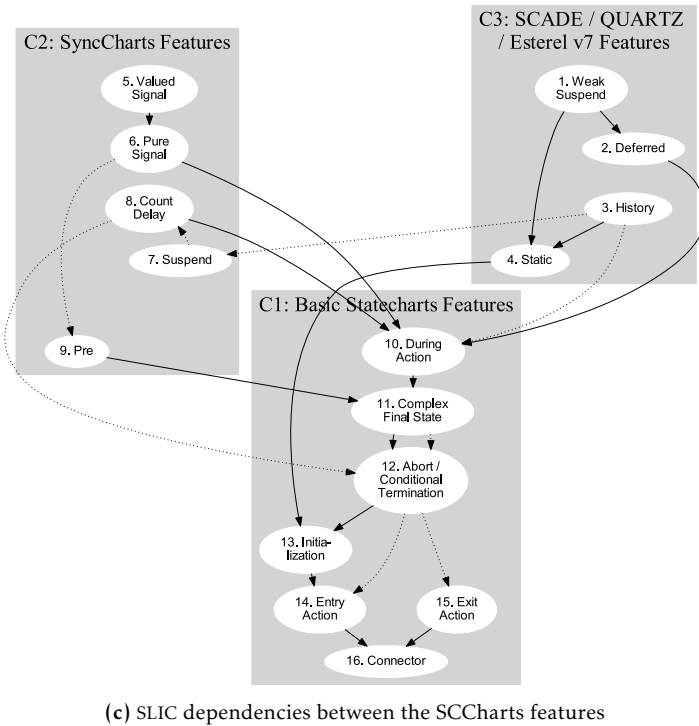
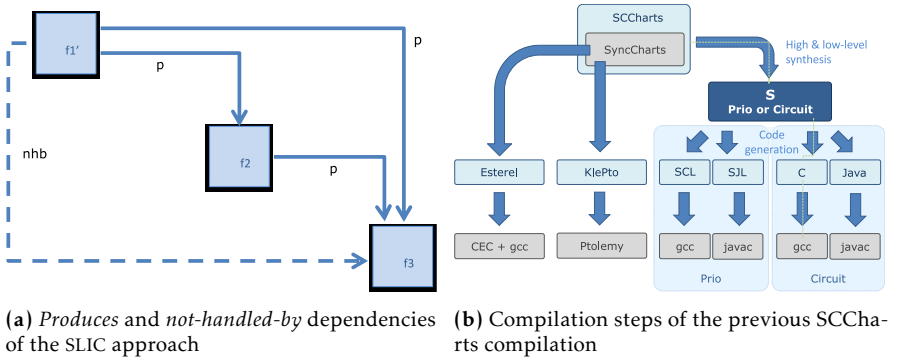


Figure 2.3.1. Previous work done regarding the compilation of SCCharts by Motika (from [Mot17])

2.3. Related Work

The developer has limited influence on the compilation and depends on the correct implemented and configured transformations. Especially, SLIC rules 2) and 3) are tailored to SCCharts and similar languages. However, even within the SCCharts downstream compilation in the SCG, an intermediate control-flow representation for synchronous models, rules 2) and 3) are weakened because features are not really resolved but extended. The SLIC implementation in KIELER does not define or enforce any specifications on the transformations and will only return limited information about the compilation itself. At most times, simply the transformation result will be returned if the compilation was successful.

The interactive model-based compilation approach presented in this thesis takes a step back. It creates a generic framework for models, transformations and measures to define, influence and evaluate model-based transformation chains. In fact, the approach presented here can be used to re-define and re-implement the previous SLIC approach, which is demonstrated in Section 3.4. On the other side, it can also be seen as continuation of the previous work in making the former SLIC definition more generic and easy to use without restriction to already established features. It is not restricted to any language or underlying meta-model and easy to configure on a project-basis.

- The second related part from Motika [Mot17] covers the compilation of SCCharts using SLIC. The work focuses on the high-level transformations from Extended SCCharts to Core and Normalized SCCharts and only sketches out further low-level compilation possibilities, as also already described previously [MSH14; SMH15]. It presented the intermediate language S, which served as common target for the model-based compilation and as source for the final code serialization. The general design is depicted in Figure 2.3.1b.

This thesis does not discuss the high-level transformations of SCCharts covered by Motika. Chapter 5 focuses on low-level compilation approaches using SCCharts as example language. While the general design flow, shown in Figure 2.3.1b, remains untouched for the most parts, the interactive model-based approach makes this process fully modular and not restricted to any meta-models. External components, such as external compilers

2. Preliminaries

(shown in grey in the figure), can be included into the compilation chains to enable in-editor simulation and deployment easily. Furthermore, S is not used anymore by default, because the default compilation systems are configured such that the intermediate results are synthesized in a model representation known by the developer to facilitate understandability. However, if desired, a meta-model for S can be implemented and used for particular steps.

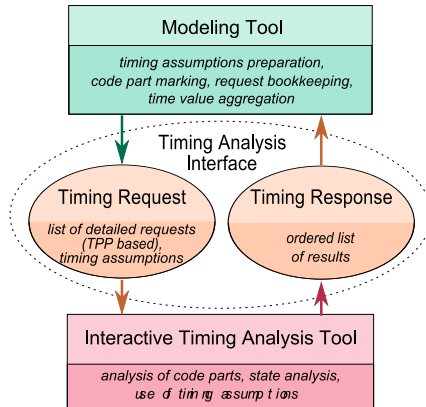
Interactive Timing Analysis Fuhrmann used the model-based approach in her thesis [Fuh17] to facilitate timing analyses. The modeller can fetch timing values from dedicated timing analysis tools interactively while modelling. If building models with safety-critical timing constraints, they are able to identify critical program parts early on in the development. A generalized interface makes the approach feasible for different modelling and timing tools. The overall design flow is depicted in Figure 2.3.2a.

The approach has exemplarily been implemented for SCCharts [FBS+14a] [FBH+16]. Figure 2.3.2b shows an SCChart model within the modelling environment with interactively annotated timing values. Critical regions w.r.t. to timing are highlighted in red. Hence, the modeller receives instant feedback about their timing constraints. Since the Interactive Timing Analysis (ITA) tool had to be partly hardwired in the previous version of the KiCo, it is a good example for the possibilities of the model-based compilation approach. In today's version of the compiler, which is explained in detail in Part I, the whole design flow and annotated visualization can be realized. Nonetheless, ITA is not part of this thesis.

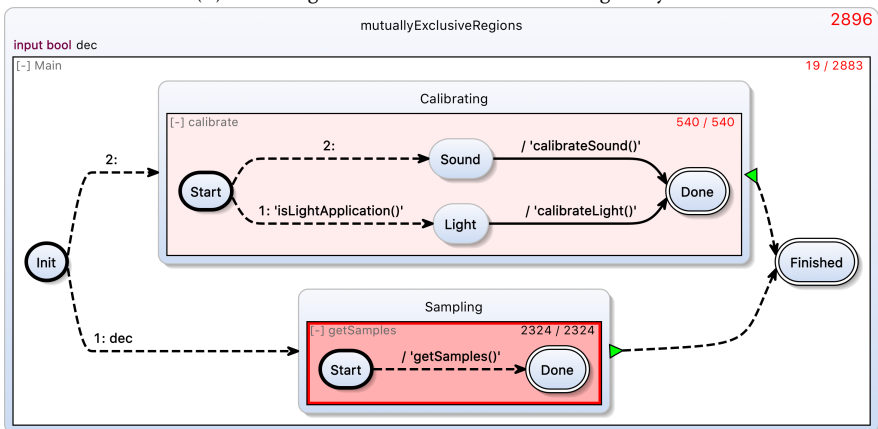
2.3.2 A Modular Compiler Infrastructure—LLVM

LLVM [Lat02] is a modular compiler infrastructure for arbitrary many programming language, with Clang being one of the most prominent ones. Since its beginnings as a research project in 2002, LLVM has grown to an umbrella project for various commercial and open-source sub-projects, such as Clang, libc++ or SAFECode, a memory safety compiler for C/C++ programs. The LLVM architecture, which can be seen in Figure 2.3.3, strictly distinguishes between front- and back-end. Different languages use their

2.3. Related Work



(a) The design flow of the interactive timing analysis



(b) SCCharts example within the modelling environment with interactively annotated timing values and hot spot highlighting for timing critical regions

Figure 2.3.2. Previous work done regarding interactive timing analysis by Fuhrmann (from [Fuh17])

2. Preliminaries

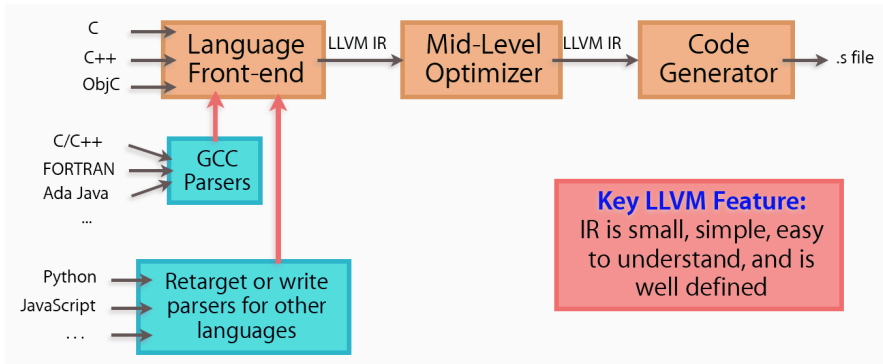


Figure 2.3.3. LLVM Architecture [Lat06]

dedicated front-ends to compile source code, including debug information, to the LLVM Intermediate Representation (LLVM IR). From here modular units can optimize the IR before it is processed further by the code generator and hence, all front-ends benefit from the succeeding optimizations and the code generation.

Lattner distinguishes a compiler from a compiler infrastructure by defining both.

Definition A *compiler* is a tool that inspects and manipulates a representation of programs. A *compiler infrastructure*

- provides *modular and reusable components* for building compilers,
- *reduces the time and cost* to construct a particular compiler,
- allows components to be *shared across different compilers* and
- allows choice of the *right component for the job* [Lat06].

This thesis shows that by this definition, KiCo, as implementation of the interactive model-based compilation approach presented in this thesis, classifies as compiler infrastructure.

Following the SLIC idea, every unit of work of the interactive model-based approach creates a fully functional artefact but defined by the underlying meta-model instead of an IR defined by the compiler. Language front-ends

often try to reach a common IR in modular compiler infrastructures to facilitate modularity. In contrast, the compilation systems in interactive model-based compilations try to stay in their domain meta-model as long as possible to support understandability and readability. This design decision sacrifices some of the cross-language capabilities but provides the modeller with views they understand. Nonetheless, even with this sacrifice, languages which operate in similar domains, e. g. synchronous languages, can profit from cross-language modularity. The approach is presented in Part I. Furthermore, Chapter 5 discusses common compilation techniques and ways how general purpose compilations can also benefit from the approach presented in this thesis.

The LLVM team won the ACM Software System Award, which recognizes *Remark* a software system that has had a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both (across the entire software industry), in 2012². —

2.3.3 Meta-Tool Generators—ETI and CINCO

The close relation between compilation and modelling techniques has been observed quite early by Steffen, who proposes to make use of *consistency models* to detect inconsistencies between different model descriptions and relates this to giving a semantics to a programming language by translation into an intermediate language [Ste97]. The operational behaviour was modelled as temporal or causal models, which fundamentally is a combination of labelled transition systems and Kripke structures [Sti92]. The definition of adequate unifying model structures was paramount here. The trade-off between expressiveness and manageability was resolved by giving priority to *simplicity*, which is also a decisive factor in designing modular modelling systems in my eyes. Steffen's observation of the trade-off between model size and complexity of interpretation can also be found in SCCharts' distinction between Core and Extended SCCharts.

The growing complexity of software and hardware systems prompted the development of the Electronic Tool Integration (ETI) platform. It is an

²<https://awards.acm.org/software-system/award-winners>

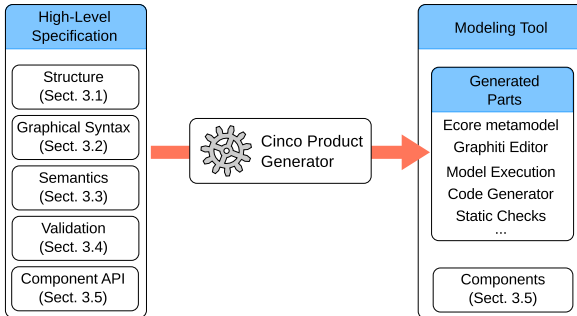


Figure 2.3.5. CInCO generates ready-to-run modelling tools from abstract tool specification (from [NLK+18]).

use *environments*, explained in detail in Section 3.1.3, as general form of inputs and outputs for its compilation units. While an environment is an amalgamation of data, a compilation unit can be written as triple of the form (input environment, process, output environment). Also, compilation units are categorized into specialized work units, which is in line with ETI’s process systems.

To fulfil coordination requests, ETI uses *coordination graphs*, which represent *all*, the *minimal* and the *shortest* solution to these requests. An example for the UNIX universe can be seen in Figure 2.3.4. The highlighted path depicts a solution for the request $gif \ \& \ F \ \{rotate, \ relief, \ display\}$. A similar graph will be constructed interactively for all compilation units available in KiCo in Section 4.2. This compilation systems’ graph represents all possible routes from source to target meta-models and therefore covers all possible compilation paths. Similar to ETI, compilation requests could be realized as service.

Despite the fact that DSLs bring problem solving solutions to domain experts, creating domain-specific graphic modelling tool is often complex and repetitive. To reduce this tedious task, the meta-model generator CInCO was developed. CInCO is a meta-model generator which generates domain-specific graphical modelling tools from specifications. Therefore, as is depicted in Figure 2.3.5, high-level specifications serve as input for the CInCO

2. Preliminaries

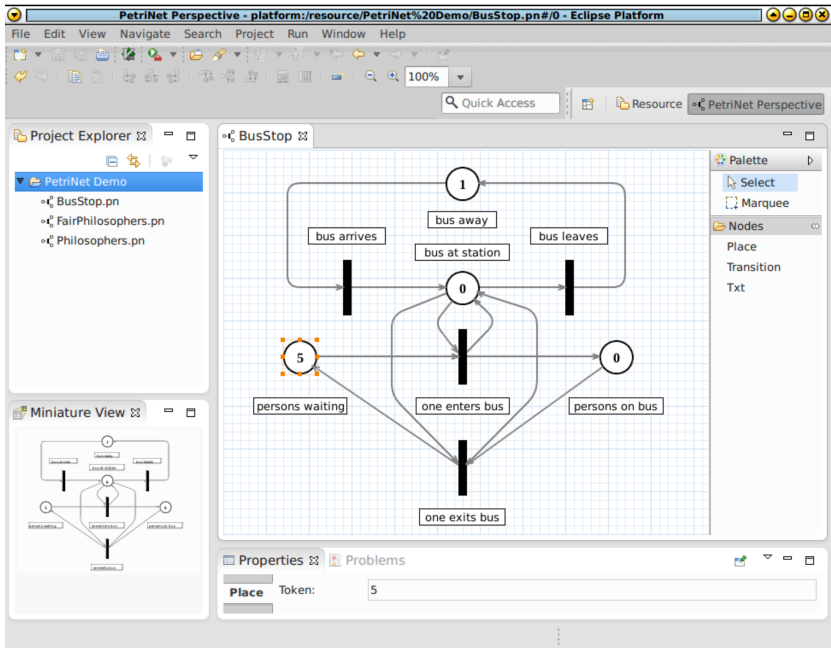


Figure 2.3.6. Petri net modelling tool automatically generated by CINCO (from [NLK+18])

product generator which generates a corresponding graph-based modelling tool fully automatically. As before, simplicity is a central design criterion [MS10]. Universality is traded for intuitive and simple specifications. While building on the modelling capabilities of the Eclipse Rich Client Platform [MLA10] and the Eclipse Modeling Framework (EMF) [SBP+08], technical details are hidden from the user. [NLK+18]

CINCO's meta graph languages specifies what kind of nodes, edges and containers are present in a certain DSL. These can be annotated by style modifiers which influence their looks. Model transformations then add semantics to the graph model. As an example, Figure 2.3.6 shows a fully generated graphical editor for Petri Nets [Pet62].

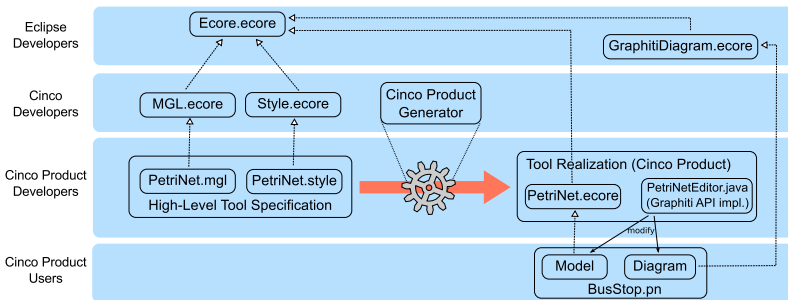


Figure 2.3.7. Developer role layer from CINCO's point of view (from [NLK+18])

The approach presented in this thesis is agnostic towards the meta-models of the compilation artefacts. The K_iCo approach is not a meta-model generator. The generic framework can be used to implement various source and target languages and re-use already existing compilation systems, which makes it a *meta-model engine*. The pragmatic transient view concept enables domain-specific modelling accompanied by instantaneous graphical support. This is an alternative to the graph model approach and attempts to combine the best of the textual and graphical modelling worlds.

Figure 2.3.7 shows the different roles involved with a CINCO product, in this case, the Petri Net example. As said, CINCO builds upon Eclipse, so the top two rows show the roles programming experts have to solve. In the third row, the experts the particular domain are involved. They specify the graph language and generate the modelling tool. Finally, domain experts can use the final product to solve their domain-specific problems.

The interactive model-based approach also separates the target audience in two groups, the domain experts using the compiler and the compiler developers. As the requirements of both groups differ, the approach tries to find a good middle-ground of complexity so that both groups can benefit from each others work-flows. Developers should be conscious of the different groups which are involved in a product. Tools should guide them in order to increase efficiency and should not be overloaded with complex features to appeal to all possible groups. Chapter 4 discusses examples towards guidance in modelling environments.

2. Preliminaries

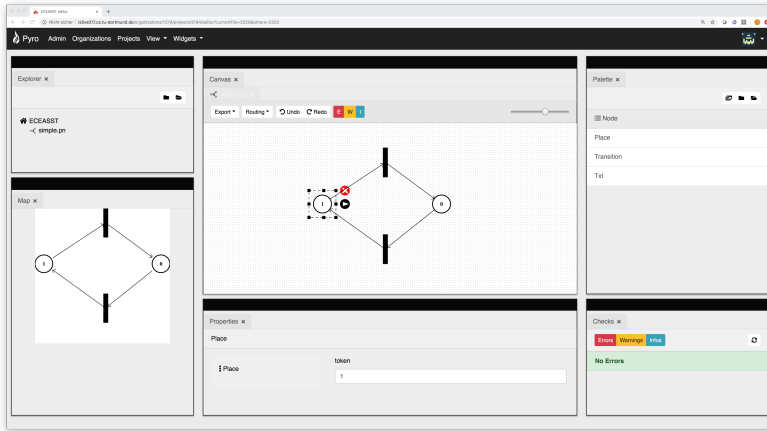


Figure 2.3.8. Web-based CINCO product development (from [Zwe18])

Another development which is becoming increasingly dominant is the Integrated Development Environments (IDEs) and products' shift from classical desktop towards web applications. This is not exclusive to CINCO but can be exemplified here as CINCO can also be used to generate a fully functional web toolkit for its graph specifications. The web version of a Petri net editor generated by CINCO is shown in Figure 2.3.8. The kit called Pyro builds upon the Graphical Language Server Protocol (GLSP) and uses common web framework for its browser support. While being generated by CINCO it is independent from the remainder of its code base Pyro [Zwe18].

The tools' movement towards the web is not part of this work. However, an outlook and more related work with concrete connection to the KIELER product is discussed in Section 8.2.

Remark Despite the fact that the CINCO and the KIELER teams may be working on different meta-levels and are focusing on different pragmatical aspects, I see the spirits of both teams very much aligned. Thereby, I would like to adopt the slogan of the CINCO team “Hard for us, easy for them” and extend it in the following to “If necessary, hard for us—but always easy for them” for the KIELER team.

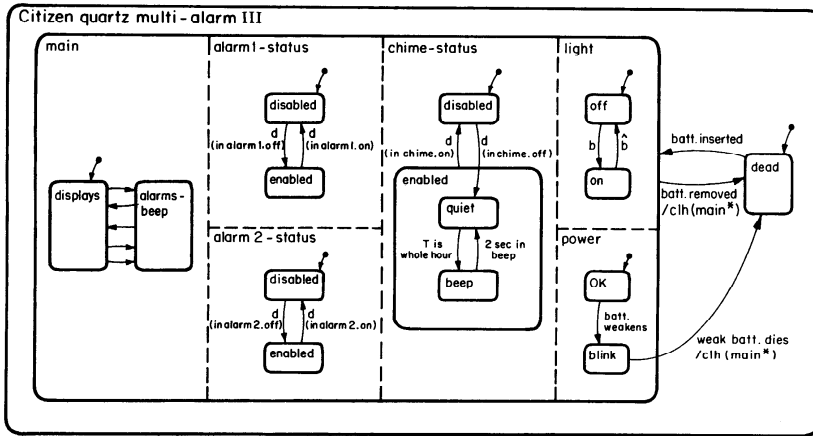


Figure 2.3.9. Part of the Citizen Quartz clock example (from [Har87])

2.3.4 Statecharts

Statecharts, presented by Harel in 1987 [Har87], are a *visual formalism* for describing behaviour. They extend the classical state-transition diagrams with hierarchy, orthogonality and communication. In that they are compact and expressive as well as modular. In fact, they were set out to counter many of the objections people had against visual representations of behaviour at the time of publishing.

Figure 2.3.9 shows a part of the Citizen Quartz clock, the leading example in Harel's initial contribution. Even without further explanation, one can identify states, hierarchy (superstates), orthogonality (different regions separated by dashed edges) and communication via trigger and effects on the transitions. Basically, statecharts are Mealy Machines [Mea55] with hierarchy, concurrency and broadcast communication. Example

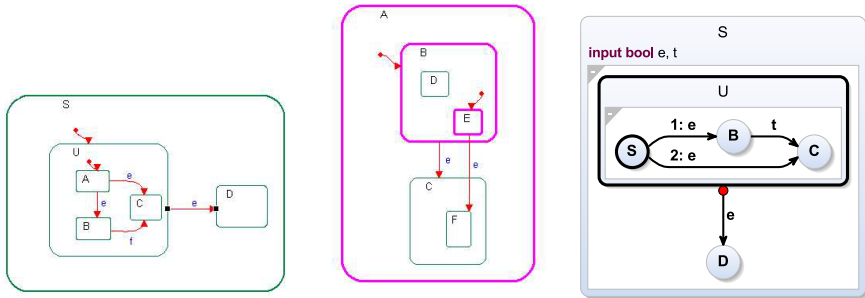
Statecharts shifted the way developers worked. Instead of relying on graphical representations only for guidance while working on textual artefacts, visual languages became the pre-dominant objects for description, verification and code generation – at least in some areas, such as the avionics industry. Statecharts are a *clear* and *precise* visual programming language

2. Preliminaries

which programs can be analysed, compiled and executed [Har09]. Harel reminds to caution because even though many people would agree that a picture is worth a thousands words, not everything can be visualized or depicted in a way that is clear and fitting for the brain. While the interactive model-based approach is agnostic towards the compilation artefacts, VDSLs, e. g. SCCharts, are well-suited for the approach presented in this thesis, since every intermediate compilation step can be presented in a way the domain experts understands.

The initial contribution also encouraged to exploit pragmatic peculiarities of the language, such as *zooming in* on different abstraction levels, which nowadays has become a common feature and is also used in KIELER [SSH13] instead of opening up separate windows for each hierarchy. While the statecharts team strongly believed in the virtues of visual languages, they acknowledged that the visuals can be replaced by their textual or algebraic descriptions. The pragmatics-aware modelling approach tries to combine the best of both worlds, which was also coined *textical modelling* by Motika [Mot17]. The modeller works on a textual description, while inspecting the instantaneously created graphical representation, even though this modelling philosophy is not a requirement for the KiCo approach.

Statecharts sparked a whole family of Statechart dialects, many of which implementing or discarding specific languages features or having semantic variation points. The lack of clear guidance regarding the semantics of statecharts proved out to be an issue; von der Beeck [Bee94] summarized 21 Statechart variations, including the initial semantics proposed by Harel et al. [HPS+87] and their differences in 19 semantic variation points. The report did not, however, contain the semantics of the original statecharts implemented in STATEMATE [HN96]. One of the more controversial questions was – and is still today in the synchronous community – if the reactions of a system can be sensed within the same tick or at earliest in the following. SCCharts, being a conservative extension to SyncCharts and hence, Esterel, follows the first approach. The latter approach is, e. g. implemented in languages such as ForeC [YGR+16]. Statecharts' standard feature of *inter-hierarchy transitions* is not part of SyncCharts and hence, also not implemented in SCCharts. However, it can be realized on extended level and could be added to the repertoire of the SCCharts language if needed.



(a) Substate transitions override superstate transitions in Rhapsody (from [HK04]).

(b) Transition priorities in Rhapsody are determined inside-out (from [HK04]).

(c) Model from (a) in SCCharts

Figure 2.3.10. Conflicting Transitions in Rhapsody and STATEMATE

Some arguably exotic features initially presented by Harel are *selection entrances*, *parameterized states*, and *recursive and probabilistic statecharts*. An implementation of probabilistic transitions in SCCharts is sketched out in Section D.2. *Hybrid systems*, implemented in KIELER SCCharts via dataflow regions as extended feature, are explained in Section 6.2. A way of model verification via *model checking* is briefly discussed in Section 7.1.2.

I recommend Harel’s notes about the making of statecharts [Har09] for further reading as they contain an interesting (personal) view on the history of statecharts, important design decisions and clearing up some confusion that arose during the years. *Remark*

The first tool built for statecharts was STATEMATE, released initially in 1986 [HLN+90]. Here, statecharts describe component behaviour. *Activity charts*, basically hierarchical dataflow diagrams, are used to model the functional structure of the system. The actual structure of the system was modelled via *module charts*, which specified real components and their connections. Created statecharts could be analysed, used for documentation and served as source for the automatic code generation, which produces code in Ada or C. Being one of the first real model-driven system development tools, STATEMATE can be seen as important step towards the standardized

2. Preliminaries

Unified Modeling Language (UML) efforts for model-driven engineering we have today.

Remark In his notes [Har09], Harel reported that he tried to convince the management of the company that developed STATEMATE to release a cheap or free version of the tool but failed. As the only available serious Statechart tool was very expensive, many smaller companies and teaching facilities could not afford it. Consequently, this may have slowed down the spreading and acceptance of visual languages. With a glance at the steadily-growing open-source community (with all its benefits and flaws) and its new business models, it seems that Harel simply was 20 years ahead of time.

STATEMATE was superseded by Rhapsody in the mid-90's [Har09]. Rhapsody is a visual modelling tool based on object-oriented statecharts with all the usual properties known from object-oriented languages, such as invocation of methods, creation of objects and inheritance. The main components are classes, which can each be associated with a statechart. Whenever a new instance of a class is created, a copy of that statechart is spawned. The action language of Rhapsody is the target language, supporting C++, Java or C. Semantically, due to the object-oriented challenges, Rhapsody differs from STATEMATE [HK04]. Effects are observable immediately in the actual step. Also, a step is not considered to be processed in zero time. Figure 2.3.10 shows the handling of conflicting transitions. It is not allowed to have two transitions on the same level which are eligible to run as depicted in Figure 2.3.10a in the A state, because this leads to non-determinism. On different levels, contrary to STATEMATE, if two transitions are eligible to fire, the lower level transition is taken in Rhapsody to resemble refinement commonly present in Object-Oriented Programming (OOP). Therefore, in the example in Figure 2.3.10a, if A is active and e is received, the control will switch to B assuming the transition from A to C is removed. In STATEMATE, the control switches to D. Priorities are also determined inside-out considering the source state instead of outside-in. Therefore, if E is active in Figure 2.3.10b when e is received, the control is given to F instead of C, which is the case in STATEMATE .

SCCharts, being a conservative extension of SyncCharts (see Section 2.3.5), does not rely on hierarchy ordering for its priorities. Transitions leaving from the same state are strictly ordered by their priority and preemptions from

a higher hierarchy must state explicitly if the inner behaviour is permitted to finish the actual tick or not. These transitions are called *weak* or *strong aborts*. Figure 2.3.10c shows the same model as in Figure 2.3.10a in SCCharts (modelled following the textical approach with automated layout in roughly one minute). The preemption is explicitly set to strong, depicted with the red circle.

The simulation modes in Rhapsody use the same code generator which is used in the production mode. For simulation, the code is enriched with additional instrumentation. Using the identical code eases verification and bugfixing. In KIELER, different means for simulation have been examined. Simulation via model alteration [Mot09], model tracing [Sch14][RSM+16] and automatically annotated code annotations [Eum20] have been implemented within the KIELER SCCharts tools. In particular, the current KIELER simulator was implemented by Schulz-Rosengarten with KiCo, which is the KIELER implementation of the interactive model-based compilation approach presented in this thesis. Section 5.5.5 gives further information on how to use this concept for simulation benchmarks. Section 8.2.2 gives an outlook towards runtime debugging of statecharts dialects.

The STATEMATE team won the ACM Software System Award³, which recognizes a software system that has had a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both (across the entire software industry), in 2007. The award announcement reads: Statemate was the first commercial computer-aided software engineering tool to successfully overcome the challenges of complex interactive, real time computer systems, known as reactive systems. The ideas reflected in Statemate underlie many of the most powerful and widely used tools in software and systems engineering today. *Remark*

2.3.5 Control-Flow-Oriented Languages—Esterel and SyncCharts

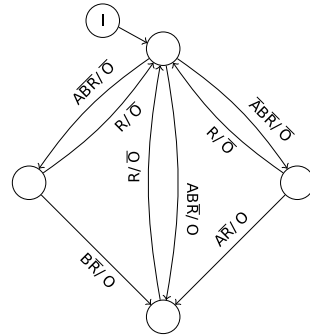
One of the first languages which employed the synchronous hypothesis (see Section 2.1.1) was Esterel [BC84], presented by Berry in 1984. It is an

³<https://awards.acm.org/software-system/award-winners>

2. Preliminaries

```
1 module ABRO:  
2 input A,B,R;  
3 output O;  
4 loop  
5   [await A || await B];  
6   emit O  
7 each R  
8 end module
```

(a) ABRO in Esterel



(b) ABRO as Mealy machine

Figure 2.3.11. The “Hello World” of synchronous languages – ABRO

imperative programming language with rigorous semantics [Ber02], which enable determinism. As one of the first synchronous languages, Esterel often serves as inspiration or foundation for new languages in this field. For example, SC [Han09a] and ForeC [YGR+16] fuse synchronicity with the C programming language, and synERJY [BPS06] and Blech [GG18] combine synchronicity with object-oriented flavours. Nowadays, what is sometimes referred to as the *classical* synchronous MoC, is the underlying MoC of Esterel.

Figure 2.3.11 shows the *Hello World* of synchronous languages – ABRO. In Esterel, as depicted in Figure 2.3.11a, the listing has three inputs, A, B, R, and one output, O. In the main loop, the program waits for A and B in parallel, indicated by the double pipe (||). Afterwards, it emits the signal O. The loop is reset every time R is present.

The program can be converted into a Finite State Machine (FSM), which is helpful to explain its semantics. Figure 2.3.11b shows the corresponding Mealy Machine [Mea55] of ABRO. Nodes in the machine represent the states of the program. Transitions are taken if the trigger before the slash is valid. A bar on top of an input indicates absence. The signal after the slash in the transition label depicts a signal emission. Hence, the left and right states are reached if either A or B is present. The bottom state is reached after both, A and B, occurred without being reset by R. This also triggers the emission of

2.3. Related Work

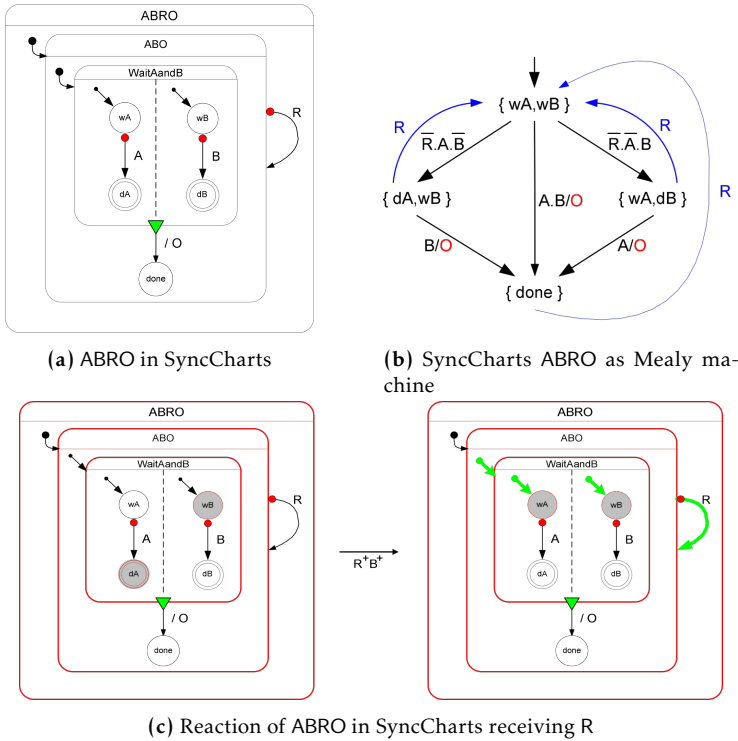


Figure 2.3.12. Combining statecharts with Esterel: SyncCharts (from [And95])

O. Regardless of any other signal being present, emission of R will reset the automaton to the initial state, which resembles a hard preemption.

SyncCharts [And96a] merges the Esterel semantics with the previously discussed statecharts formalism (Section 2.3.4). It was presented by Charles André in mid-90's. Figure 2.3.12 shows the ABRO program in SyncCharts. The parallel waiting for the occurrence of A and B in Figure 2.3.11a is now depicted in Figure 2.3.12a with two separated regions with each having two states. The regions start in their initial state, wA and wB respectively. Once the corresponding signal is received, the state switches from wA (wB) to dA (dB). As soon as both regions reach their final state (double border),

2. Preliminaries

the enclosing state is left via its *normal termination* (green triangle) and O is emitted. As before, the whole behaviour is reset if R is present.

The corresponding FSM is shown in Figure 2.3.12b. It is identical to the FSM shown for Esterel’s version of ABRO. Semantically, SyncCharts follows the MoC from Esterel [And03]. However, not all of Esterel’s kernel language is depicted directly. Instead of using traps, for example, state transitions are realized with strong or weak aborts. Also, states can have actions associated with them, e. g. when a state is entered or left. While these examples can be expressed in Esterel, the visual statecharts formalism facilitates them. Transformations from textual control-flow-based languages, such as Esterel, to its graphical counter-parts have been studied further by Prochnow et al. [PTH06] and Rüegg [Rüe11].

Utilizing a visual formalism also enables the intuitive depiction of state changes. The modeller can see first hand what is happening. Figure 2.3.12c depicts a reaction of the ABRO model. Initially, the system is in the dA and wB states. Then, the signals R and B are received, indicated at the arrow between the two model depictions. The transitions that fire now become green and the user can see that the reset transition is taken and O is not emitted. Using this kind of visualization comes natural for model simulation. However, it also can be used to show intermediate results and compiler annotations between M2MT as is shown in Chapter 3 of this thesis.

Since SCCharts is a conservative extension of SyncCharts, every SyncCharts model is also valid in SCCharts. However, there are some carefully decided but sometimes subtle differences. Firstly, immediate transitions, that are transitions which are eligible to run in the same tick as their originating state was entered, are depicted dashed and are not indicated by a hashtag (#) prefixing transition labels as in SyncCharts. In the SCCharts syntax, delay behaviour is a property of the transition itself and does not need a transition label or trigger to be depicted. Secondly, the asymmetrical handling of entry and exit actions w.r.t. preemption is resolved in SCCharts. There are preemptable and non-preemptable versions of both actions in SCCharts (explained in Section 6.3), so the modeller can decide, which actions suits the model best.

2.3.6 Dataflow-Oriented Languages—Lustre and SCADE

The two main ways with which a program flow can be described are by its control-flow or by its dataflow. While a form can be described in its counter-part, languages often tend towards the one or the other. Esterel and SyncCharts are control-flow-oriented languages. The control of the program rests at specific places, named states, until emitted events fire state changes. In dataflow-oriented languages, the modeller writes down equations, which model the relations between the data of that program. The arguably most prominent synchronous dataflow language is Lustre [CPH+87; HCR+91].

Listing 2.3.1 shows the ABRO program in Lustre. A Lustre program consists of nodes. The main data objects are streams, which can be used to combine data from the actual or previous ticks. In the example, the EDGE node models if a certain state has been reached. The arrow operator (->) is an initialization: In the first tick, take the first stream. Proceed with the second stream subsequently. In the main node ABRO, O is set to true if seenA and seenB are true. seenA and seenB are calculated in the rows below. They are initialized with false at first and, accordingly to the behaviour of ABRO, set if A (resp. B) occurs and reset at R.

Compared to its control-flow-oriented cousin in Esterel, there is some modelling needed to encode the behaviour of ABRO in pure dataflow. However, there are other examples, e. g. circuits or computation heavy controllers, in which it comes natural to describe the behaviour in dataflow, and modelling in control-flow-oriented languages becomes cumbersome. Lustre also supports a way to specify automata, which are then translated to clocked dataflow during compilation. A way to add dataflow functionality to SCCharts to facilitate the modelling of such programs will be discussed further in Section 6.2.

Built upon a dialect of Lustre is SCADE [Dor08]. The ability to create deterministic dataflow models upon a rigorous semantics with a graphical syntax makes SCADE one of the state-of-the-art tools for safety-critical domains, such as the avionics industry. It includes SyncCharts-like automata called *Safe State Machines* [CPP05]. To satisfy the high safety-critical requirements of, e. g. the avionics industry, the compiler is officially certified [CPP17]

Figure 2.3.13 shows the user interface of the SCADE suite. It contains

2. Preliminaries

```
1 node EDGE(X:bool) returns (Y:bool);
2 let
3   Y = false -> X and not pre(X);
4 tel
5
6 node ABRO (A,B,R : bool) returns (O : bool);
7   var seenA, seenB : bool;
8 let
9   O = EDGE(seenA and seenB);
10  seenA = false -> not R and (A or pre(seenA));
11  seenB = false -> not R and (B or pre(seenB));
12 tel
```

Listing 2.3.1. Possible ABRO implementation in Lustre

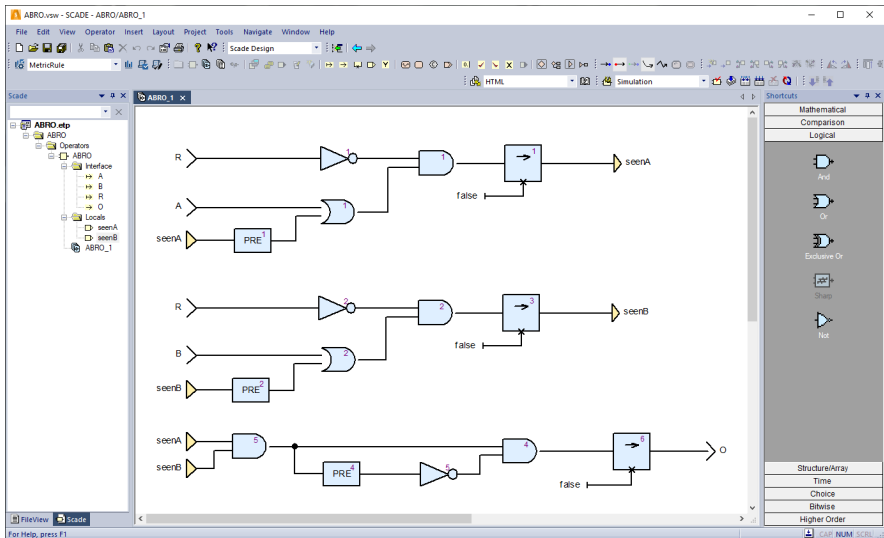


Figure 2.3.13. SCADE Suite user interface showing ABRO

standard IDE features, such as a project explorer and tool palettes. The working area in the center of the applications shows the Lustre ABRO in graphical SCADE syntax. Semantically, both programs are identical. SCADE programs can be exported as Lustre equations. Grimm [GSS+20][Gri19] showed how to form complete Lustre programs from these exports and how to subsequently convert these programs to SCCharts. In SCCharts, being a control-flow-oriented language in its core, Section 6.2 shows how pure dataflow aspects can also be expressed in control-flow models. Through this, programs can be expressed as control-data-flow *hybrid models* in SCCharts.

2.3.7 Compilation Approaches for Statecharts

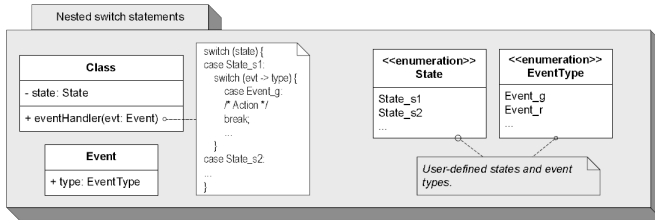
Pintér, Majzik [PM03] and Samek [Sam02] give overviews over common statecharts compilation techniques. Figure 2.3.14 shows four different design patterns. A common approach is the implementation of states via *nested switch* statements (Figure 2.3.14a). Switches are used to select the active state and execute its inner behaviour. The behaviour of a state can also be implemented via a switch statement by testing different events, such as transition triggers. There are two main drawbacks with this method. Firstly, the structure, including hierarchy, of the original statechart is often not recognizable. Secondly, the manually generated code which follows this pattern is often difficult to maintain.

Figure 2.3.14b shows the structure of *action-state tables*. Action-state tables store pointers to functions of the events of a state. While this approach is slightly faster than the previous, it requires more memory, which is often unused because most of the table pointers are unused. Also, the structure of the original statechart is not represented.

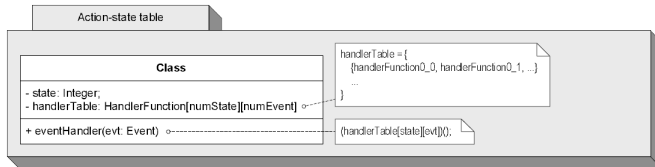
Following the *state design pattern* for object-oriented languages [GHJ+95], descendants of a common interface implement concrete states, depicted in Figure 2.3.14c. The actual state is stored in a reference and updated on transition change. There is no explicit support for structural features, such as hierarchy or concurrency. For example, the state pattern was used by Allegrini to generate code from UML state machines in the ArgoUML⁴ CASE tool [All02].

⁴<http://argouml.tigris.org>

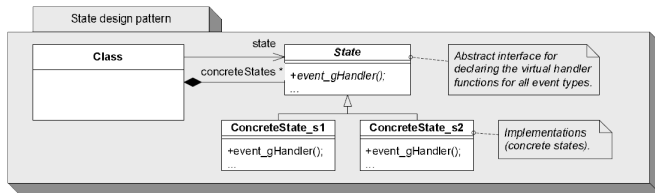
2. Preliminaries



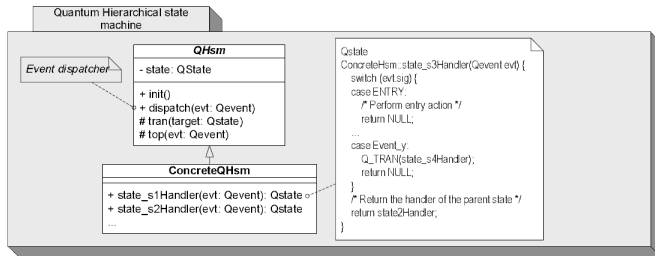
(a) Nested switch pattern for statecharts



(b) Action-state pattern for statecharts



(c) State pattern for statecharts



(d) Quantum hierarchical state machine pattern

Figure 2.3.14. Different design patterns for Statechart code generation (from [PM03])

The structure of Samek's *quantum hierarchical state machine* [Sam02] is depicted in Figure 2.3.14d. Quantum state machines improve the state pattern by adding hierarchy support. Events not handled by the concrete state implementation are delegated to their parent state. The pattern does not fully implement the UML model as actions associated to transitions must be implemented as entry or exit actions. Further, concurrency is not supported, but a sketch is given by Samek. Pintér and Majzik extend the quantum state machine by adding support for transition actions, shallow history and most concurrent operations [PM03] by explicitly adding these structures to the interface. Concurrent composites states are disassembled into individual state machines with no immediate communication.

Ali and Tanaka [AT00] demonstrate a translation of UML statecharts into Java code. Here, subclasses which implement the behaviour of states are declared and instantiated with a context of the model. The context serves as interface for the model's actions and holds the actual state. All behaviour of a state is contained in one class, which makes additions easy. However, in the context of this book, the readability of this approach is arguably difficult. The original structure of the statechart is difficult to reconstruct due to necessary wrapper objects.

For synchronous languages, Potop-Butucaru et al. [PEB07], Edwards and Zeng [EZ07] extensively explain various different compilation approaches for compiling Esterel and similar languages. Several compilers for Esterel exists. Two of the more prominent ones are the INRIA⁵ and the Columbia⁶ (CEC) Esterel compiler. Figure 2.3.15a shows different compilation flows of the INRIA compiler. Esterel serves as input and is then compiled into intermediate code, which is transformed further into C code using one of the compilation approaches. The possible approaches cover expansions into classical FSMs, netlists and Graph Code (GRC), a control-flow expansion developed by Potop-Butucaru [Pot02]. GRC serves as intermediate representation in the CEC. An example translation is depicted in Figure 2.3.15b. The Esterel module on the left side is represented by its GRC on the right. As GRC is a control-flow expansion, the active parts of the program are selected at the

⁵<http://www-sop.inria.fr/meije/esterel/esterel-eng.html>

⁶<http://www.cs.columbia.edu/~sedwards/cec>

2. Preliminaries

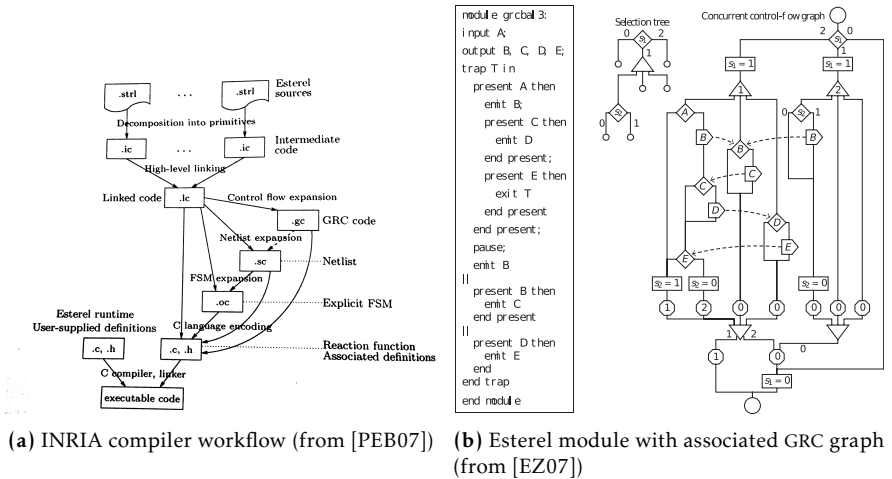


Figure 2.3.15. Different code generation approach for Esterel

beginning of the graph. The first approach of the CEC converts the GRC into a PDG [FOW87] using the Static Single Assignment (SSA) form [CFR+91]. Edwards and Zeng show how to reorder Esterel programs to create sequentialized code. The second approach creates dynamic lists which hold instruction sets that can be executed without context switch. The generated code is only faster for selected examples due to the dynamic overhead, but only program parts that are truly active are executed. The third compilation strategy of the CEC is optimized towards code size. It generates code for a tiny virtual machine, which provides an instruction set with little overhead but supports lightweight concurrency, signals and exceptions.

In SyncCharts, the unit of reaction is called a ReactiveCell [And04], as depicted in Syncchart’s meta model in Figure 2.3.16. Conceptionally, the cells run concurrently and compute their reactions, which rely on the reactions of their components. A reaction may be stalled until new *facts* about the presence of signals are broadcast. Analogously to Esterel, if there exists pending evaluations without defined signal statuses at the end of an instance, the program is considered *non-constructive* and the SyncChart has

2.3. Related Work

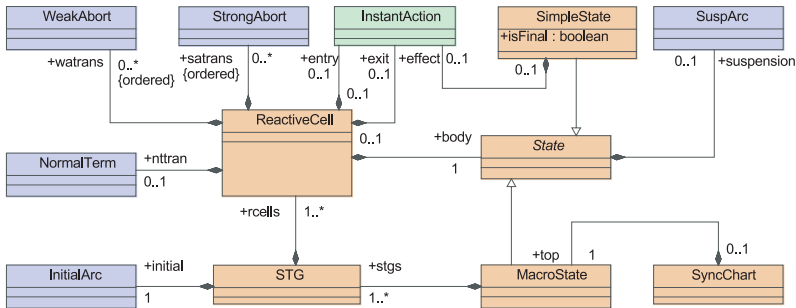


Figure 2.3.16. SyncChart’s meta model (from [And04])

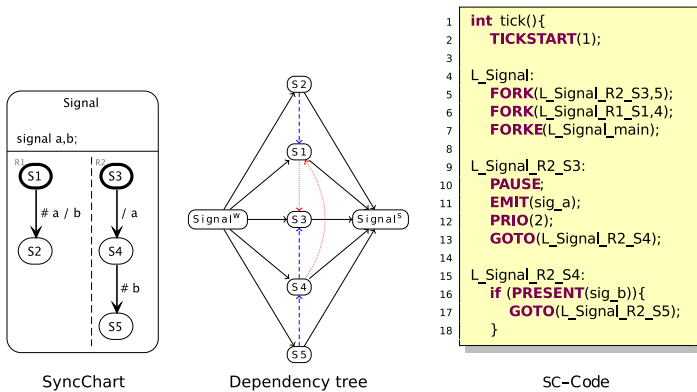


Figure 2.3.17. SyncCharts to SC translation (from [Ame10])

to be rejected. A commercial version of SyncCharts is implemented in Esterel Studio, which effectively merged into SCADE [CPP05].

Amende [Ame10] describes translation rules for SyncCharts to the SC language [Han09b], a lightweight synchronous macro-based extension to C. While the structural translation of a SyncCharts program is done straightforwardly, concurrency is solved by assigning static priorities to the target code. Effectively, a light-weight scheduler implemented by the macros executes the program according to the pre-calculated priorities.

2. Preliminaries

An exemplary translation is depicted in Figure 2.3.17. The Signal model has two threads, R1 and R2. The control flow of the threads depend on each others signal emission. S1 waits for a, which is emitted by the outgoing transition of S3 and S4 is waiting on b, which is emitted if S1 transition to S2 is taken.

The dependencies can be seen in the dependency tree. They determine where in the code a PRIO statement has to be inserted. Köser extended the priority-based compilation by giving synchronization approaches for multi-core architectures [Kös10]. Amende also gives optimization rules for the generated SC code. While some, such as dead code elimination, can be achieved by standard code optimizing techniques, others, such as optimizing priorities and jumps, are SC-specific. Similar optimizations were later made in the netlist-based compilation approach [Smy13].

Fundamentally, the SC compilation approach laid the foundations for the priority-based compilation, which is discussed in Section 5.3.

Biernacki et al. [BCH+08] presented a formalized modular code generation for synchronous dataflow languages, which is the foundation of the mathematically certified compiler used in SCADE. The code generation uses clocks which express the activation of computations to produce efficient sequential modular code. Equations are statically scheduled according to their data dependencies. To enable modularity, node feedbacks have to be broken explicitly by delays as modularity is not always feasible, even in the absence of causality loops as noticed by Gonthier [Gon88]. After type verification and annotation of expressions by the so called *clock calculus*, the program is translated into a minimal object-oriented intermediate language. The object-oriented nature is only used to describe the state of objects and is not for inheritance or polymorphism reasons. The intermediate representation can be transformed into common general purpose languages, such as C or Java. Clocks are transformed into control structures. A minimal, certified reference compiler was written in OCaml and COQ⁷.

Similar to this approach, the netlist-based compilation strategy of SCCharts, which is explained in detail in Section 5.2, also transforms a partially ordered set into a sequence of assignments. This step is called *sequentializa-*

⁷<http://coq.inria.fr>

tion. While SCCharts is control-flow-based naturally its dataflow extension, see Section 6.2, is first compiled into control-flow SCCharts before processed further. Since SCCharts does not support different clocks by default, all of SCCharts dataflow runs on the global clock.

This section gave an overview over different compilation approaches for statecharts and related languages. Part I explains the interactive model-based compilation, which can be used to model and implement the vast majority of the aforementioned approaches and optimizations into one tool with modern pragmatic modelling features. While several common algorithms and practices can be found implemented in KIELER without any serious changes, other are tailored specifically to the MoC of SCCharts and are explained in Chapter 5. Additional to the compilation of SCCharts, Chapter 6 discusses further language variations and Chapter 7 shows exemplary practical applications.

Part I

**Interactive Model-Based
Compilation**

Model-Based Compilation Systems

*Abstraction is the elimination of the irrelevant
and the amplification of the essential.*
— Robert C. Martin – Agile Principles, Patterns, and Practices in C#

Model-based compilation systems compile source models to desired targets via series of M2MTs. However, these compilation systems can also be modelled to achieve a variety of tasks. This chapter gives a motivation for model-based compilation systems and explains their general concepts.

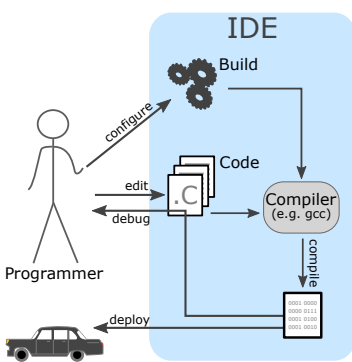
Since every incoming model is considered a source model from the perspective of a M2MT, the source model that is the source of the overall compilation is called Original Source Model (OSM). *Definition*

Section 3.1 motivates the interactive compilation systems presented in this thesis and explains the relationship between programming and modelling from a Model-driven Engineering (MDE) perspective. Section 3.2 introduces necessary terminology w.r.t. model-based compilations. Interactive model-based compilation systems are then introduced in Section 3.3. Section 3.4 discusses derived compilation systems, such as the SLIC approach by Motika [Mot17].

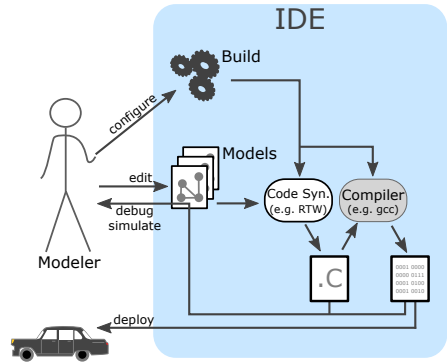
3.1 A Generic User Story

This section takes a closer look at three alternative development processes sketched in Figure 3.1.1. It is assumed that the developer uses an IDE to work on a particular software project. Usually, the build process (or project)

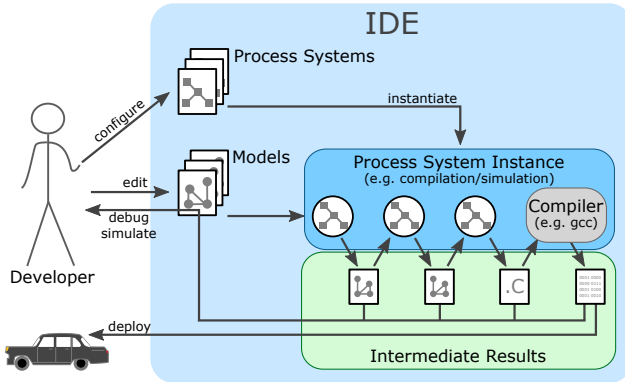
3. Model-Based Compilation Systems



(a) An abstract view on the classical programming development process



(b) An abstract view on the classical modeling development process



(c) Development story with interactive compilation system instances

Figure 3.1.1. Three alternative development processes

has to be configured by either the developer themselves or by another build expert. Typically, the developer in all three approaches works directly on the artefact in question. However, the work foci differ.

3.1.1 Classical Programming

Figure 3.1.1a gives an abstract view on a **classical programming** development process, which is fairly straightforward. The IDE might be the Eclipse CDT¹ or similar. The developer often has to be a programming expert and generally also configures the build process. While they usually work on one file at a time, they must keep an eye on the whole project, which is usually a collection of files, because it might influence the compilation. When they complete a development step, they issue a compilation command. An embedded (often external) compiler then compiles the source files to binary code, which can be executed or embedded elsewhere if the source code is error free. Errors and warnings are fed back to the editor inside the IDE. They mark the erroneous line and give more or less processed information about the actual error or warning.

3.1.2 Classical Modelling

The **classical modelling** work-flow, depicted in Figure 3.1.1b, looks quite similar. The modeller has to configure their project and can explore the project's files. Instead of editing a text file, the modeller usually works on a domain-specific, often graphical, model. The IDE, which may be something like Eclipse or a classical modelling tool, such as Matlab/Simulink or Ptolemy, uses an integrated code synthesis, such as the Real-Time Workshop (RTW) in Matlab/Simulink, to synthesize code. Similar to the classical programming paradigm, as soon as a development step is finished, the source models are compiled to a classical, general purpose language, such as C. Afterwards, they are compiled to binary code like before, with the addition that the user feedback often includes some sort of simulation. Here it depends on the concrete design choices if the simulation runs inside the IDE or on the compiled product.

Although the development processes are quite similar, there is a subtle shift in the focus on the developer. In the first case, the developer has to be a programmer, whereas the models in the second case are typically maintained by a domain expert. However, even in the second case programming experts

¹<https://www.eclipse.org/cdt>

3. Model-Based Compilation Systems

are sometimes required to aid the modeller with special requirements or IDE tool extensions.

3.1.3 Interactive Compilation Systems

Figure 3.1.1c depicts the interactive model-based compilation approach proposed here. With *interactive compilation systems*, operating procedures, such as compilation or simulation, can be created and modified by the developer. In fact, these procedures are modelled and their corresponding compilation systems are simply models just like the working artefact but perhaps models of another meta-model. When the modeller wants to compile (resp. simulate) the actual status of the model, the respective compilation system gets instantiated. Afterwards, the issued command can be processed by that system's instance. The feedback of the compilation includes error messages, *intermediate* and *final results*. These are directly available as individual model instances of the appropriate meta-models. They can be inspected by the modeller by the same views which are used to inspect manually created models or serve as source for further compilation systems.

In the figure, we see an instantiated compilation system. The artefact is processed sequentially by single processors, e. g. model-to-model transformations, of the instance. All intermediate steps are observable. Eventually, the user wants to deploy binary code. The second to last of the intermediate results may be general purpose source code, e. g. in C, which can be sent to an external compiler as before. Conceptually, the compiler call is just another process in the sequential chain of the system. Its result is a new intermediate result of the whole compilation system.

Note that this approach is agnostic to the question whether the intermediate results (or, in fact, the OSM) are graphical or textual. If the syntax is graphical, transient views and automatic layout technologies are a key enabler to represent (intermediate) artefacts of perhaps different meta-models instantaneously. KIELER makes use of the ELK to synthesize the views. This is an example of *pragmatic modelling* concepts [FH10; HLM+12], which aim to enhance modeller productivity by allowing to seamlessly switch between textual and graphical representations tailored to specific use cases.

The interactivity of the approach becomes apparent in the ability to

3.1. A Generic User Story

observe all intermediate steps, to run system instances as they are needed and to create new or change existing systems. There is no need to go through long re-build and re-start cycles, which are often necessary in classical tools when the project requirements or settings change, as these steps can all be performed during run-time. A compilation system can basically perform any kind of job. As an example, the figure depicts a Model-to-Model Compilation (M2MC). Technically, the term *interactive* subsumes the dynamic nature of the approach, meaning that instances of systems are generated dynamically as they are needed. These instances carry dynamic properties on their own and exist as long as they are required. This also resembles the classical class-object hierarchy of the object-orientated paradigm.

Another take on this is to view compilation systems as a—rather abstract—data flow model. Traditionally, data flow models are collections of *actors* which consume and produce data [LNW03]. Conceptually, the processors of a compilation system correspond to actors, and the data they consume and produce correspond to the intermediate results generated along a synthesis chain. One difference is that in compilation systems, each actor typically fires only once, and the schedule which governs how the compilation system is executed is rather simple, usually just a single sequential execution of all processors. However, more complex, dynamic execution schemes are also possible, as is explained in Section 3.3.5. Alternatively, if one wishes to focus on the schedule of the processors, the compilation systems can be seen as control-oriented state machines, where one processor can be active at a time, and when it is finished, control advances to the next processor. This view can be helpful for example to define more elaborate schedules, but hides what is actually produced and consumed by the processors, which is why I consider the data-flow analogy typically more fitting than the state-oriented analogy.

Due to the interactivity of the approach, tool developers and domain experts can easily create, explore and modify different aspects of the whole development process. The difference is not disparate work-flows but the diverse work-flow artefacts which are being worked on. Figure 3.1.2 shows the different layers of models and the two main roles of users. On the left hand side, the domain expert mainly works on the system's input, e. g. a particular model in a specific DSL. The model's meta-model also belongs

3. Model-Based Compilation Systems

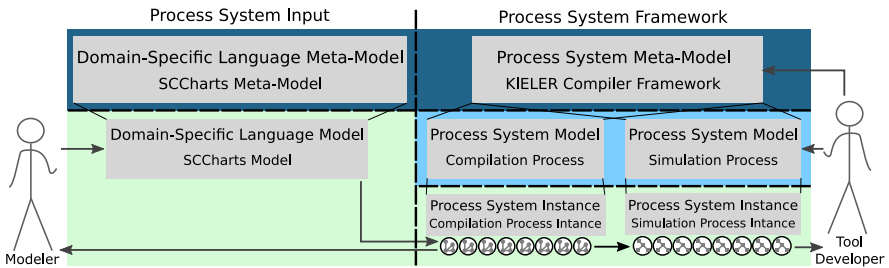


Figure 3.1.2. Different model layer and user roles of interactive compilation systems

to the system’s input but is usually outside of the modeller’s scope w.r.t. making changes during a particular project. In the example in the figure, the modeller works on an SCCharts model, whose syntax is defined in the corresponding meta-model.

On the framework’s right hand side, there is also the framework’s meta-model for defining system models. Derived from this, different systems can be created which hold the necessary instructions. These systems can be instantiated to be applied on a specific artefact. In the example shown, the created SCCharts model is fed into an compilation system instance. During compilation, several observable intermediate results are created. The result of the whole context also serves as input for a simulation instance.

In general, the domain expert will be more interested in the actual project’s model and the systems’ results, whereas the tool developer’s focus will lie on the systems and the underlying framework, including the relevant meta-models. However, both can utilize all aspects of the development process to drive their work. For example, the domain expert may also change a particular system to toggle optimizations if necessary. More obviously, the tool developer can use different model inputs to test and extend the framework. This leads to closer feedback loops between domain experts and tool developers.

Remark Note that these systems were called *interactive process systems* in the original contribution [SSH18c], because the term *compilation* usually implies some kind of transformation in computer science. However, since specialized systems were already sometimes called compilation systems if they

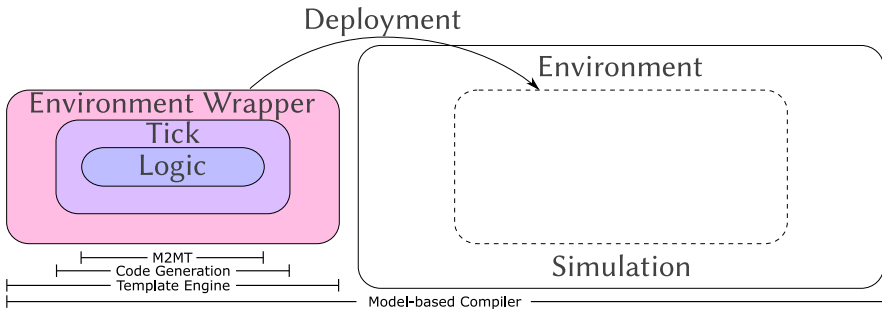


Figure 3.2.1. Different parts of a model-based compilation

transform a model, and because *process system* is also an overloaded term, the general term for these systems, transformative or not, proposed here is now simply *compilation system*.

3.2 Modelling Foundations

Models are used frequently. However, depending on the field, people use models for various things. Lee distinguishes between *scientific models* and *engineering models* [Lee17]. He clarifies that scientists and engineers ask different questions when working with models. A scientist asks “Can I build a model for this system?” whereas an engineer asks the opposite “Can I build a system for this model?”. While this book will not go into the philosophical aspects of models, this section will introduce the concept of models as they are needed in the model-based compiler. However, as a developer using such a compiler usually wants to compile a model to a specific target, models of the approach proposed here can be seen as engineering models.

3.2.1 Layered Terminology

During compilation, the target artefact gets constructed in layers. Different parts of the compiler are responsible for the layers. To keep them apart, the layer names shown in Figure 3.2.1 will be used. The approach uses four

3. Model-Based Compilation Systems

distinct layers.

The OSM will be compiled via M2MTs to the *logic layer*. It contains the pure logic of the source model and can be used to run tick instances in software or to generate hardware circuits. The second layer, the *tick layer*, encapsulates the logic layer. It provides a *reset* function, which resets the status of a particular instance to its start configuration. It also provides a *tick* function. The tick function calls the logic and handles register saves and the setting of status flags, such as the `_GO` flag, which signals that the program/circuit started in this tick. This separation is particularly helpful when creating circuits since the reset and register logic is often identical across different projects and can be ignored at times when only the pure logic is of relevance.

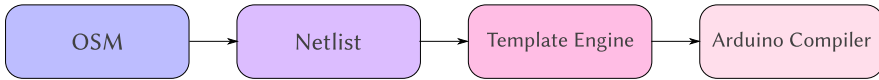
The program often has to be embedded in a greater context, usually called *environment*. An environment can be a dedicated live system, such as an Arduino Uno², or a simulation environment within the IDE. Regardless of the specific target, the environment determines how the tick function is called. In this context, the most common methods of invocation are *periodic*, *event-based* or a combination of both. The environment will give inputs to the program and the program has to return its outputs back to environment. These steps are done on the *wrapper layer*. Usually, for every target environment, the necessary steps are the same even across different models, so that these tasks can be modelled in templates and therefore can be processed easily by appropriate template engines.

Eventually, the wrapped program has to be deployed to its target. Even though this is often done in one step, multiple steps, e. g. separated upload and execution, are imaginable. This layer is called *deployment layer*. All tasks of the different layers can be performed modularly with the same model-based compiler framework, as will be demonstrated in the following sections.

Example Figure 3.2.2 gives a complete example of a compilation from an ABRO program to the deployment to an Arduino Commercial Off-The-Shelf (COTS) board. Figure 3.2.2a depicts the concrete layering steps for ABRO according to Figure 3.2.1. The OSM ABRO is compiled with the netlist-based compilation

²<https://www.arduino.cc>

3.2. Modelling Foundations



(a) Compilation workflow

```
1 scchart ABRO {
2   input bool
3     @pin "7" @pullup @invert AButton,
4     @pin "6" @pullup @invert BButton,
5     @pin "5" @pullup @invert RButton
6   output int @macro "Delay" delay = 100
7   output bool
8     @pin "4" ALED = false,
9     @pin "3" BLEED = false,
10    @pin "2" OLED = false
11
12  initial state ABRO { ...
```

(b) ABRO Arduino declaration in the OSM

```
1 void setup() {
2   pinMode(5, INPUT_PULLUP);
3   pinMode(6, INPUT_PULLUP);
4   pinMode(7, INPUT_PULLUP);
5   pinMode(4, OUTPUT);
6   pinMode(3, OUTPUT);
7   pinMode(2, OUTPUT);
8   reset(&model);
9 }
```

(d) Arduino setup and loop functions generated by the template engine

```
1 arduino --upload --board arduino:avr:uno --port COM5 kieler-gen.ino
```

(e) Deployment via external call to the Arduino compiler

```
1 #include "ABRO.h"
2
3 void logic(TickData* d) { ... }
4
5 void reset(TickData* d) { ... }
6
7 void tick(TickData* d) { ... }
```

(c) logic, reset, and tick functions generated by the netlist-based compilation approach

```
10 void loop() {
11   model.RButton = !digitalRead(5);
12   model.BButton = !digitalRead(6);
13   model.AButton = !digitalRead(7);
14   tick(&model);
15   if(model.delay > 0) {
16     delay(model.delay);
17   }
18   digitalWrite(4, model.ALED);
19   digitalWrite(3, model.BLED);
20   digitalWrite(2, model.OLED);
21 }
```

Figure 3.2.2. Compilation workflow example – ABRO for Arduino

approach, which is explained in detail in Section 5.2.4, to C code. The template engine wraps the generated logic into a framework the target platform can understand. Finally, the external Arduino compiler is used to upload the model onto the hardware board. Annotations, prefixed with an “at” (@), in the SCCharts model declaration in Figure 3.2.2b, tell the

3. Model-Based Compilation Systems

template engine how to wire model inputs and outputs to the hardware. The pure logic, which is not influenced by the wiring to the concrete target platform, is generated by KiCo. The concrete function signatures are shown in Figure 3.2.2c. The complete compilation result of a similar model, including optimizations, will be shown in Section 5.2.8. Subsequent to the generation of the tick function of the model, the template engine uses the annotation information to set-up the correct pin modes and wires the input and outputs to the actual hardware. In the example, Figure 3.2.2d shows the concrete Arduino .ino code for ABRO. Eventually, the generated source is sent to the Arduino compiler, which generates a binary for the selected board and uploads the program for execution. All these steps can be modelled and implemented within the same compilation framework.

3.2.2 Programs, Models and Diagrams

There is sometimes a controversy on what the difference is between programming and modelling [MS18]. Some may see programming as the art of writing a textual program in a classical programming language, such as C or Java. In the context of model-based compilation, if there is a distinction between the two, it is merely a matter of abstraction. A program is a model. It gives a set of abstract instructions that are to be executed by a target, e. g. a computer.

In this context, when one speaks about *a model*, they usually mean an abstract description of a problem solution, or for that matter, a program. As an example, *abstract models* do not bother with technical details of a particular target system. The wrapper or deployment layer should handle this. In the following, there is no hard distinction between a (classical) program and a model instance of any DSL or meta-model, as both are eligible source and also target models of M2MTs.

Example Both can in fact be combined or seen as the same. Figure 3.2.3 shows variations of the introductory AO example from Figure 1.0.1 on page 5. Figure 3.2.3b shows a C version of AO. In KIELER, SCCharts are usually modelled in textual form by the developer. The textual language of SCCharts is called

3.2. Modelling Foundations

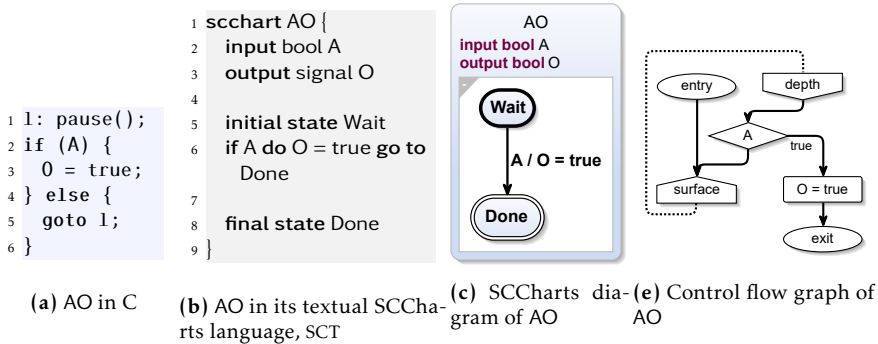


Figure 3.2.3. Different views of the same program, AO: C, Textual SCCharts Language (SCT), SCCharts, Sequentially Constructive Graph (SCG)

SCT and can be seen in Figure 3.2.3b for AO. The diagram, Figure 3.2.3c, is then generated instantaneously automatically while editing. Eventually, if the program gets compiled, the user can also inspect the control-flow graph. The SCG for AO is depicted in Figure 3.2.3e. These variations can be representations of the same model. However, for facilitating different tasks, a model, whose structure is given by its meta-model, can be transformed to another model, which can be of another meta-model.

Different representations of the same model are called *views*.

— Definition

A *meta-model* is an abstract description of a model and therefore defines the structure of all models of this meta-model. Conversely, the meta-model is also a model in defining the structure of its models. From the perspective of a model, the meta-models' meta-model is called *meta-meta-model*.

Definition

For example, while the textual and graphical syntax of SCCharts share the same model as basis in KIELER, SCGs have their own meta-model to ease the downstream compilation.

Example For the AO example, Figure 3.2.4 shows different model layers. The representation of the models corresponds with the ones depicted in Figure 3.2.3. There are four different views for three models, where each has its own meta-model. Notice that while C and the SCCharts model can both be se-

3. Model-Based Compilation Systems

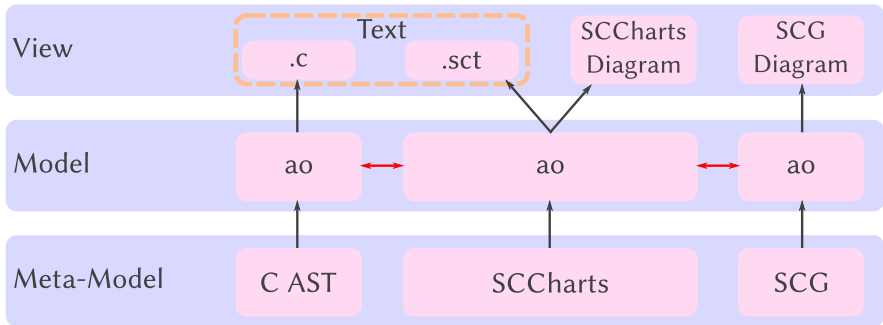


Figure 3.2.4. Model layer of the AO examples in Figure 3.2.3

rialized to text, the grammar of both serializations is different, so it is not really the same “text”. This might be important when generating editors and IDE features for these grammars. Nonetheless, from a pragmatic point of view, the views both show the model in textual form. The SCCharts model can also be displayed graphically. There is no need for a new model here, because all models of the AO program are semantically identical. The model instances can be transformed between the different meta-models without loss of information.

A closer look at SCCharts’ representation of classical C programs, which can be seen as an abstract model to a classical textual program, is discussed in Section 6.5.

3.2.3 Roles

Besides different usages of models, different user roles sometimes do not receive the kind of attention they deserve in academia. In the generic framework proposed here for model-based transformations, there are primarily two user groups which will use the framework, namely the *domain experts* and *tool developers*. Domain experts use the finished product to model solutions to their domain-specific problems, whereas tool developers create the framework and strive to improve the modelling capabilities and efficiency of the domain experts. Both groups have distinct needs with regards to the

3.2. Modelling Foundations

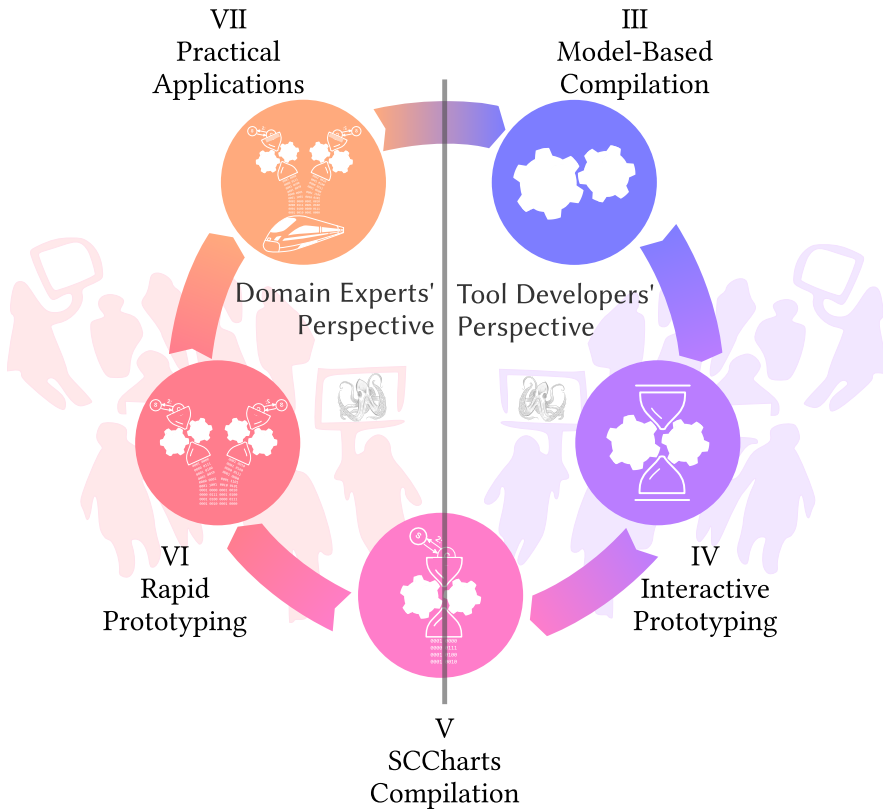


Figure 3.2.5. User roles' focus w.r.t. the chapters of this book

tooling, which should be covered by the framework without disrupting the efficiency of the other group.

With respect to the outline introduced in Section 1.1 on page 9, the focus of interest of each group is shown in Figure 3.2.5. The tool developers are assigned to the right chapters III, IV, and V. The domain experts are assigned to chapters V, VI, VII. Both groups can profit from the experience of the others group, as discussed in Section 3.1.3, however, in general the tool developers will concentrate on matters of the framework further down the

3. Model-Based Compilation Systems

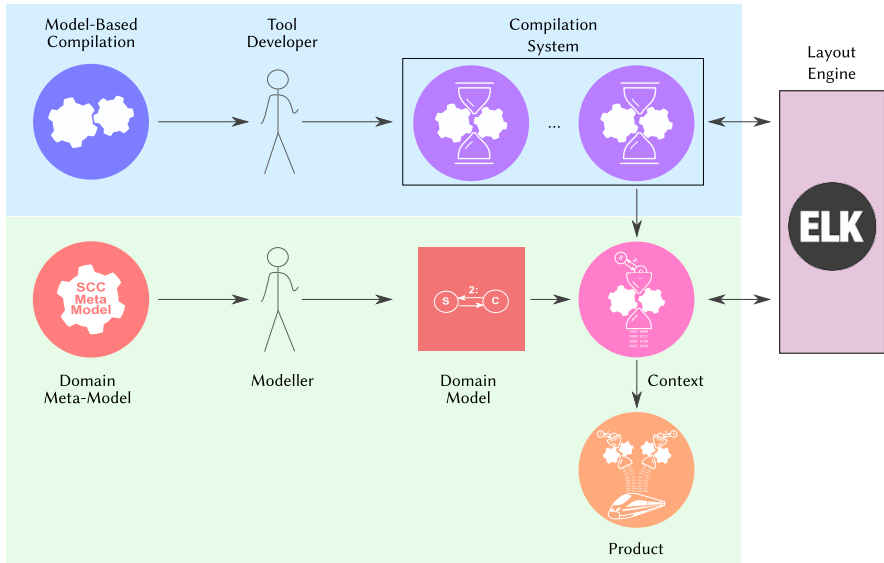


Figure 3.2.6. Different role's interactions with the framework

stack, whereas the domain experts will focus on the high-level issues.

Figure 3.2.6 shows how the two roles interact conceptually in the interactive modelling approach. The tool developers use the compiler framework to create new compilation systems. Modellers create models of specific meta-models for concrete tasks and use the previously crafted systems to generate solutions. Both groups use state-of-the-art pragmatic frameworks, such as ELK, to create transient views, which facilitate different modelling procedures. It might also be beneficial to differentiate tool developers further into *tool* and *framework developers* to enable more domain-specific use-cases, which is discussed in Section 8.2.1.

3.3 Interactive Compilation Systems

This core section explains the overall concepts of model-based compilations. Section 3.3.1 introduces the atomic, modular work units, *processors*, and the differentiation between processors and their instances, analogously to the object-oriented paradigm. Afterwards, Section 3.3.2 discusses different types of processors. Section 3.3.3 explains how these processors can be linked together to form a meaningful *compilation systems* and their instantiation. Section 3.3.5 then introduces two strategies for non-linear compilation systems. Their compositions are explained in Section 3.3.4. Subsequently, Section 3.3.6 will discuss different approaches for ambiguity w.r.t. available paths within transformation chains and how interactive transient views can be used to instantly gain feedback on a system will be shown in Section 3.3.7.

3.3.1 Processors and Processor Instances

The smallest compilation unit is called a *processor*, usually denoted P . \mathbb{P} is Definition the set that holds all processors. —

Although the main task of processors are M2MTs, the specific term *transformation* is avoided to emphasize that a processor does not have to perform a transformation. Instead processors are categorized into *transformer*, *optimizer* and *analyzer* to specify their role. A variety of tasks can be implemented as processors, such as M2MT, optional optimizations and, e. g. object counting. The role of a processor should be restricted to its atomic task to facilitate modularity and reuse.

A *processor instance* p is an instantiated processor $P \in \mathbb{P}$ which received Definition dedicated *source* and *target environments*. The instance is defined as triple $p = (E, P, E')$ with E, E' being arbitrary but defined data storages. —

A processor instance receives input data from the associated source environment. It works on the associated target environment in which it can also store data for subsequent processor instances. Conceptionally, the developer is free to choose the nature of their environments. In KIELER, typed but arbitrary data storages are used. Hence, processors may store arbitrary ancillary data in their environments but have a form of type-safety when accessing it.

3. Model-Based Compilation Systems

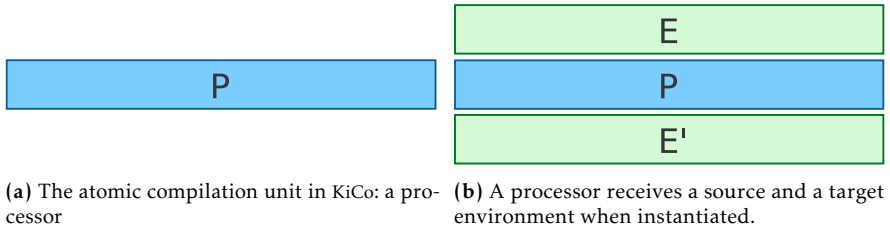


Figure 3.3.1. Processors and processor instances

Figure 3.3.1 shows the previous definitions graphically. A processor, shown in Figure 3.3.1a, is the *blueprint* for a specific task, e. g. implemented as a class programmatically by the tool developer. The processor instance, shown in Figure 3.3.1b, represents an object of this class during execution. The instance can work on data stored in its environment. Hence, as common for object-oriented designs, instances of the same blueprint can work on different compilations due to the separation of instruction and data contexts.

3.3.2 Processor Types

As indicated in Section 3.3.1, KiCo uses the more generic term of *processor* to describe its atomic work units. However, depending on the semantic task, a categorization can be helpful. I here propose four different categories.

Meter Processors can be used to measure aspects of a definite model. They can be used as post-processor to measure the quality of the transformation, e. g. by counting model elements.

Definition A *meter* is a measuring processor that calculates a measure $m \in \mathbb{R}^+$ for a specific feature of a model. The measure is stored in the target environment of the processor. Values of $m < 1$ indicate an improvement over the source model, whereas $m > 1$ measures the contrary.

Example In EMF-based models, which are common in Eclipse, the overall count of model elements can be retrieved with $c = \text{rootObject.eAllContents.size} + 1$. A measure to grade transformations can then be calculated with $m = \frac{c_{\text{target}}}{c_{\text{source}}}$.

3.3. Interactive Compilation Systems

Since meters have access to the whole environment of a transformation and do not necessarily have to judge model elements, a normalization to 1 might not always be reasonable. More generally, if two measures $m_1, m_2 \in \mathbb{R}^+$ belonging to two transformations t_1, t_2 exist, t_1 performed better than t_2 if and only if $m_1 < m_2$.

An example for grading the performance of a transformation based on meta information in the environment is a meter which gives a measure m for the execution time of the processor via $m = 1 + \frac{t_{processor}}{10^9}$, with $t_{processor} \in E'$ storing the execution time of the processor in *ns*. *Example*

Transformer A transformer is a processor which alters the source model in some way.

In an *endogenous transformer*, the meta-model of the source and target model are identical, whereas the meta-models differ in *exogenous transformers*. *Definition*

Optimizer Optimizers change models to improve efficiency without changing its semantics. They are a special form of endogenous transformers w.r.t. to the model quality. If an optimization is not successful, the unmodified source model can be used as result.

Graded by a measure m , an *optimizer* improves a model w.r.t. a model aspect. An optimization is effective if $m < 1$ holds. *Definition*

Analyser An analyser does not change the original model. Its task is to gather information about the model and to store it in the environment. Thus, measures m of analysers should always be $m = 1$. Meters are also a special kind of analyser.

For example, a loop analyser searches for loops in the control-flow graph of a model and stores the information in the target environment. Subsequent processors can pick up this information and use it for decision making. A detailed example of this behaviour is shown in Section 5.2.3, where this synergy is used to cure schizophrenic SCCharts models in KIELER. *Example*

3. Model-Based Compilation Systems

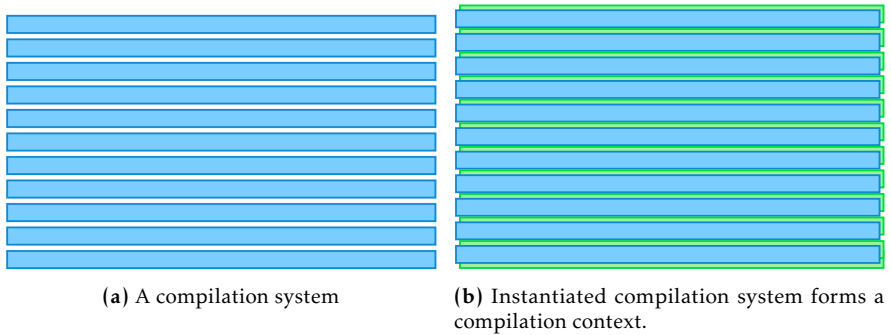


Figure 3.3.2. Compilation systems and compilation contexts

3.3.3 Compilation Systems and Contexts

Definition A list (or directed graph) of processors forms a *compilation system*.

A compilation system describes a single compilation from a certain source type down to the desired target. The system can be instantiated to perform a task for a definite source artefact. The instantiation creates an instance for each processor in the system. As explained in Section 3.3.1, each processor instance will receive dedicated source and target environments when instantiated. In the system, the source environment of a processor instance will be the target environment from its predecessor. The first environment in the system is the *start environment* of the whole system holding the system's configuration. The last environment of the system will hold the compilation result. Analogously to Figure 3.3.1, Figure 3.3.2 shows this relationship for systems and contexts. From an object-oriented point of view, the compilation (Figure 3.3.2a) system serves as blueprint for the compilation and the instantiated system (Figure 3.3.2b) can be seen as a single compilation context for a concrete compilation at run-time.

Definition An instantiated compilation system is called *compilation context*.

Example The netlist-based and the priority-based compilation approaches for SCCharts [HDM+14] were introduced before the development of KiCo and implemented statically. Both constitute examples of compilation systems and are nowadays modelled in KIELER. They are explained in detail in Chapter 5.

3.3. Interactive Compilation Systems

The simplest system possible is shown in Figure 3.3.1b. It consists of a single processor with its corresponding source and target environments. Once the system is fully instantiated, a new *compilation context* exists, which can be used to compile an artefact. A context, including all environments with all data, is observable during compilation. Further, it will remain accessible even after the compilation is finished until discarded so that all data and results can be inspected for as long as is desired.

The developer does not have to bother with all the instances and environments. The KiCo framework will do most of the work. In general, when invoking a compilation programmatically, one only needs two lines of code. A context has first to be created. The context needs to know which system model it should use and on which artefact the compilation should be invoked. Once the context has been created, the compilation can be executed. Listing 3.3.1 shows an excerpt from the KIELER project where a compilation is started asynchronously as soon as the user presses a particular button. The programmer could make adjustments to the context before the compile method is executed, but it is not necessary in this case. Implementation

```
1 val context = Compile.createCompilationContext(view.activeSystem, model)
2 context.compileAsynchronously
```

Listing 3.3.1. Compilation invocation excerpt from the KIELER project

Usually compilation systems include more than one processor to perform linked tasks. Figure 3.3.3 shows how processors interact with their environments to orchestrate the entire compilation. As described before, once a context has been created, it needs an input artefact and can be configured if necessary. The first environment in the context receives the *start configuration* ① as can be seen in the figure. After the invocation of the compilation, the first processor begins its work. It fetches the model from its source environment ② and begins its process, e. g. a transformation. That model is the *source model* from the processor's perspective. While working on the model, the processor can create several snapshots of the current intermediate state and store them in its *target environment* ③. More on different types of snapshots is discussed in Section 4.1. These intermediate states can be inspected during or after the compilation. At the end of the processor's

3. Model-Based Compilation Systems

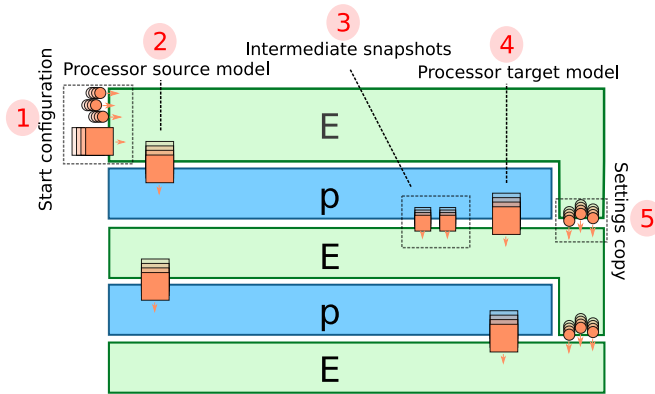


Figure 3.3.3. Concept of a compilation context with two processors.

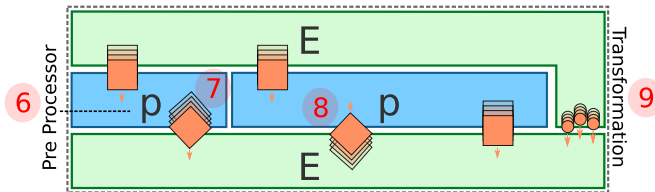


Figure 3.3.4. To save resources, several processors can be grouped together. Generally, everything which happens between two environments is commonly called a transformation.

job, the result is saved ⁴. In the example, the shape indicates that the result is of the same meta-model as the source model. However, any type can be used. E. g. , as targets are often other programming-languages, the backends usually give simple text as results. Once the processor terminates, the next processor starts its job. From its perspective, the former target environment now becomes the source environment and the processor can work on the next one. The framework manages that all settings, model references, and additional auxiliary data get copied to the new environments if necessary ⁵.

To facilitate modularity and consume less resources, processors often perform pre- or post-processing jobs for transformers without the need of

3.3. Interactive Compilation Systems

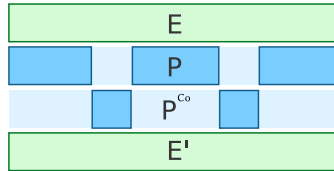


Figure 3.3.5. Schema of a co-processor relationship

dedicated environments, as depicted in Figure 3.3.4. Hence, a processor can run within the same environments as another one ⁶. In the example, the job saves a second model with a different meta-model (indicated by a different shape) in the environment ⁷. This secondary model may store ancillary data (e. g. loop information from a loop analyzer), which can be picked up by subsequent processors ⁸. Usually, what is commonly called *transformation* is everything which happens between the source and the target environments ⁹. The result which is stored in the last environment represents the result of the whole compilation.

The triple $(E, (P_0, \dots, P_n), E')$ of n processors, which all work in between *Definition*
the same pair of source and target environments, is called a *transformation*. —

There is one further type of processor that is not directly depicted in *Definition*
the sequential model of the compilation systems: the *co-processor*. A co-processor is a compilation unit, which is neither a pre- or post-processor, which works within the environment of another processor. It is usually invoked programmatically. —

There are repetitive tasks that should be encapsulated in distinct units to *Implementation*
facilitate modularity. However, their invocation is sometimes tightly bound to conditions of the calling processor. Hence, at the moment these processors can only be called programmatically in KiCo. —

It is an open question if more complex control structures in the compilation system model would allow a more abstract modelling of these transformations. At least in the KIELER project, the need did not arise so far. A visualization of co-processor relations is still be useful for overview and documentation reasons. A real-life example of co-processor usage is discussed in the dataflow extension of SCCharts in Section 6.2.

3. Model-Based Compilation Systems

Implementation

Note that pre-, post- or co-processors also store their data in the target environment of the main processors as they are not permitted to change the source environment. However, the developer is not required to handle these inputs differently as the framework will ensure correct accesses. In KiCo, for example, processors internally always only work on the target environment. The framework automatically creates a copy from the source environment before a processor is called.

To enable even more resource-saving, a compilation can be set to work *in-place*. Compiling in-place does not create new model instances to work on. The processors all work on the same models, hence intermediate results are only observable during compilation and only one at a time. Only the final result remains at the end of the compilation. Conceptually, this would also look like the schema in Figure 3.3.4, where only two environments exist and all processor instances live in between. While this restricts the possibility to inspect the intermediate compilation steps after the compilation, saving these resources might be necessary in large projects. However, retrieving intermediate representations one at a time and saving these models explicitly is still an option and an improvement over classical compilations.

3.3.4 Compilation System Composition

As previously described in Section 3.3.1, a processor is an atomic unit of work. Usually, a compilation requires more than one step to reach the desired compilation target. This is achieved by compilation systems in the interactive model-based approach as explained in Section 3.3.3. To facilitate modularity, compilation systems can reference other systems to form larger units. According to the features described in the previous section, modular compilation systems must be able to express the following properties:

- 1) **Sequential Execution of Processors** The ordered invocation of processors is mandatory. To execute a compilation, at least one processor must be invoked. However, as stated in the introduction of this section, a compilation usually includes more steps, and therefore, more than one processor is needed.
- 2) **Alternative Execution of Processors** It may be possible during compilation that a certain compilation step can be handled by more than one

3.3. Interactive Compilation Systems

processor. For instance, when compiling high-level SCCharts, there are different mutual exclusive possibilities to transform certain language features, e. g. the abort feature [Mot17]. The developer specifies a default transformation in the former SCCharts approach. The user or the system then may override this default.

- 3) **Processor Grouping** Especially when allowing alternative execution sequences, grouping of processors becomes necessary. This enables the developer to execute more than two processors concurrently. In addition, groups of processors can be named to facilitate overview of the system. For example, in the original SLIC contribution, transformations were grouped together by their origin, e. g. SCaDE or SyncCharts. Even when not working with concurrent processors, grouping processors of similar tasks or origins may become helpful.
- 4) **System References** Organizing systems in meaningful compilation units and their potential re-use facilitates modularity.

Overall, only 1) is necessary to form a minimal transformation system which always transforms given source models to specified targets without any decision-making. Even in this scenario the interactive model-based compilation approach can provide helpful information on the compilation, such as intermediate results and processor performance. Intermediate results help to develop new transformations and to detect errors in existing ones. Users may also inspect the intermediate results to understand the overall compilation and semantics of single features. The total system can be saved as trace to compare it to subsequent tries. However, 2 – 4) make compilation systems powerful enough to decide for the best compilation path if there are alternative possibilities or if a static schedule may not be feasible. Since 3) and 4) are a technical separation as both can be seen as a group of processors, a compilation system can recursively be described by

$$\text{System } S = \begin{cases} P & \text{single processor} \\ P S & \text{processor groups} \\ [S] \mid [S] & \text{alternative groups.} \end{cases} \quad \text{Definition}$$

3. Model-Based Compilation Systems

```
1 public system de.cau.cs.kieler.sccharts.netlist
2   label "Netlist-based Compilation"
3
4 system de.cau.cs.kieler.sccharts.extended
5 system de.cau.cs.kieler.sccharts.core
6 de.cau.cs.kieler.sccharts.scg.processors.SCG
7 post process de.cau.cs.kieler.scg.processors.threadAnalyzer
8 de.cau.cs.kieler.scg.processors.dependency
9 de.cau.cs.kieler.scg.processors.basicBlocks
10 post process de.cau.cs.kieler.scg.processors.expressions
11 de.cau.cs.kieler.scg.processors.guards
12 de.cau.cs.kieler.scg.processors.scheduler
13 de.cau.cs.kieler.scg.processors.sequentializer
14 de.cau.cs.kieler.scg.processors.codegen.c
```

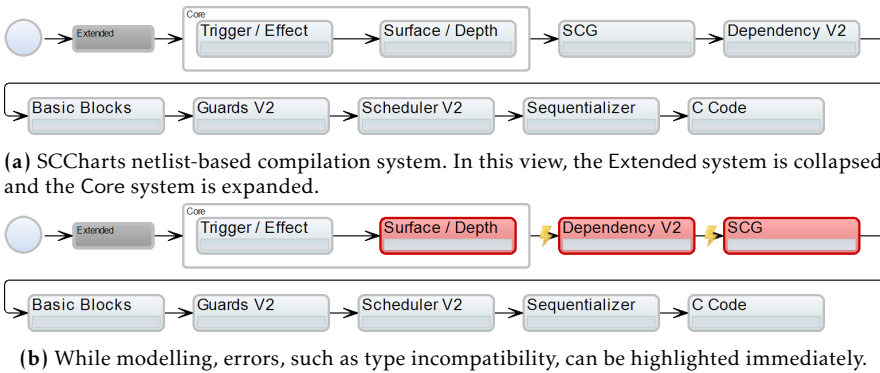
Listing 3.3.2. Model description of the netlist-based SCCharts compilation

Example In practice, systems also often carry meta-information, such as a name and visibility flags for pragmatic reasons. For example, the netlist-based compilation system used for SCCharts, which is explained in detail in Section 5.2, uses 33 processors. Listing 3.3.2 shows a shortened description for this compilation in textual form, which is in fact the way they are stored within KIELER to drive the compilations. Every processor has its own unique identifier. The downstream compilation builds upon the standard high-level SCCharts compilation (line 4–5) and nine further processors identified by their identifiers.

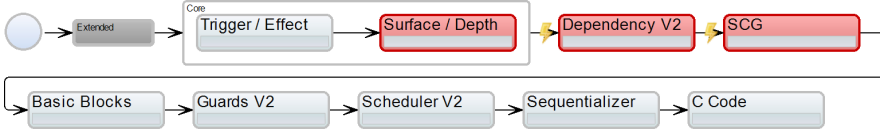
As these descriptions define compilation models (also interactively), concepts such as transient views can be used to visualize the system graphically and, point to problems, such as unknown or type incompatible processors, if necessary. Interactively here means that one can inspect, change and save the model during run-time to invoke altered compilation runs without the need of long re-configure and re-start cycles.

Example Figure 3.3.6a shows the automatically generated graphical representation of the netlist-based compilation system during editing. This view is synchronized with the editor of the model's description and instantaneously re-generated upon change. The referenced high-level SCCharts systems can be expanded and collapsed for inspection. Problems appear in red. The

3.3. Interactive Compilation Systems



(a) SCCharts netlist-based compilation system. In this view, the Extended system is collapsed and the Core system is expanded.



(b) While modelling, errors, such as type incompatibility, can be highlighted immediately.

Figure 3.3.6. Example of an automatically generated graphical view of a compilation system

generated views are also used as control panel in the KIELER project to invoke the compilations and to select intermediate results.

In the example depicted in Figure 3.3.6a, the Surface / Depth processor creates an SCCharts model which is then transformed to its corresponding Sequentially Constructive Graph (SCG), a sequentially constructive variant of a control-flow graph, by the SCG processor. The subsequent Dependency processor expects an SCG as input. If one would swap the SCG and Dependency processors, the compilation chain becomes type incompatible, as depicted in Figure 3.3.6b.

Complete tool example Figure 3.3.7 shows a complete example of a running KIELER instance during simulation. In the SCCharts editor tool, the abstract model is described with a textual syntax [1](#). A graphical view of the model is instantaneously generated by the transient view framework [2](#). The user can further influence the visualization of the presented data via options on the right hand sidebar [3](#). However, these options consist mainly of rather coarse convenience settings to set the current focus to specific points of interests. [4](#) – [6](#) show examples of different information views. These can be configured (and saved per perspective) individually. Together with the transient live visualization [2](#), they resemble the systems and intermediate

3. Model-Based Compilation Systems

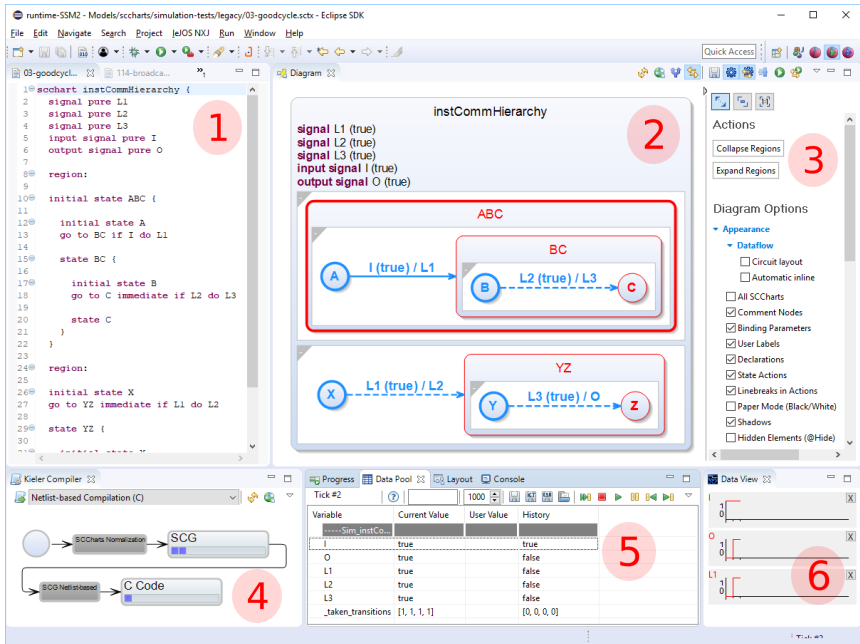


Figure 3.3.7. Complete example of a running KIELER instance during simulation.

result regions from the previous figures. The selected compilation system is depicted in 4. A view to manipulate the running simulation is open in 5. Selected data observers can be inspected in 6. Note that information of the running simulation is visible in the model diagram 2, the simulation view 5, and the observers 6 simultaneously. The variable states and current active model elements can be highlighted directly in the model. The user can input new environment settings in the simulation view. The single forward and backward steps of the simulation can also be controlled. The actual and past data of selected variables can also be visualized in the data observer 6.

3.3. Interactive Compilation Systems

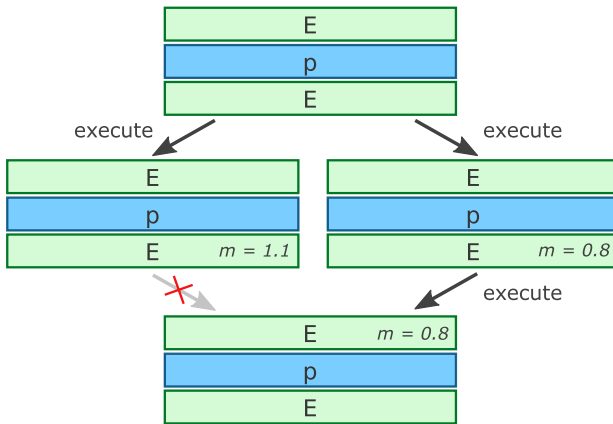


Figure 3.3.8. When joining different branches, model measures rate the quality of the preceding results to determine the new source environment.

3.3.5 Dynamic Compilation Systems

The interactive model-based compilation approach can handle branching, which is the reason why compilation systems can also be directed graphs and not only pure lists of processors. The approach includes two alternatives for decision making. In KIELER, as systems tend to be small, concise systems, this mechanic is rarely used in current releases.

Measure-based Continuation In the first alternative, all directly succeeding processors are executed. When joining different branches, one compares quality measures (m) inside the joining environments to decide which result will be the source for the joining processor, as can be seen in Figure 3.3.8. The measure is determined by a meter as described in Section 3.3.2 on page 74 and can be handled and customized like every other processor. Which characteristics of the models are used to determine m is up to the compilation system. By definition, a smaller value generally means a better result, e. g. model size. In the figure, two paths branch from a source. On both paths, a processor performs its job. The result is then judged by the meter. Compared to the source model, the result on the left branch is greater, i. e. worse. As

3. Model-Based Compilation Systems

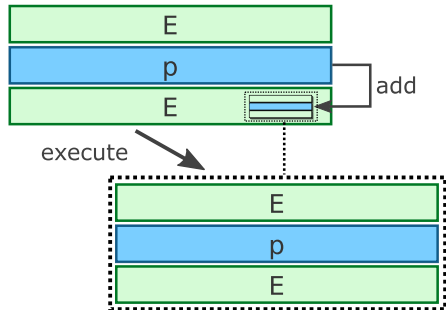


Figure 3.3.9. A processor can access the compilation context during run-time and change the subsequent compilation chain.

```

1 module SuspendAbort:
2   input A,B;
3   suspend
4   abort
5   pause;
6   when 2 B;
7   when A;
8 end module

```

(a) Abort statement nested inside a suspend statement in SCEst

```

1 module AbortSuspend:
2   input A,B;
3   abort
4   suspend
5   pause;
6   when 2 B;
7   when A;
8 end module

```

(b) Suspend statement nested inside an abort statement in SCEst

Listing 3.3.3. Nested statements in incremental SCEst compilations (from [Rah17])

a consequence, the right branch is chosen for the joining processor. Note that this mechanism can also be used to exclude invalid paths. An invalid model results in an infinite measure (∞) and is discarded. This, however, depends on the task of the processor. For example, if an optimizer fails, it should simply return the source model with $m = 1.0$ as a failed optimization should not change the artefact semantically.

Run-Time Continuation In the second approach, as compilation systems are also models and accessible from the contained processors, a processor can alter the system and, therefore, affect the succeeding processors. Therefore,

3.3. Interactive Compilation Systems

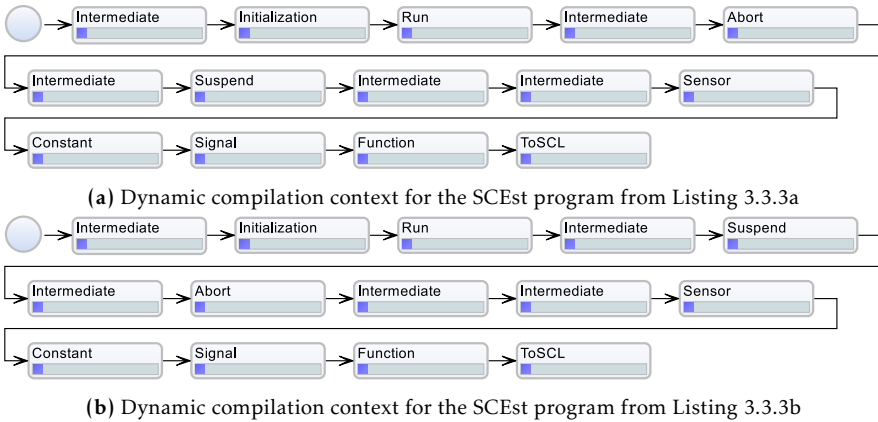


Figure 3.3.10. Illustration of a dynamically extended compilation context

it is possible to decide for the next processor during run-time. Figure 3.3.9 shows this conceptually. The processor in execution adds the next processor instance to the current context, which is accessible via the environment. After the processor has completed its operation, one then proceeds to the newly added processor instance. This is particularly helpful if a static schedule is not determinable, as has been shown by Rahimi-Barfeh [Rah17].

Rahimi-Barfeh showed that an incremental compilation with the translation rules presented by Rathlev [Rat15] according to a static SLIC schedule (cf. Section 2.3.1 on page 27) is not possible due to the nested structure of SCEst programs. While it is generally possible to compile hierarchical structures in SLIC, the approach fails here, because a static schedule cannot be found. The order in which different features are resolved in SLIC is fixed and therefore the transformation rules are restricted. They must always obey this static principle. However, run-time continuation relaxes this SLIC restriction. It can be used to compile these programs, e. g. from inner to outer statements, determining during compilation which features have to be transformed next.

Listing 3.3.3 shows two SCEst programs with nested elements. Listing 3.3.3a depicts an abort statement within a suspend block, whereas

3. Model-Based Compilation Systems

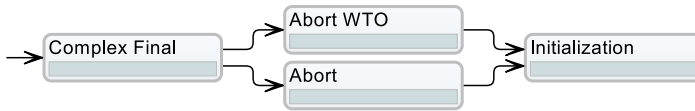


Figure 3.3.11. Immediate join of branched processors in KIELER

Listing 3.3.3b swaps the two instructions. Figure 3.3.10 shows the two compilation contexts after compilation of the SCEst programs shown in Listing 3.3.3. The Intermediate processor decides on-the-fly which processor follows. This *structural compilation system* is capable to compile the aforementioned nested programs. The dynamic context shown in Figure 3.3.10a resolves the inner abort before the suspend. In Figure 3.3.10b the suspend is transformed first. Both contexts use the same compilation system. More on the SCEst compilation is shown in Section 6.4.

3.3.6 Compilation System Branches

With the fork and join example in Section 3.3.5 it becomes clear that a compilation system can be seen as graphs, which may fork at specific nodes and must be joined again later according to the selected strategy.

Definition A *path* in a compilation system describes a distinctive directed route from the OSM to the desired target. Every path through a compilation system is a processor sequence, which can be executed to achieve a compilation from the OSM to the target. The *length* of a path is equal to the number of processors within the sequence. All possible paths of a compilation system represent the whole system.

A join of different paths can happen *immediately* after a path length of 1. This depicts an alternative of a single compilation step.

Example Figure 3.3.11 shows the immediate fork-join of two processor paths, which was originally presented elsewhere [MSH14]. SCCharts' abort feature was handled by two processors. The first followed the *write-things-once* principle and does not multiply used expressions, whereas the second used a more straight-forward approach and simply copied the expressions.

3.3. Interactive Compilation Systems

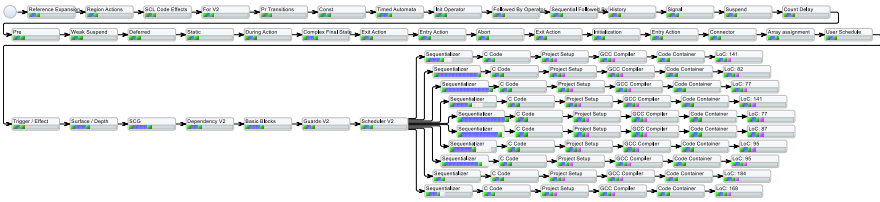


Figure 3.3.12. Complete system with LoC measurement of branches with different internal optimizations determine the quality of the optimizations by measuring the LoC of the generated assembler code.

Following the measure-based continuation from the previous section, both can be compiled and the better result w.r.t. a particular measure is then chosen, e. g. model element count or concurrent communication. Paths greater than one can be joined at any time. This is helpful if different compilation strategies lead to the same desired result. The best result, e. g. measured in LoCs, can then be used for deployment.

Finally, paths do not need to join. One can compose a compilation system to measure different strategies directly.

For example, to measure the quality of the optimizations of the netlist-based compilation approach, which is explained in Section 5.2, a system can be created that sends every optimization result to the GCC and measures the LoC of the generated assembler code. Therefore, the system branches after the scheduling step of the compilation chain and runs the sequentializer, which is responsible for generating the sequentialized code, with different optimizing steps. Eventually, the results are measured by appropriate analysers. Figure 3.3.12 shows final steps of such a system.

Example

This can be driven further by now measuring different GCC settings. The system depicted in Figure 3.3.13 shows two compilation rows with different optimization processors. The upper one resembles the system from Figure 3.3.12 running the GCC without further optimizations `-O0`, whereas the one below invokes the GCC with `-O2`, letting it optimize even more. Note that there is no need to add additional code to any processor. It is only a matter of system configuration. Besides, these results also show that

Example

3. Model-Based Compilation Systems

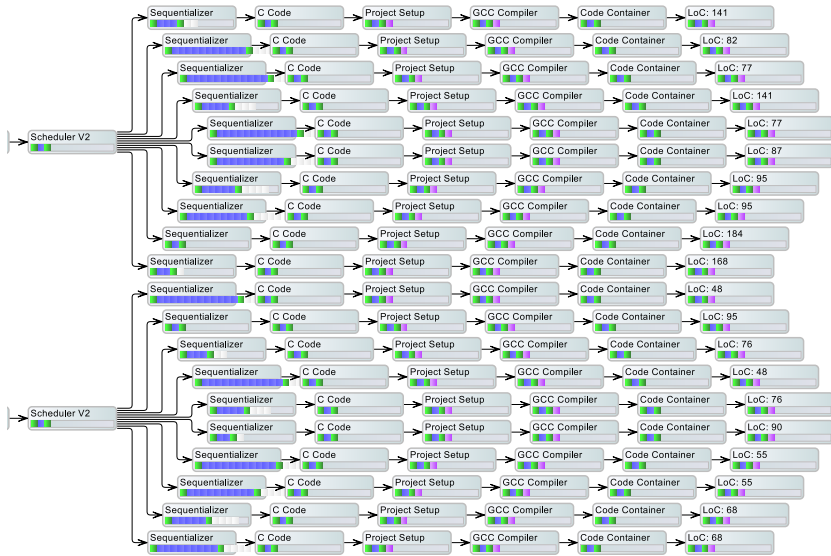


Figure 3.3.13. LoC measurement of branches with different external optimizations in a compilation system

optimizing the code before sending it to the GCC actually improves the results and is therefore effective. More details on the actual evaluation of the different compilation approaches are given in Section 5.5.

3.3.7 Visual Feedback

Figure 3.3.12 and Figure 3.3.13 also depict that the processors can add meta information directly to the view of the compilation system. Besides adding the result as textual log to the intermediate results of its environment, the LoC meter adds the final LoC count to the caption of its depiction. However, the transient view framework can also underline efficiency of the transformations by other means, e. g. , scaling or annotations. Figure 3.3.14 shows the ending of the same compilation system as before but gives immediate visual feedback about the quality of the result by scaling the corresponding

3.4. Derived Compilation Systems

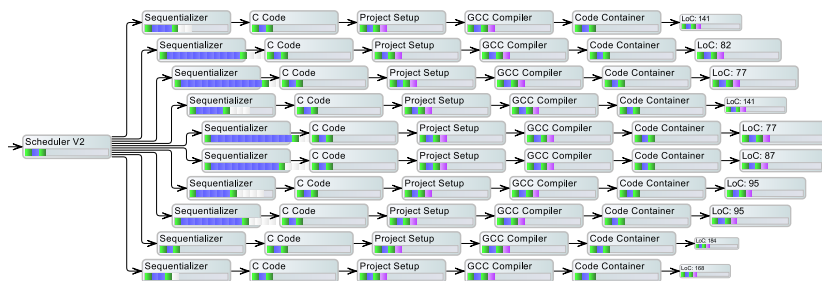


Figure 3.3.14. Instantaneous visual feedback of the quality of different compilation branches, via differently sized analysers at the end of each chain

processor nodes. Less prominent results get scaled down. Chapter 4 gives further insights on how these interactive compilation models can be used to facilitate MDE.

3.4 Derived Compilation Systems

As demonstrated by Motika [Mot17] transformation schedules of compilation processes do not have to be fixed or unique but can be calculated by dependencies between their transformation. In his works towards SLIC, two types of dependencies, namely *produces* and *not-handled-by*, are used to create such *SLIC schedules* automatically. One possible schedule was already depicted in Figure 2.3.1c on page 28 for the high-level SCCharts compilation. The dependencies which created this schedule are shown in Figure 3.4.1.

Since compilation systems are models, M2MTs can be used to create them. For example, in KiCo, one can use KiCo to create new compilation systems for KiCo. If the source language is expressive enough, it is unnecessary to create a new meta-model for the source and target languages to prototype.

While it is of course possible to create a dedicated meta-model for a *Example* SLIC schedule if desired, the following system uses the existing Sequentially Constructive Language (SCL) meta-model, which comes with an integrated editor and pre-existing processors in KIELER for demonstration reasons. The SCL is discussed in detail in Section 5.1, but the language includes the

3. Model-Based Compilation Systems

produces ↗	Weak Suspend	Deferred	History	Static	Valued Signal	Pure Signal	Count Delay	Pre	Suspend	Complex Final	Abort	During	Exit	Initialization	Entry	Connector
not handled by ↖																
Weak Suspend		↗		↗						↗		↗		↗	↗	
Deferred												↗		↗		
History				↗										↗	↗	
Static														↗		
Valued Signal						↗								↗		
Pure Signal												↗				
Count Delay												↗			↗	
Pre														↗		
Suspend												↗		↗		
Complex Final											↗			↗		
Abort							↖							↗	↗	↗
During												↖		↗	↗	↗
Exit												↖				↗
Initialization															↗	
Entry																↗
Connector																↗

Figure 3.4.1. SLIC dependency table (originally adapted from [MSH14])

```

1 @produces Deferred, Static
2 WeakSuspend();
3
4 @produces DuringAction
5 Deferred();
6
7 @produces Static
8 @notHandledBy Suspend, DuringAction
9 History();

```

Listing 3.4.1. SLIC dependency excerpt in SCL

KIELER expression language, which allows for simple calls and comprises annotations.

Example The high-level SCCharts transformations can be modelled as invocations in SCL. Their dependencies are then added as annotations. An excerpt is shown in Listing 3.4.1. All dependencies, depicted in Figure 3.4.1, can be

3.4. Derived Compilation Systems

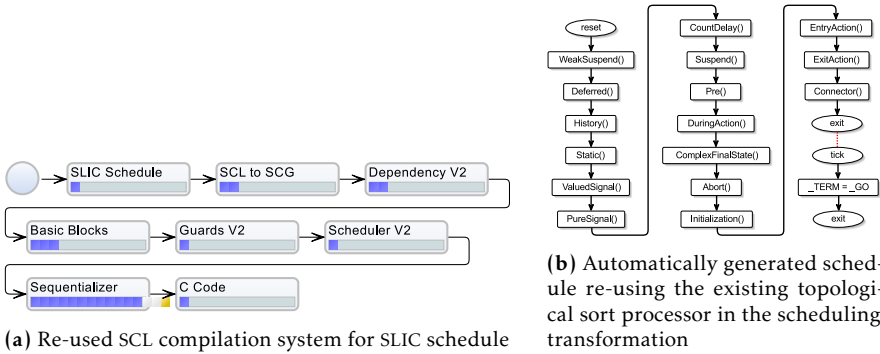


Figure 3.4.2. Generated SLIC schedule re-using existing processor modules

```

1 system de.cau.cs.kieler.slic.^schedule
2 label "SLIC Schedule"
3
4 de.cau.cs.kieler.scl.slic.^schedule
5 system de.cau.cs.kieler.scl.netlist.c

```

Listing 3.4.2. SLIC Schedule system

modelled live in the KIELER environment. To retrieve a feasible schedule, one can use the already existing scheduler processor on the control-flow graph of the program. It performs a topological sort taking all dependencies into account. Therefore, the only thing to do is to write a small processor which brings the already modelled dependencies in SCL annotations into a concurrent form, so that the sequential control-flow can be ignored and let the topological sort scheduler do the rest. This compilation system is listed in Listing 3.4.2. The new processor's transformation code only needs 23 LoCs. _

Interactive Prototyping

Simplicity is prerequisite for reliability.
— Edsger W. Dijkstra

Section 4.1 shows how different views can guide the modeller to refine a project through a feedback loop. Exemplary, six different kinds of information views, which were implemented in the KIELER project to increase the modelling efficiency, are presented. Section 4.2 shows the compilation system universe in KIELER, which comprises systems and processors for all implemented synchronous languages, simulation, and deployment to various target platforms. The views presented in this section target the tool builders audience and are not restricted to KIELER. Section 3.3.6 has shown that the interactive model-based approach can be used directly to gain interactive visual feedback about compilation systems or single processors. Section 4.3 builds upon these principles and gives further measures which rate the complexity of transformations. Experimental results are given for Extended SCCharts.

Please notice that all model diagram figures depicted in Chapters 3–7 are accessible interactively while modelling unless stated otherwise. They can of course be stored permanently for documentation reasons as has been done for this thesis.

4. Interactive Prototyping

4.1 Interactive Guidance

If a program is rejected by a compiler, it is important to guide the user towards the problem. Graphical languages have the advantage of intuitive visual problem reporting. However, regarding synchronous languages, such as SyncCharts and SCADE, this potential is often only used for simulation. For SCCharts as a descendant of SyncCharts, KIELER relies on a visual representation close to the original statecharts introduced by Harel. This graphical representation has been proven to be intuitively understandable and is also the basis of the UML statecharts dialect, as has been covered in Section 2.3 on page 26 ff.

```
1 x = 3;
2 if (x > 0) {
3     x = x * 7;
4 }
5 y = x * 2;

1 x_3 = 3;
2 if (x_3 > 0)
3     goto <bb 3>; [0.00%]
4 else
5     goto <bb 4>; [0.00%]
6

7 <bb 3> [0.00%]:
8 x_4 = x_3 * 7;
9
10 <bb 4> [0.00%]:
11 # x_1 = PHI <x_3(2), x_4(3)>
12 y_5 = x_1 * 2;
```

Listing 4.1.1. A small C program

Listing 4.1.2. Code snippet of the SSA intermediate representation in GCC generated with the `-fdump-tree-ssa` option. (Example taken from TR-1806 [SSH18d].)

For general purpose compilation, compilers also often allow access to intermediate representations. One motivating example from the introduction is the BB representation of the GCC. The GCC includes intermediate representation for basic blocks and optimizations in textual form. However, the accessibility and understandability is tailored to the needs of a compiler expert. Recall the extract of the SSA representation in Listing 4.1.2 of the C program in Listing 4.1.1. It illustrates the basic block separation (lines 7 and 10), renaming of variables (lines 1, 2, 8, 11 and 12), and placement of a Φ -function (line 11) in the partially translated code. While all information is present, without additional affiliated tools which further process these intermediate results, the representation is arguably rarely helpful for the modeller.

In recent years, a number of domain-specific modelling approaches have been developed, such as CINCO [NLK+18] and MARAMA [GHL+13] (also see Section 2.3 and Appendix B). While these tools provide sophisticated means

to work on the artefacts in question, once a modelling step is done and the model is compiled, there is little information or interactivity that guide the developer on what happened. The same is true for other statecharts modelling tools, such as Rhapsody or Simulink Stateflow.

As has been covered in Chapter 3, the interactive model-based approach provides the modeller with generic, interactive tools to orchestrate compilation processes. These are divided into atomic steps, which aid the modeller to refine the process and to find errors without the need for long development cycles. The source, intermediate, target and additional models are presented in well-readable graphical views using transient view and automatic layout technologies. To guide the modeller, one goal is to provide meaningful model representations in the domain of the modeller. Therefore, contrary to popular compiler infrastructures, such as LLVM, which try to reach a common intermediate language as soon as possible to maximize modularity. I propose to stay in a domain meta-model as long as is reasonable to facilitate understandability. The concept of strong interactive developer guidance should help modellers to develop more efficiently. Therefore, I propose that model-based tool developers should adhere to two tooling principles:

Guidance The modeller should not be burdened with maintaining an overview over all potential conflicts but should be assisted with finding solutions to these. Modelling tools should provide transient views automatically while the modeller works on their model.

Observation The modeller should be able to understand what is happening during compilations and transformations. Intermediate results, automatic mappings and meaningful annotations facilitate understandability and ease manual verifications.

Both principles, regardless of the concrete views in use, help to refine the source model, which in turn creates more refined representation in the dedicated views. Simultaneously, the generated views and intermediate modelling steps can be used to create project documentation. I propose to call this positive feedback loop Interactive Model-Understanding-Refinement-Documentation feedback loop (iMURD). It is depicted graphically in Figure 4.1.1. The modeller works on the OSM, which is compiled by the model-based compiler. During the compilation, intermediate models are generated

4. Interactive Prototyping

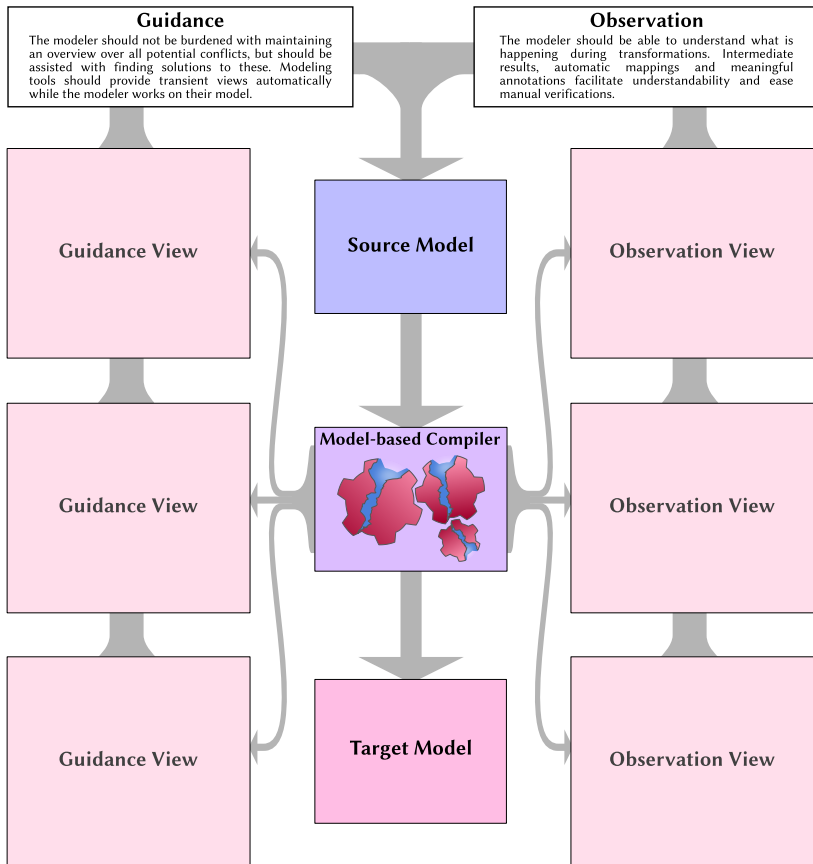


Figure 4.1.1. Schema of the Interactive Model-Understanding-Refinement-Documentation feedback loop (iMURD)

automatically and presented as model annotations or in dedicated views. The modeller uses this feed back to refine the OSM, which in turn will give more refined views. The cycle continues. While facilitating understandability, the generated views can be stored simultaneously separately and serve as ongoing documentation.

4.1. Interactive Guidance

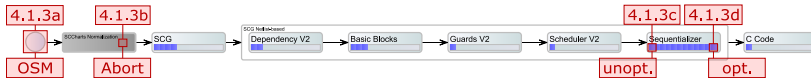


Figure 4.1.2. Interactive models in a KiCo compilation system

While SCCharts serve as leading example throughout this chapter, the main principles can be applied to all modelling tooling environments. Besides SCCharts, KIELER supports other synchronous languages, such as Esterel [SMR+17][Rah17] and Lustre [Gri19], as sources, which within KIELER already profit from the iMURD methodology. Since KiCo is a modular compiler, further languages can be added to the compiler easily. Intermediate models can be used to generate transformation *snapshots* easily and present them with or without additional information to the modeller. A *simple snapshot* represents the state of the transformation chain at that particular moment in the meta-model of the active processor, whereas an *augmented snapshot* is enriched with additional information, such as model element annotations or mappings to the original model, which basically come for free in the KiCo framework. Furthermore, an augmented snapshot can use any meta-model.

Figure 4.1.2 shows a KiCo compilation context within the KIELER SCCharts Editor. Each of the coloured rectangles in the context is an inspectable intermediate model. There are different ways of presenting the data which is gathered. The transient view technology will instantaneously generate a diagram if an appropriate synthesis exists, which is usually the case in KIELER though not a necessity. In the figure, annotations show the intermediate models, which are used in the next example in Figure 4.1.3 in Section 4.1.1, which gives an example for simple snapshots. Afterwards, Section 4.1.2 will discuss different guidance possibilities with augmented snapshots, partially tailed to synchronous languages. The principles are applicable beyond them. Both variants can be shown interactively by selecting the desired interactive result or as dedicated view.

4.1.1 Simple Snapshots

While transformation steps are mandatory for the desired compilation system to fulfil its task, they can also guide the user without further annotations

4. Interactive Prototyping

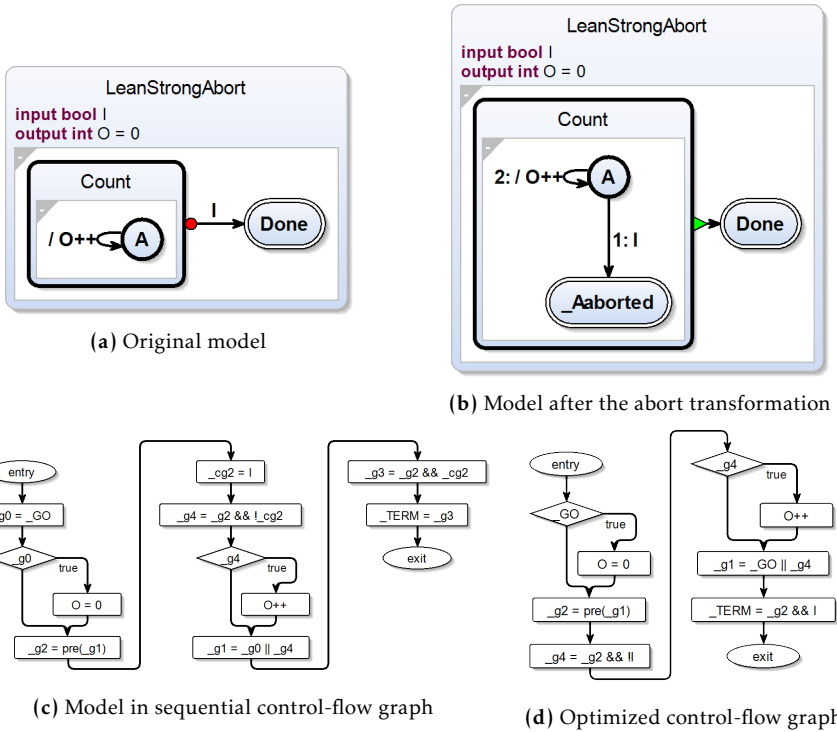


Figure 4.1.3. Example of different unmodified transformation snapshots: LeanStrongAbort

to understand the transformations stepwise. These steps can be shown as simple snapshots on arbitrary granularity.

Example Figure 4.1.3 shows four different steps of the KIELER netlist-based compilation, which in total consists of more than 30 transformations. The original SCCharts model can be seen in in Figure 4.1.3a. It serves as the source for the compilation. After initialization in the first cycle, the program increases an integer output O in every clock cycle. As soon as an input I is present, the counting stops immediately without increasing the counter in this cycle due to a so called *strong abort* (red circle transition) and the program is ter-

minated. Since strong aborts are an extended language feature, the feature has to be transformed into more simpler language constructs. Figure 4.1.3b shows the semantically equivalent model after the application of the abort transformation. The strong abort transition is now resolved into a normal termination transition (green triangle transition), which triggers if the control of its source state reaches a final state (double border). Transition priorities (with lower integers having higher priorities) now manage the counter. If l becomes true, the control switches to the `_Aborted` state. Otherwise, O is incremented. Eventually, the program is sequentialized into a controlflow graph which only contains *assignments* (rectangles) and *conditionals* (diamonds) in the netlist-based compilation as can be seen in Figure 4.1.3c. The whole netlist logic is executed in every cycle, with the `_GO` variable signaling the (re-)start of the program. `_TERM` is set to true if the program terminates. Values from previous cycles can be obtained with a *pre* operator. The sequentialized code can further be optimized. Figure 4.1.3d shows the results of the copy propagation [ASU86]. Even if not familiar with the netlist-based compilation approach, the behaviour of the program is observable. O is set to 0 in the beginning. It is increased in subsequent cycles managed by the *guard* `_g2` and if the input l is false. As soon as l becomes true, the program terminates without counting O . —

4.1.2 Augmented Snapshots

In the following, six different views which may guide the developer are demonstrated. Other possibilities of data processing and visualization are imaginable, especially when tailored to specific use-cases. There is no hard limit for the compiler framework. However, the usefulness of the processed data is tightly connected to its presentation. The guidance may be more effective if supported by transient view syntheses. Nonetheless, even plain text or unmodified artefacts may help the modeller as long as they are interactively accessible and easy to understand.

All figures show different variations of the data gathered from the compilation of the same model with slight modifications w.l.o.g. to show the potential of the different views. The first three views, described in Section 4.1.2.1 – 4.1.2.3, follow a more pro-active approach in notifying the

4. Interactive Prototyping

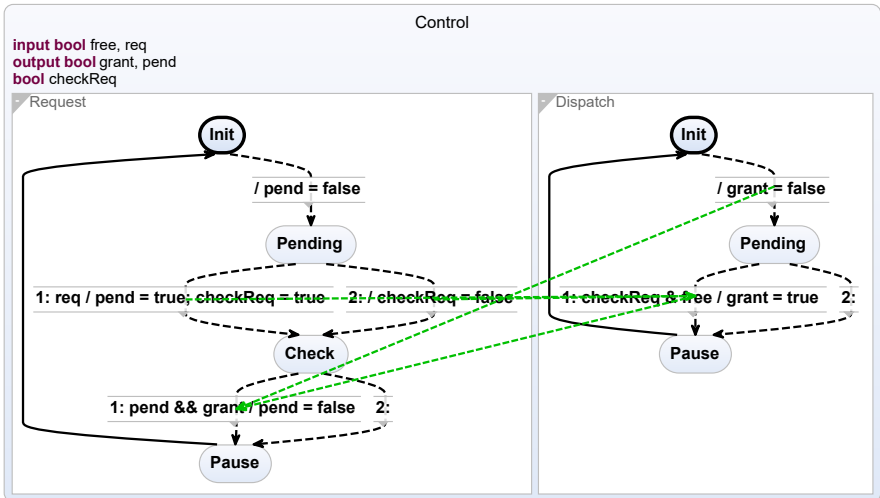


Figure 4.1.4. Data dependencies

modeller about potential conflicts, also called *causality guidance*. The last three views, explained in Section 4.1.2.4–4.1.2.6, help the modeller to understand what is happening during the transformations, which we call *transformation observation*. Unmodified snapshots also fall into this category.

4.1.2.1 Data Dependencies

Data dependencies usually govern the scheduling order of the program. Figure 4.1.4 shows an example SCChart with two concurrent regions. Dependencies between concurrent variable accesses can be visualized directly in the diagram of the model, instead of showing a corresponding control or dataflow flow graph. The data dependencies here are depicted as green dashed edges. They can also be visualized on different granularity levels within the model or only if specific states are selected by the user if desired. It is also possible to display all, i. e. non-concurrent, variable dependencies and mark ineffective accesses, which can help to refine the model further.

4.1. Interactive Guidance

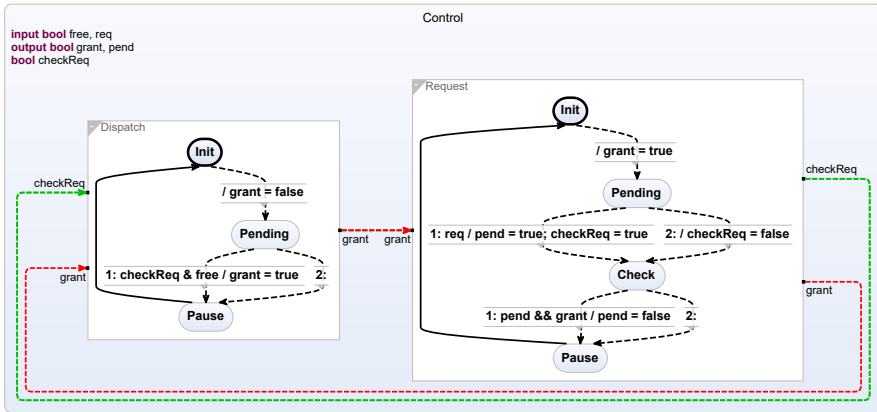


Figure 4.1.5. Causal dataflow

4.1.2.2 Causal & Induced Dataflow

The induced dataflow view [WSS+18] shows communication between concurrent regions. It visualizes the dataflow of the program even if the underlying model uses the control flow paradigm, such as SCCharts. A variant thereof, the causality dataflow view, focusses on identifying data dependency cycles. If the source model is changed, such that conflicting values are written to the variable `grant`, the dependency cycle is depicted in red, as can be seen in Figure 4.1.5. The modeller is now informed that a dependency cycle between concurrent regions is present and that the model is not constructive in the sense of the underlying MoC. It becomes clear that the compiler is going to reject the program without the need to actually run the compilation chain.

4.1.2.3 Scheduling Propagation

Nonetheless, if desired or if constructiveness is still not determined, programs can be compiled. Information gathered during compilation can be propagated back to the original model to provide reasonable feedback. On this level, complex scheduling relationships are simply displayed as annota-

4. Interactive Prototyping

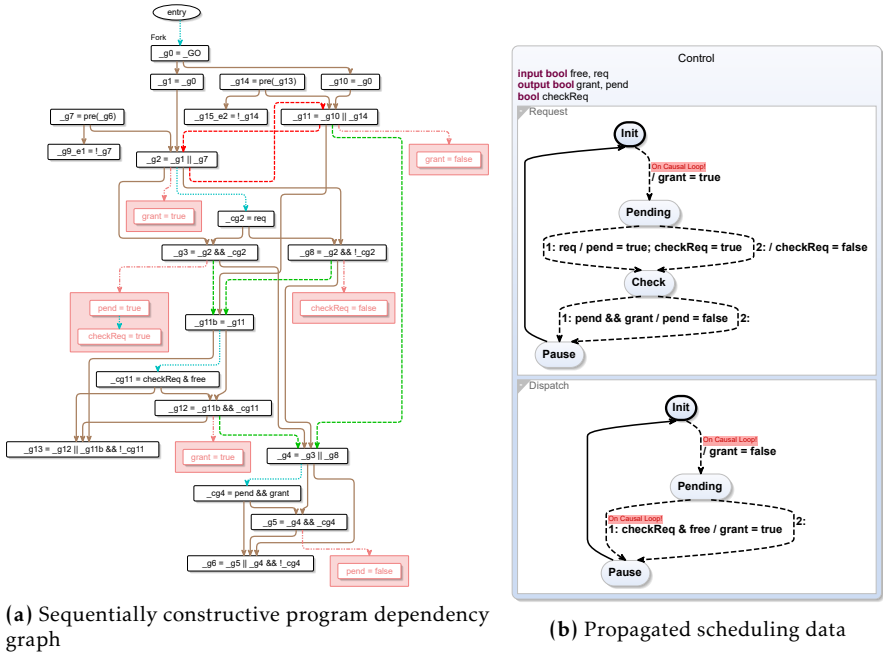


Figure 4.1.6. Scheduling propagation

tions [SSH18c]. The problematic cyclic variable access, which was introduced in Section 4.1.2.2, is shown to the modeller within the model.

The compiler (or expert thereof) can detect the dependency cycle in the program dependency graph of the program, which is depicted in Figure 4.1.6a. The complete scheduling information, which is needed during compilation anyway, is accessible interactively in a readable way. It can be used by experts to solve complex scheduling issues. The dependency cycle is depicted as cyclic dashed edge in the example. However, the modeller is informed via propagation in the original model on the right side in Figure 4.1.6b. Once notified, the issue can be fixed easily without the need to dive deep into the compilation chain.

4.1. Interactive Guidance

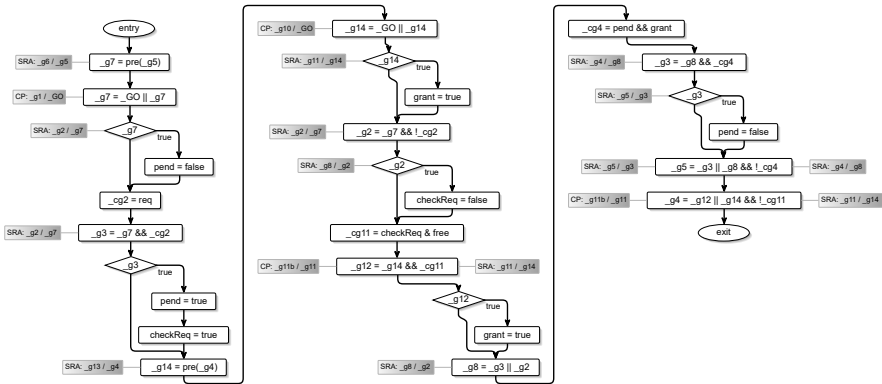


Figure 4.1.7. Transformation snapshots

4.1.2.4 Transformation Snapshots

Every intermediate step of the transformation chain can be preserved as intermediate snapshot model by the transformations and enriched with individual annotations [MSH14; SSH18c]. In the example shown in Figure 4.1.7, different classical optimizations, such as copy propagation (CP) [ASU86] and smart register allocation (SRA) [CAC+81], annotate which nodes were modified by their processes. The framework handles the mapping between model elements automatically. Therefore, it is not necessary for the compiler developer to keep track of these changes manually. They can simply add annotations to specific model elements, e. g. , graph nodes in the example, programmatically during optimization. The annotations will be attached automatically to the appropriate visualization of the selected nodes.

4.1.2.5 Automatic Element Tracing

The compiler framework keeps track of the transitive model element relations [RSM+16]. Therefore, the relationship between the elements of arbitrary intermediate models of the compilation chain can be made visible. In the example shown in Figure 4.1.8, the nodes of the final model of the netlist-based compilation [MSH14] are mapped back to the original model.

4. Interactive Prototyping

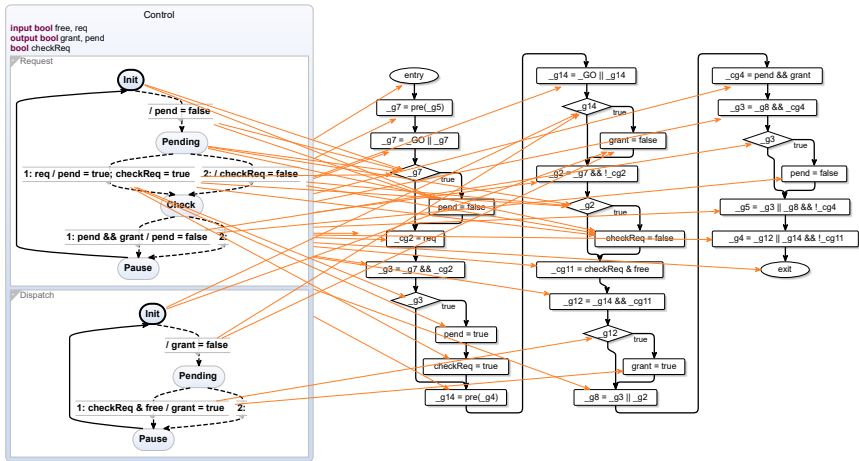


Figure 4.1.8. Automatic element tracing

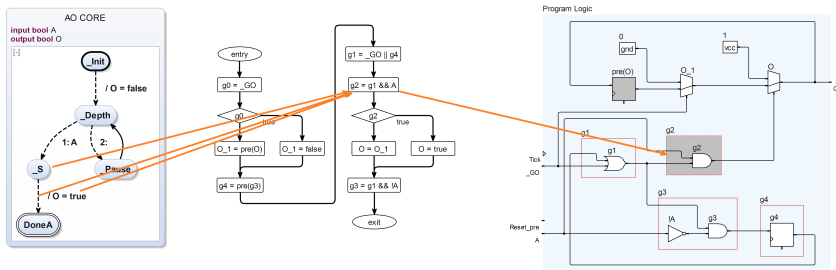


Figure 4.1.9. Automatic element tracing of selected elements

Displaying all relationships simultaneously can be confusing. Therefore, the user should be able to select points of interest by clicking on the elements. Figure 4.1.9 shows a smaller program with a transitive element tracing over three transformation steps. In this case, the final model represents a hardware circuit. The paths which lead to the creation of single gates can be easily inspected.

4.2. Compilation Systems Universes

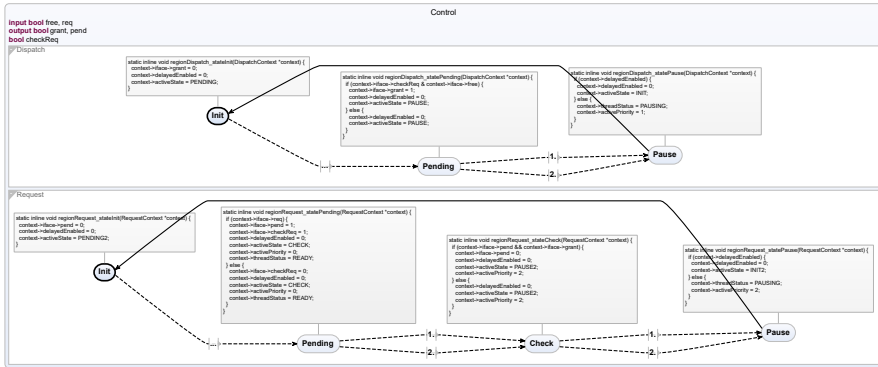


Figure 4.1.10. Built-in code mapping

4.1.2.6 Integrated Code Mapping

I propose that to annotate generated target code directly in the original model. This must be supported by the chosen code generator, i. e. the modelling environment must be able to map reasonable code blocks to corresponding model element.

For example (see Figure 4.1.10), SCCharts' state-based code generation creates one function for each state besides other functions. These final code fragments can be shown directly in the original model. As shown in the figure, the generated source codes of each state function are attached as comment nodes to the state in the original model.

For SCCharts, all aforementioned views and the iMURD cycle for SCCharts are depicted in the accompanying poster of the guidance presentation [SSH18a], which can be seen in Appendix C.

4.2 Compilation Systems Universes

Since the proposed model-based compilers use models to describe their compilations, these models can be used to generate new views of the whole compiler universe. Similar to the Unix universe, which describes all possible

4. Interactive Prototyping

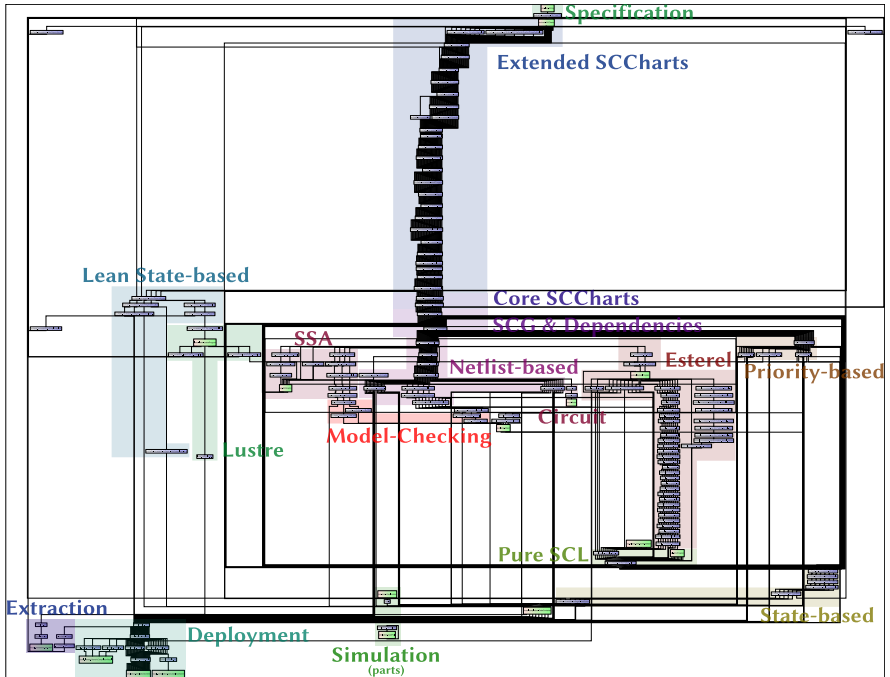


Figure 4.2.1. KiCos compilation system universe depicting the overall usage of different processors

routes for ETI in Section 2.3.3 on page 33, a KiCo universe can be displayed interactively to guide both, the modeller and the tool developer. Such a universe depicts all relations between all registered compilation systems and gives information about potential source and target meta models.

Example

Figure 4.2.1 shows the compilation universe of KiCo in the 1.0 release of KIELER. It depicts the overall usage of processors and their connections to other processors. Processors are shown as nodes with blue colour gradient. Start and end meta models of compilation systems are shown as nodes with red-green colour gradient. Each edge represents an sequential execution order of processors within one compilation system. The figure is annotated with the different areas of KiCo, of which some will be explained in detail in

4.2. Compilation Systems Universes

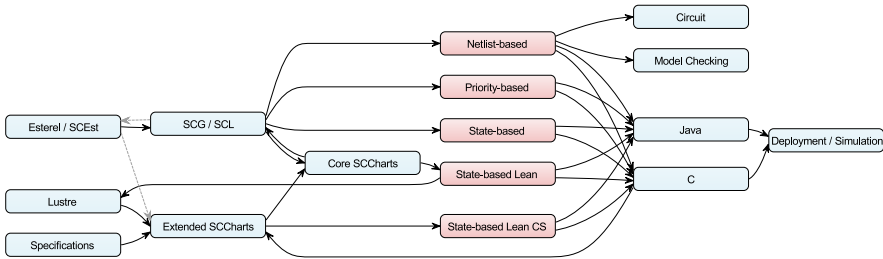


Figure 4.2.2. Top-level view on the KIELER systems universe (created manually)

Part II. It can be seen that the whole block of Extended SCCharts processors precede the Core SCCharts section. Afterwards, different compilation approaches, such as the netlist-based, priority-based and state-based, fork off. Additionally, compilation paths for Esterel and Lustre are present. Besides compilation, the universe also comprises systems for deployment, simulation and model-checking.

These overviews are generated on-the-fly and help tool developers to orientate. They can also be used to generate arbitrary systems between meta-models as long as a directed path from source to target exists. Dynamic systems could be created similar to the SLIC system generated in Section 3.4.

A top-level view of the current systems in KIELER and their compilation paths is shown in Figure 4.2.2. The meta-models, which define the source and target models, are illustrated in blue colour. Different compilation strategies, explained in Chapter 5, are shown in red colour. The paths depicted in grey were implemented in the KIELER in former version but are not part of the current release. Although Figure 4.2.2 was created manually as an overview here, such a view could be generated automatically with appropriate meta information.

While Figure 4.2.1 shows the total universe of the model-based compiler, specific information can be visualized as well. Figure 4.2.3 shows an excerpt from the KiCo registry, displaying which KiCo elements possess a relationship to the `scharts.ui` plugin. All classes in this plugin which reference compilation systems are depicted in green. The compilation systems are shown in red and the processors they use in blue. For example,

4. Interactive Prototyping

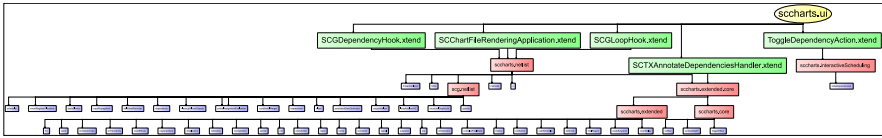


Figure 4.2.3. Example of the KiCo Registry

the three classes `SCGDependencyHook`, `SCChartsFileRenderingApplication`, and `SCGLoopHook` reference the `sccharts.netlist` system. Besides using the processors `threadAnalyzer`, `SCG`, `methods`, and `C` directly, the system itself references the systems `scg.netlist` and `sccharts.extended.core`.

4.3 Transformation Complexity

M2MTs often resolve compact languages features into more basic features, which often increase the overall model element size. The *internal complexity* of a transformation is hidden from the modeller by design to enable them to create more compact models. In some scenarios, a high internal complexity becomes an issue, because the modeller is not really aware of the model growth. Real-life examples of a system with scarce resources are Lego Mindstorms, which are sometimes used in teaching. They are discussed in Section 7.3.1.

Following the approach presented in Section 3.3.6, the interactive model-based compilation can give feedback about the transformation complexity to the modeller. I propose two measures for transformation complexity and presents the experimental data for KIELER SCCharts. These values could be displayed in a dedicated view or model annotations, similar to the Interactive Timing Analysis (ITA) hotspot highlighting done by Fuhrmann. In the ITA, worst-case execution time are annotated to their corresponding model regions.

Motika showed that every Extended SCCharts model feature can be transformed into a semantically equivalent Core SCCharts model [Mot17]. In order to measure the complexity of specific extended features, a *base model*, which can be seen in Figure 4.3.1, serves as OSM. The model only uses

4.3. Transformation Complexity

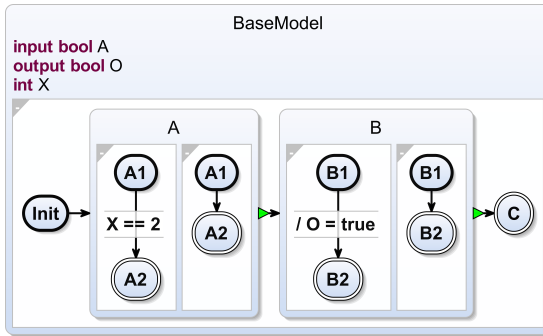


Figure 4.3.1. The SCCharts base model for complexity approximations

features included within the scope of Core SCCharts. It is constructed such that extended features can be added without introducing new structural model elements. Therefore, only the complexity resulting from the use of one designated extended feature is added to the base model. The complexity values of the base model and the model including the extended feature allow the calculation of the complexity ratio C_{ratio} as follows

$$C_{\text{ratio}} = \frac{f(P_{\text{feature}})}{f(P_{\text{base}})}.$$

Depending on the particular meta-model in use, different model characteristics can be measured. Exemplary for meta-models similar to SCCharts, two different domains are measured here: the amount of variables f_{vars} and the complexity of expressions f_{exp} . Both values are measured in the final sequentialized control-flow graph resulting from the netlist-based compilation approach. The complexity value f_{vars} counts the amount of used variables including the variables for the guards of the netlist. The value f_{exp} evaluates the expressions used by counting the amount of operands and operators. More precisely, the following formula is used for the calculation:

4. Interactive Prototyping

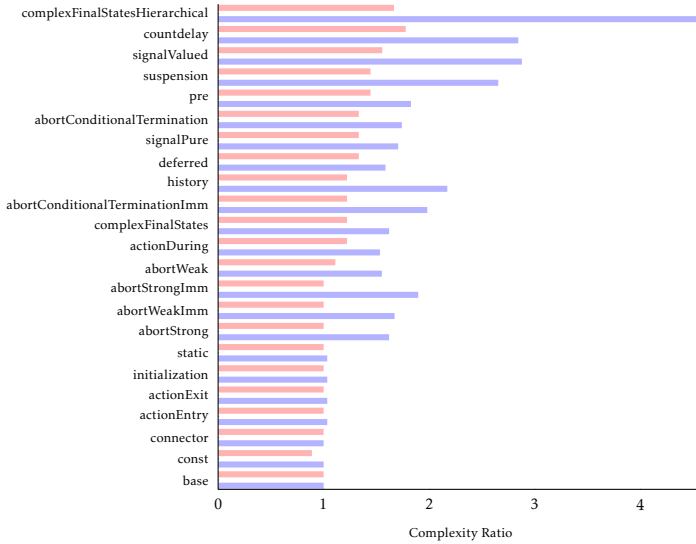


Figure 4.3.2. Complexity ratio for expressions (blue) and valued objects (red)

$$f_{\text{exp}}(P) = \sum_{e \in E} \begin{cases} 1 & \text{type}(e) = \text{Literal} \\ 1 & \text{type}(e) = \text{ObjectReference} \\ 1 + \sum_{e' \in \text{exprs}(e)} f_{\text{exp}}(e') & \text{type}(e) = \text{OperatorExpression} \end{cases}$$

where E is defined as the set of all root expressions included in the program P . The function $\text{exprs}(x)$ is defined as the shallow set of expressions of the expression x .

Example For an expression e with $e = (a + b) - c$ the function is defined to return $\text{exprs}(e) = \{a + b, c\}$. The function exprs does not traverse the expression tree recursively.

For each extended feature, cf. Figure 3.4.1 on page 92, a new model was created and measured with both aforementioned measures. The results are summarized in Figure 4.3.2. The ratio for expression complexity lies within $[1, 4.55]$, and the ratio for the amount of valued objects is within $[0.89, 1.78]$.

4.3. Transformation Complexity

The ratio below 1 is achieved using the constants declaration, which allows to optimize code generation and thus saves one variable. However, both approaches lead to a similar conclusion. The transformations of hierarchical complex final states, count delays and valued signals seem to introduce the most complexity regardless of the measurement approach. In conclusion, some of the extended features might expand the original model in such ways that they are not viable w.r.t. systems with limited resources. Features with a high internal complexity ratio should not be used in such systems. Furthermore, transformations with high complexity ratio could be replaced by alternative transformations, e. g. transformations with specialized or down-graded features.

Note that the concrete values presented here significantly depend on the nature of the base model and also on the underlying meta-model. More or less hierarchy may lead to different peaks in the ratio. Especially combinations of different extended features may result in an unexpected growth of the model. The complexity results illustrated here give an idea of the implications of different transformations, but they need to be generalized in order to make a more elaborated statement on the effects of different transformations. However, with growing and more complex models the measurements are expected to further increase.

Part II

Beyond Compiling SCCharts

Interactive Compilation for SCCharts

Simple, few parts, easy to maintain, very strong.
— General Chuck Yeager (praising an airplane's engine)

In this chapter, the previously mentioned compilation methodology from Part I is used to realise a complete compiler with different compilation approaches for SCCharts. While not all steps, such as dependency or basic block analyses, are novel w.r.t. functionality, the interactive nature is. It is difficult to illustrate all the gained modelling pragmatics from the previously introduced approach in print. The following chapters attempt to illustrate the advantages of the approach bearing in mind that all the following engineering steps can always be inspected interactively. All model examples presented in this chapter can be inspected live and instantaneously within a model-based framework, such as KIELER in the case of SCCharts. While SCCharts serve as primary example, the steps taken to compile an SCCharts program can be applied to any language and to statecharts dialects in particular.

Section 5.1 introduces the Sequentially Constructive Language (SCL), an imperative minimal language for sequential constructiveness, which is used within the compiler to do most analyses. Section 5.2 explains the so far most used compilation approach of SCCharts, the netlist-based compilation in detail. The initial SCCharts proposal [HDM+14] also featured a second compilation approach: The priority-based compilation approach, which is explained in Section 5.3. As more light-weight compilation alternative to these

5. Interactive Compilation for SCCharts

established approaches, I propose the state-based compilation in Section 5.4. The goals targeted with this approach are understandability and simplicity. In two variants of the state-based compilation, complex inter-thread communications are forbidden to gain more readable automatically generated code. These approaches combined constitute the SCCharts reference compiler, which is implemented in the KIELER SCCharts tools. Section 5.5 compares the approaches against each other.

The chapter presents the “the whole story” w.r.t. SCCharts compilation. It contains everything that is necessary to realize the three compilation approaches with the interactive model-based compilation approach, which is also a major contribution, as stated in Section 1.1. However, to clearly separate novelties in single compilation steps from already established methodology, smaller novel contributions besides the interactive model-based nature are:

1. Three variations of the state-based compilation approach,
2. differentiation between Basic Blocks (BBs) and Scheduling Blocks (SBs),
3. three different types of synchronizers and how they integrate into the model-based compilation approach,
4. curing schizophrenia with Structural Depth Join (SDJ),
5. the Sequentially Constructive Program Dependency Graph (SCPDG) as representation for netlists,
6. a relaxation of the SCM_oC, called SC+, to schedule programs with spurious control-flows,
7. ten optimizers, which facilitate the interactive model-based approach.

5.1 The Sequentially Constructive Kernel Language

The Sequentially Constructive Language (SCL) [HMA+13] is an imperative, small language which is used to express sequential constructiveness in a minimalistic way. It only consists of seven constructs:

$$s ::= x = e \mid s_1 ; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid l : s \mid \text{goto } l \mid \\ \text{fork } s_1 \text{ [par } s_2 \text{ ... par } s_n \text{] join} \mid \text{pause}$$

5.1. The Sequentially Constructive Kernel Language

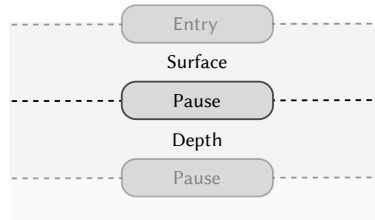


Figure 5.1.1. Surface and depth w.r.t. a pause statement

where x is a *variable*, e an *expression* and l is a *program label*. In detail the statement s comprises the following standard operations.

Assignment An assignment is a statement of the form $x = e$, where x denotes a variable which is written to. The expression e determines what is written to x . The assignment may not have any side effects w.r.t. to the rest of the SCL program. This is a write access to variable x with arbitrary variable read accesses within e . SCL does not specify the kind of expression language which should be used but to form meaningful expressions, literals (e. g. integers), variable read accesses and a set of standard operations (e. g. addition, subtraction) should be supported. To check for conditions a boolean data type and compare operators should also be included.

Conditional A conditional if e then s_1 else s_2 checks an expression e to be true or false. If it is true, the control-flow continues with s_1 and ignores s_2 . Otherwise, s_1 is ignored and the control flow continues in the else-branch with s_2 .

Sequence A sequence separates two statements. Semantically, s_2 is executed immediately after s_1 . Note that a sequence is an own operator and not a line delimiter. Hence, the last statement in a scope usually does not need a semicolon. It is permitted though since s_2 is considered an an empty statement, or *no operation*, in this case.

Label Every statement within an SCL program can be prefixed with a label. A label is a globally unique identifier.

Goto The goto statement directs the control-flow to the given label. The flow continues immediately.

5. Interactive Compilation for SCCharts

Parallel A parallel statement forks the control-flow into s_1 and s_2 . Both flows run concurrently until joined again with the join keyword. The separated flows between fork and join are called *threads*.

Pause A pause consumes time. As tick delimiter, the pause statement lives in two ticks. The control flow stops at this statement for this tick. It will continue from here in the next tick. To separate the different tick instances, one speaks of the *surface* of a pause when entering the pause statement. Leaving the statement in the next tick is called the *depth* of the pause. The surface, resp. depth, of a pause spans over all statements before, resp. after, the pause until the next pause delimiter or the program start or end, as depicted in Figure 5.1.1.

Similar to the parallel statement's threads, the whole program also forms a thread between the beginning and the end.

Definition A *well-formed* SCL program is one(1) in which expressions and variable assignments are type correct, (2) which has no duplicate or missing program labels and (3) has no goto jumps into or out of a parallel composition [HMA+13].

Motika et al. [MSH14] has shown that every SCCharts program can be transformed into a semantically equivalent SCCharts program which consists of only the five normalized SCCharts pattern which are now known as the SCCharts Kernel Pattern (SKP).

Definition The five normalized SCCharts patterns which represent the abstract concepts of *region*, *superstate*, *trigger*, *effect* and *state*, form the SCCharts Kernel Pattern (SKP).

KIELER SCCharts only accept programs which can be transformed into the SKP. An SCChart in SKP form is called *Normalized Core SCChart*. Figure 5.1.2 shows the direct mapping from normalized SCCharts to SCL and its graphical representation, the SCG. Every normalized SCCharts program can be written as well-formed SCL program. Since the complete mapping is done one-to-one, all transformations can be realized with the processor concept established in Part I using appropriate meta-models.

Region Regions are translated to SCL threads. The contents between their initial and final states are mapped to the SCG nodes between the *entry* and *exit nodes* of the corresponding threads.

5.1. The Sequentially Constructive Kernel Language

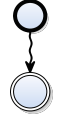
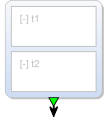



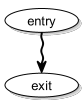
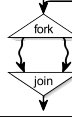
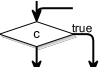

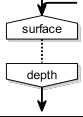
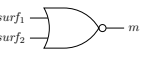
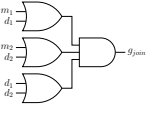
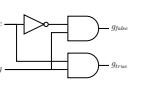
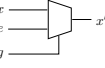
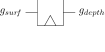
	Region	Superstate	Trigger	Effect	State
Normalized SCCharts					
	Thread	Concurrency	Conditional	Assignment	Delay
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
SCG					
Data-Flow Code	$d = \mathcal{E}_{\text{exit}}$ $m = \neg \bigvee_{\text{surf} \in t} \mathcal{E}_{\text{surf}}$	$\mathcal{E}_{\text{fork}} = \bigvee \mathcal{E}_{\text{in}}$ $\mathcal{E}_{\text{join}} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$	$g = \bigvee \mathcal{E}_{\text{in}}$ $\mathcal{E}_{\text{true}} = g \wedge c$ $\mathcal{E}_{\text{false}} = g \wedge \neg c$	$g = \bigvee \mathcal{E}_{\text{in}}$ $x' = g ? e : x$	$\mathcal{E}_{\text{depth}} = \text{pre}(\mathcal{E}_{\text{surf}})$
Circuits					

Figure 5.1.2. Matrix showing the entire mapping throughout the transformation process from SCCharts to circuits (adapted from [HDM+14])

Superstate A superstate depicts concurrency. The regions of a superstate form the different threads of a fork-join. The superstate pattern also includes the normal termination, which joins the different regions. Note that it is perfectly fine to have only one region in a superstate, e. g. for structural reasons. A direct mapping would create a fork-join construct with only one thread. While this is valid, the superfluous fork and join statements can be omitted in this case, which is achieved by a dedicated optimizer described in Section 5.2.8.

Trigger A trigger in normalized SCCharts is a simple state with two outgoing transitions. The transition with the higher priority is guarded by a conditional c , which is called the *trigger*. This transition is taken, if and only if c evaluated to true. Otherwise, the second *default* transition

5. Interactive Compilation for SCCharts

is taken. Both transitions are immediate and do not consume time. In the SCG, a *conditional node* is depicted as a diamond with the condition c as label. The true branch is attached on the right or left side, whereas the else branch is connected to the bottom.

Effect An effect consists of a simple state with one outgoing immediate transition. The effect is added to this transition without a trigger. Once the state is entered, it is left immediately and the effect is executed. Effects are rectangular assignment nodes in the SCG.

State To consume time, i. e. a tick, a simple state can have one outgoing delayed transition. Analogously to a pause in SCL, a delay state lives in two ticks. Its *surface* is the part of the pause that is entered in the current tick when the corresponding thread enters its pause. In the next tick, execution starts in the second part of the pause, namely its *depth*. Therefore, in the SCG, a pause is depicted by two nodes connected by a dotted *tick boundary edge*.

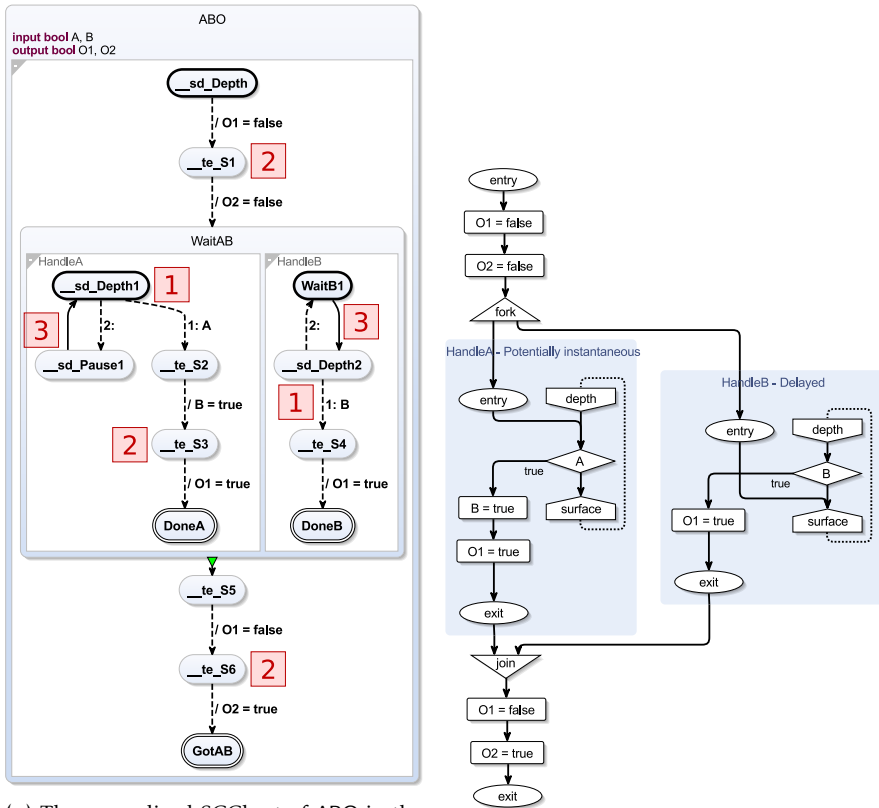
Motika et al. [Mot17; MSH14] have shown that over 30 extended features, including features which other language must include in their kernel language, such as preemption and suspend in Esterel, can be expressed with these five patterns.

Conclusion Since every SCCharts program can be transformed into a semantically equivalent normalized SCChart, every SCCharts program can eventually be written as well-formed SCL program.

Sequentially Constructive Graph

An Sequentially Constructive Graph (SCG) is a labelled graph $G = (N, E)$ whose *statement nodes* N correspond to the statements of the program, and whose edges E reflect the sequential execution ordering and data dependencies between the statements. Nodes and edges are further described by various attributes. A node n is labelled by the *statement type* as shown in the matrix in Figure 5.1.2. Nodes labelled with $x = e$ are referred to as *assignment nodes*, those with $if(e)$ as *condition nodes*, all other nodes are referred by their statement type (*entry nodes*, *exit nodes*, etc.). The matrix illustrates how SCG elements correspond to an SCL program. Every edge e has a type $e.type$ which specifies the nature of the particular ordering constraint expressed by e . The

5.1. The Sequentially Constructive Kernel Language



(a) The normalized SCChart of ABO in the SCCharts language

(b) ABO expressed as SCG

Figure 5.1.3. The ABO program in two different meta-models. Note that both views use the same OSM. They show two different points of time during the same compilation.

5. Interactive Compilation for SCCharts

black edges which connect nodes represent the sequential control-flow of the program. Edges that follow the IURP are labelled *iur*-edges. *iur*-edges combined with the sequential control-flow edges are termed *instantaneous* edges. A dotted edge is a *tick boundary edge*. Tick edges connect surfaces and depths from pause statements. They are *delayed*.

The normalized version of ABO, which was introduced in Figure 2.1.2a on page 17, is shown in Figure 5.1.3a. During the normalization steps from Core SCCharts to Normalized Core SCCharts, triggers and effects get separated [1]. If there are multiple effects present, each effect will get its own simple state pattern [2]. Eventually, if a transition is delayed, a pause pattern without additional triggers or effects is generated [3] by extracting and placing them in front of the pause. Once the normalized version is created, a one-to-one processor can transform the program from the SCCharts meta-model to the SCG meta-model, which is depicted in Figure 5.1.3b. Note that each thread has exactly one entry and one exit node, which mark their start and the end points. However, SCCharts models can have multiple final states within one region. They are combined into a single exit node to ease subsequent tasks on the SCG since multiple final states only serve the modeller's convenience and are semantically equivalent. This is a design decision, which for the most parts makes further downstream processing easier. However, it should be noted that the reverse transformation from SCGs back to the SCCharts meta-model, while semantically still correct, may not result in the same model as before.

When constructing the SCG, it is useful to analyse the delay behaviour of every thread. Within a fork-join construct, the thread types determine how to join these threads again. With a Breadth-First Search (BFS) or Depth-First Search (DFS) every path from the threads entry node to its exit node is checked.

Instantaneous If there is no surface (or depth) node in any all paths from the entry to the exit node, the thread is considered *instantaneous*. It will terminate in the same tick as it was entered.

Delayed If every path from the entry to exit node is interrupted by a surface–depth node combination at least once, the thread is considered *delayed*. It will never terminate in the same tick in which it was entered.

5.2. Netlist-Based Compilation



Figure 5.2.1. Core processors of the netlist-based compilation

Potentially Instantaneous If the exit node can be reached instantaneously from the entry node but is not required to do so within one tick, due to a conditional node, the thread is considered *potentially instantaneous*.

In KIELER, analysing the thread delay behaviour is done as a post-process- *Example* step within SCG transformation. The result is indicated at the beginning of the thread in the graphical SCG syntax as can be seen in Figure 5.1.3b. Here, HandleA is marked as potentially instantaneous, because the thread can terminate at once if A is true. However, it will not terminate if A is false. Thread HandleB is marked as delayed, because there is no direct way to reach the exit node from the entry node without passing a surface–depth combination.

5.2 Netlist-Based Compilation

The so far perhaps most used code generation approach for SCCharts is the netlist-based compilation approach. Netlists can serve as basis for software and hardware code generation. Figure 5.2.1 shows the workflow of the approach. Every single step is explained in the following sections. After the SCG is constructed from a normalized SCCharts, a dependency analysis, explained in Section 5.2.1, is run to gather information about variable accesses. Afterwards, the program is partitioned into *basic blocks* (Section 5.2.2). A netlist can then be constructed from the guards and the guarded assignments of the generated basic blocks (Section 5.2.4).

An *SC-schedule* is a subset of instantaneous edges of an SCG. A *structural SC-schedule* is an *SC-schedule* which is solely derived by analysis of the program structure. A program for which the structural SC-schedule is acyclic is *structurally acyclic SC*, abbreviated SASC. The data-flow approach presented here requires that the SCG is SASC; this, for example, forbids any loops which are *instantaneous*, i. e. where the loop body is not interrupted by a tick

5. Interactive Compilation for SCCharts

boundary, because it is impossible to construct a cyclic (instantaneous) netlist.

Implementation

The total configuration in KIELER of the compilation is listed in Listing 5.2.1. The system's id is set to `de.cau.cs.kieler.scg.netlist` in Line 1. The label, which is used within the IDE, is set to `SCG Netlist-based` in Line 2. Lines 3–10 configure the start environment of the compilation, which can be overridden by referencing systems. The following Lines 12–32 describe the list of processors with potentially individual configurations in the order of their execution. For example, Line 13 configures the dependency analysis to report instantaneous loops as warnings. If possible, the *structural depth join* processor in Line 16 will cure the loops. If it would not be available, the dependency analysis should throw an error instead, because the model is not compilable with this system.

As explained in Chapter 3, the configuration is an interactive model, i. e. a system, and can be inspected and changed as required. It is instantiated as new context to drive a concrete compilation. The workflow in Figure 5.2.1 is automatically synthesized from the system. Single processors can be configured differently via the configs in the environment. The start config configures the total system and is transferred to any system that references this one, so that single configurations can be overridden.

5.2.1 Dependency Analysis

The dependency analysis detects variable accesses, as introduced in Section 2.1.3 on page 19, and stores them in the SCG model. For this, the detection must traverse through the whole SCG, which can be accomplished by BFS or DFS.

SCCharts follow the IURP for concurrent threads and the sequential control flow otherwise. Two threads t_1, t_2 with $t_1 \neq t_2$ are concurrent if they share a *least common ancestor fork*. Ancestor threads are the transitive closure of parent threads ($\{parent(t), parent(parent(t)), \dots, root\}$). The inner-most nested common ancestor thread of t_1 and t_2 is called the least common ancestor thread. The spawning fork of t_1 and t_2 in that thread is called least common ancestor fork, or LCAF. However, a scheduling order on nodes in concurrent threads has only to be imposed if the two nodes in question

5.2. Netlist-Based Compilation

```
1 system de.cau.cs.kieler.scg.netlist
2   label "SCG Netlist-based"
3   start config {
4     "de.cau.cs.kieler.scg.opt.copyPropagation":true,
5     "de.cau.cs.kieler.scg.opt.conditionalMerger":true,
6     "de.cau.cs.kieler.scg.opt.haltStateRemover":true,
7     "de.cau.cs.kieler.scg.opt.smartRegisterAllocation":true,
8     "de.cau.cs.kieler.scg.opt.persistentStateOptimizer":true,
9     "de.cau.cs.kieler.scg.opt.partialAssignmentEvaluation":true
10  }
11
12 de.cau.cs.kieler.scg.processors.dependency config {
13   "de.cau.cs.kieler.scg.processors.loopAnalyzer.warningOnInstantaneousLoop": true
14 }
15 post process de.cau.cs.kieler.scg.processors.loopAnalyzerV2
16 pre process de.cau.cs.kieler.scg.processors.structuralDepthJoin
17 de.cau.cs.kieler.scg.processors.basicBlocks
18 post process de.cau.cs.kieler.scg.processors.expressions
19 de.cau.cs.kieler.scg.processors.guards config {
20   "de.cau.cs.kieler.scg.processors.loopAnalyzer.considerAllDependencies": true,
21   "de.cau.cs.kieler.scg.processors.loopAnalyzer.warningOnInstantaneousLoop": false
22 }
23 de.cau.cs.kieler.scg.processors.scheduler
24 de.cau.cs.kieler.scg.processors.sequentializer
25 post process de.cau.cs.kieler.scg.processors.copyPropagation
26 post process de.cau.cs.kieler.scg.processors.conditionalMerger
27 post process de.cau.cs.kieler.scg.processors.haltStateRemover
28 post process de.cau.cs.kieler.scg.processors.smartRegisterAllocation
29 post process de.cau.cs.kieler.scg.processors.persistentStateOptimizer
30 post process de.cau.cs.kieler.scg.processors.partialAssignmentEvaluation
31 post process silent de.cau.cs.kieler.scg.processors.cleanupValuedObjects
32 post process de.cau.cs.kieler.kicool.deploy.variable.store.clean
```

Listing 5.2.1. Textual description of the netlist-based compilation system in KiCo

are not commuting. Commuting means that the order in which they are executed does not matter.

Figure 5.2.2 shows an example of concurrent threads. In the inner threads, *Example* $y = 1$ and $y = 2$ are conflicting, because they are concurrent but not commuting. However, they do not conflict with $y = 3$, because they are not concurrent to $y = 3$. Similarly, $x = 1$ and $x = 2$ are concurrent and share the outer-most fork node as LCAF.

5. Interactive Compilation for SCCharts

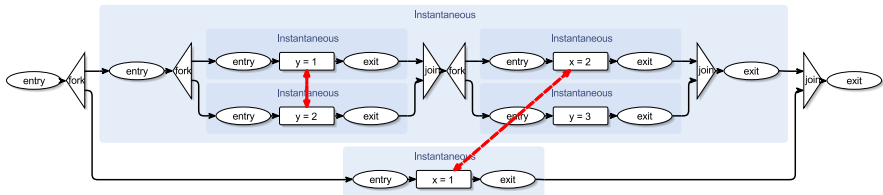


Figure 5.2.2. Concurrent accesses share an LCAF

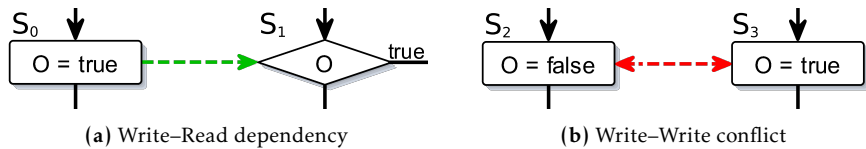


Figure 5.2.3. Write-Read and write-write dependencies

The inspection of dependencies is a typical use-case for interactive compilation, which is rather difficult for traditional *black box* compilers. Although VDSLs may profit more from the interactive approach than classical textual programming languages due to existing syntheses, Chapter 6 also gives examples for traditional programming.

Implementation

It is advantageous to be able to toggle between all, only concurrent, and conflicting dependencies when inspecting **and storing** SCG models with dependency information. Storing is a decisive factor, because big data of large projects that may not even be relevant for the downstream compilation can cripple the performance and hence, user experience. The same is true if a dedicated dependency view shows every single dependency even though they are neglectable w.r.t. scheduling. One example is the railway project, which will be discussed in more detail in Section 7.3.2. In KIELER, these different behaviours can be toggled via the environment configuration dynamically.

It is sufficient w.r.t. scheduling to categorize all relevant accesses w.r.t. to the scheduling into write-read and write-write accesses. Write-read accesses impose that the writer has to be scheduled before the reader. Write-write dependencies resemble a conflict, because no order can be established. It

5.2. Netlist-Based Compilation

can be sometimes helpful for a modeller to differentiate even more between absolute write, relative writes, and reads, however, w.r.t. to scheduling it is sufficient to establish a single *before* order, denoted $n_0 \rightarrow n_1$ with $n_0, n_1 \in \mathbb{N}$, meaning n_0 is scheduled before n_1 .

Figure 5.2.3 shows different kinds of concurrent dependency relationships. Figure 5.2.3a depicts a write–read dependency going from S_0 to S_1 , meaning that the assignment in S_0 must be scheduled before the read in the conditional of S_1 . The control-flow edges are also indicated as black incoming and outgoing edges. Any preceding nodes must be scheduled before their successors, and any succeeding nodes cannot be scheduled before their predecessor. Figure 5.2.3b shows a write–write conflict. Both concurrent assignments assign different, i. e. non-commuting, values to the same variable. No order can be established and the program has to be rejected.

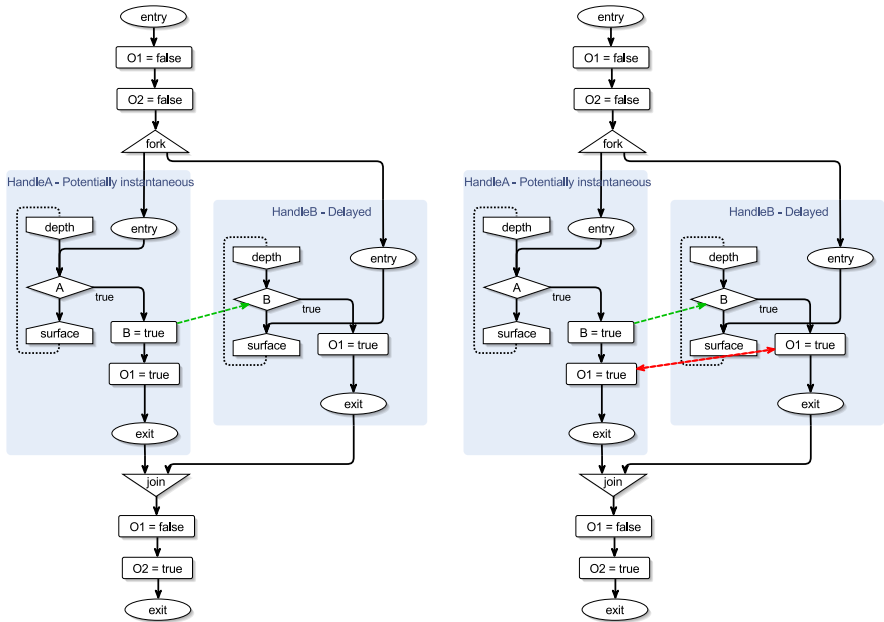
The SCG has to be traversed once to find all assignments. Dependencies are added to the model after the search for accesses that share the same variable. They hold information about the *concurrency* and *commuting* properties to facilitate the subsequent scheduling of the underlying MoC.

In KIELER, a modified version of the DFS which considers the structure *Implementation* of an SCG is used, which is explained in Section 5.2.2. Technically, LCAF information can be stored on a stack when entering a fork node, which is removed again after the accompanying join node is reached. This accelerates the determination of the concurrency property of the dependencies. —

Loop Detection Since the final netlist is not permitted to comprise any circular dependencies, it is mandatory to find all dependency loops. However, not every loop on SCG level is forbidden.

Figure 5.2.4 shows the SCG of ABO annotated with concurrent dependencies. *Example* Figure 5.2.4a illustrates a view showing only data dependencies which are relevant for the subsequent scheduling, i. e. concurrent non-confluent dependencies: $B = \text{true}$ must be scheduled before the test of B . In Figure 5.2.4b all concurrent dependencies are visible. The two assignments $O1 = \text{true}$ create a dependency loop. However, the cycle between the two nodes is benign because the accesses are commuting. The program would be rejected otherwise. —

5. Interactive Compilation for SCCharts



(a) ABO SCG as shown in KIELER after the dependency analysis step

(b) ABO expressed as SCG with all concurrent access dependencies

Figure 5.2.4. Dependencies shown in the SCG of ABO

Example

Figure 5.2.5 shows further views on the SCG of ABO. Figure 5.2.5a illustrates all dependencies, which might be helpful during debugging but becomes confusing fast and is usually not reasonable for common modelling. Figure 5.2.5b shows the same dependency as in Figure 5.2.4a but automatically layouted according to the dependency direction. The layouted view on the SCG gives an overall idea about dependencies between threads similar to the causal dataflow view presented in Section 4.1. In fact, both dependency processors build on the same analysis in the KIELER implementation.

While pair-wise relationships between the nodes of a dependency are established easily, immediate cyclic behaviour across multiple nodes with or without data dependencies involved, is also possible.

5.2. Netlist-Based Compilation

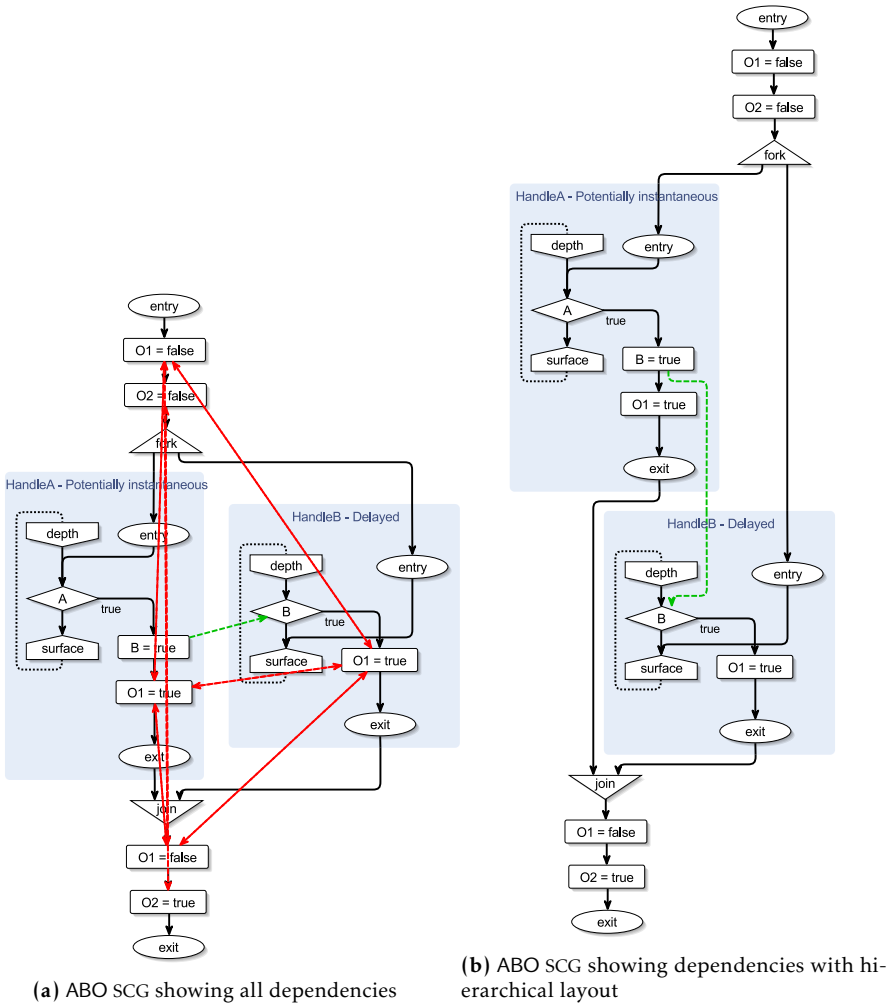
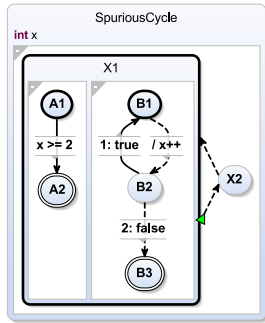
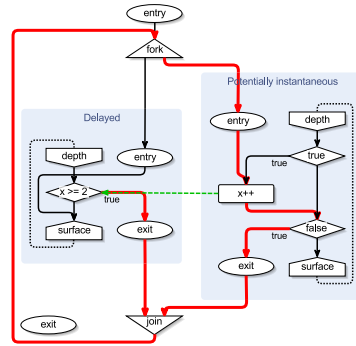


Figure 5.2.5. Further dependency views on the SCG of ABO

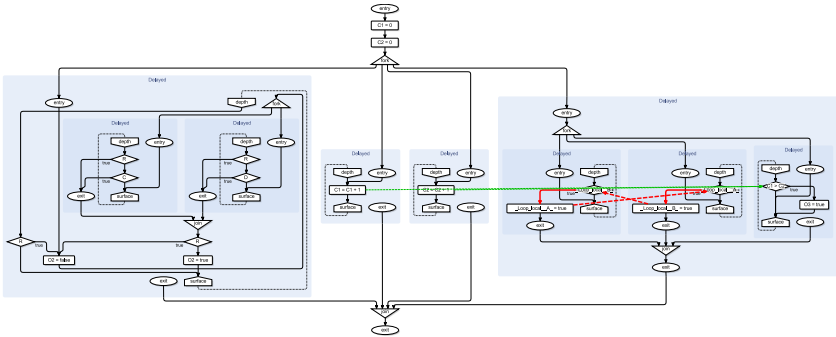
5. Interactive Compilation for SCCharts



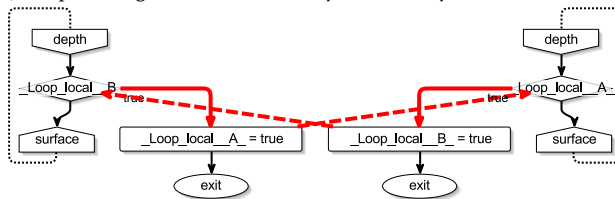
(a) SCCharts model containing a spurious cycle



(b) Detected loop in the SCG



(c) Loops in larger models; too many details may obfuscate the issue.



(d) Automatically synthesized view with heuristically extracted loop; important information is stored automatically during the compilation

Figure 5.2.6. Visualization of a detected loop within the compilation chain

5.2. Netlist-Based Compilation

Figure 5.2.6 shows how detected loops can be visualized inside KIELER. *Example* The SpuriousCycle model in Figure 5.2.6a shows a program which is semantically good but at first unschedulable by the netlist-based approach. The program increments x in one thread and waits for x reaching 2 in another. An immediate cycle, which is never taken, creates a feedback loop. The cycle is depicted in Figure 5.2.6b in red. Even without the dependency on the two accesses to x , the program would not be schedulable due to the control-flow cycle. Section 5.2.3 explains how this program can be scheduled anyhow as it is constructive under the SCMoC. Furthermore, expressions which can be evaluated statically can also be replaced and affected nodes may be removed, explained in Section 5.2.8.

Figure 5.2.6c shows an SCG of a larger model. While in this case, standard navigation and zooming capabilities may be enough to find potential problems in the graph, the overview gets worse as the graph grows. The loop detection, and any other processor, can extract parts of a model and save it for inspection inside the environment of the processor. Figure 5.2.6d shows a heuristically extracted section of the problematic graph. It includes all nodes of the cycle and their neighbours, which will prompt the modeller to the issue. Both views are accessible interactively.

In KIELER, the loop detection is implemented using Tarjan's algorithm [Tar72]. The running time is linear in the number of edges and nodes of the graph. *Implementation*

5.2.2 Basic Blocks

The netlist-based compilation approach converts all control-flow, be it sequential or concurrent, into a sequence of *guarded commands*. As a consequence, instantaneous loops cannot be handled with this code generation approach, as previously mentioned.

A *guarded command* is a statement which gets executed in the current tick if and only if the associated boolean *guard* evaluates to true in the current tick. Guards have a unique value throughout each tick. *Definition*

To economize on the number of guards, the approach makes use of the standard concept of Basic Blocks (BBs). BBs denote sets of statements which are executed together in a tick, i. e. either all or none of them are executed.

5. Interactive Compilation for SCCharts

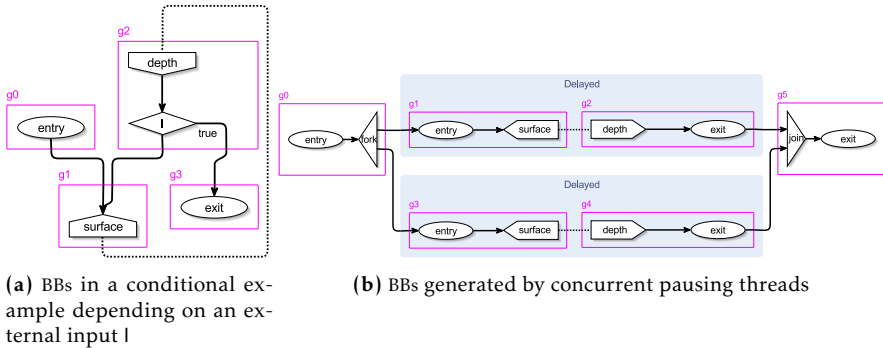


Figure 5.2.7. Two different examples of basic blocks as imposed by the defined rules in Section 5.2.2. Note that the views of the compilation state are flexible w.r.t. the orientation of the graph.

Thus, all statements within a BB share the same guard.

Definition In the context of the SCG, BBs are defined by the following rules:

1. A BB begins if the SCG node of a statement has two or more incoming control-flow edges or at the start of the program.
2. A BB ends with a statement which has two or more outgoing control-flow edges, or at the end of the program.
3. BBs are split at pause statements.
4. SCG fork nodes close a BB, whereas join nodes start a new one.

Consequently, any node of a given SCG can only be included in one BB at any time. On 4., note that forks do not necessarily have to fork more than one thread and, therefore, would not need to be included in their own BBs. However, the assumption that all fork and join nodes are included in own BBs eases the downstream compilation since guards of join blocks have to be treated differently, which will be explained in Section 5.2.2. These nodes are semantically superfluous and can be resolved modularly by the compiler framework beforehand, as will be shown in Section 5.2.8.

Example Figure 5.2.7 shows two BB partitions. In Figure 5.2.7a one can see an SCG of a program which waits until an input I arrives. The BB partitioning starts at the beginning of the program and is immediately followed by a new BB

5.2. Netlist-Based Compilation

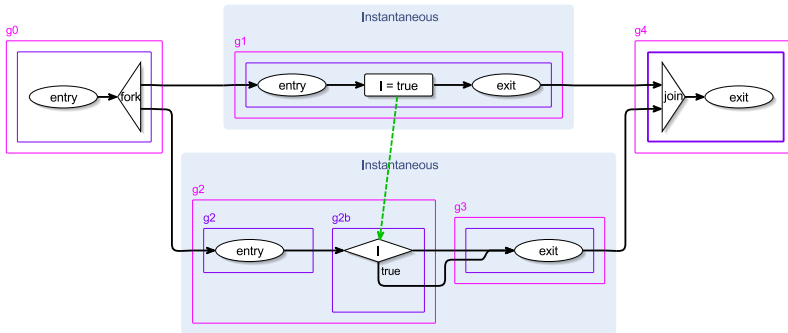


Figure 5.2.8. Example of two SBs dividing a BB

after the first node, because the subsequent node has two incoming control-flows. This block, $g1$, also only consists of one node since it is the surface of a pause. $g2$ has two nodes: The starting depth node is followed by the conditional node which tests for l . The program ends in $g3$. The name of a BB also denotes its guard name and is, therefore, prefixed by a g followed by a unique integer. The second example in Figure 5.2.7b shows the separation of the BBs due to fork and join nodes. The program forks two threads, which wait concurrently for one tick to occur.

The statements in a BB are not necessarily executed strictly subsequently. The execution of BBs may be interleaved with each other in order to satisfy dependencies induced by shared variables as described in Section 5.2.1 on page 126. Therefore, BBs are further divided into Scheduling Blocks (SBs).

Scheduling Blocks (SBs) subdivide a BB at incoming concurrent dependencies. *Definition*

Figure 5.2.8 shows a program which is partitioned by SBs. The program forks two threads. One thread emits l , while the other thread is waiting for l to become true. As before, the BBs are created according to the defined rules. Additionally, as the dependency edge targets the conditional node within the BB $g2$, the block is split into two SBs. This enables the compiler to schedule whole units of SBs without further processing. *Example*

For example, as the program starts, $g0$ is executed. Afterwards, a scheduler may decide to run $g2$ before $g1$, as both threads run concurrently. How- *Example*

5. Interactive Compilation for SCCharts

ever, the control-flow can only proceed until it reaches the conditional node in g_{2b} since the IURP requires write accesses to be scheduled first. Therefore, after the first SB has been executed, the scheduler has to reschedule to g_1 , before BB g_2 has finished its execution. It can then finish the execution of g_1 , before returning to g_2 and eventually g_3 and g_4 . The depicted scheduling order can be also expressed in short form: $g_0 \rightarrow g_2 \rightarrow g_1 \rightarrow g_{2b} \rightarrow g_3 \rightarrow g_4$. In this case, it might have been more efficient to schedule g_1 before g_2 in the first place, which can be established by the directions of the inter-thread dependencies.

Note that while additional SBs are suffixed with an extra letter so that they can be referenced, there is no need to evaluate the guard expression anew. All SBs are active if their parent BB is active, which is uniquely defined during the complete tick. SBs define the order in which they are executed and hence, where the statements will be situated in the corresponding sequentialization, which will be shown in Section 5.2.7. To emphasize that the complete BB should be executed before rescheduling, one can write $g_1 \rightarrow \overline{g_2} \rightarrow g_3$ instead of $g_1 \rightarrow g_2 \rightarrow g_{2b} \rightarrow g_3$. Also, explicit scheduling between threads can be denoted with a curved arrow: $g_0 \rightarrow g_2 \curvearrowright g_1 \curvearrowright g_{2b} \rightarrow g_3 \rightarrow g_4$

Eventually, as SBs mark the points where a scheduler might want to reschedule, they form the possible execution orders of the program. As SBs are only created due to incoming concurrent dependencies, all BBs without such only consist of one SB. This sounds natural since the whole block can be executed as one unit and must not fear an interruption of the scheduler. In Figure 5.2.8, as the program is instantaneous, all guards $g_0 - g_4$ evaluate to true, but the order of their execution, e. g. $g_0 \rightarrow g_1 \rightarrow \overline{g_2} \rightarrow g_3 \rightarrow g_4$, induced by concurrent dependencies, matters for the correct behaviour.

Conclusion BBs define **which** set of statements becomes active in a tick, whereas SBs define **when** a particular statement set is executed during a tick.

Basic Block Analysis The analysis of the SCG can be done straightforwardly in linear complexity using a DFS or BFS. Each node is visited once and basic blocks are created on-the-fly as soon as aforementioned rules apply.

Implementation In the SCCharts compiler, the generation of BBs is capsuled inside a dedicated processor which works on the SCG meta-model. To achieve a reasonable naming distribution, i. e. that threads are numbered sequentially

5.2. Netlist-Based Compilation

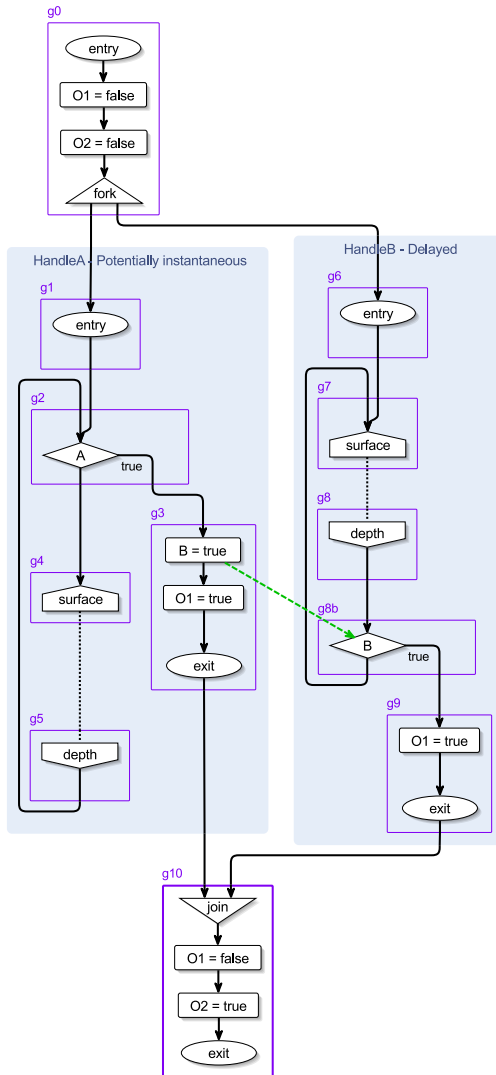


Figure 5.2.9. SB partitioning of ABO

5. Interactive Compilation for SCCharts

```
Algorithm JDFS(node) {
  visited[node] = true
  stack = switch type(node) {
    case Exit:  $\emptyset$ 
    case Fork: node.join + allNext(node)
    else: allNext(node)
  }
  while (stack is not empty) {
    node' = pop(stack)
    if (!visited[node']) JDFS(node')
  }
}
```

Listing 5.2.2. Joined Depth-First Search (JDFS) algorithm

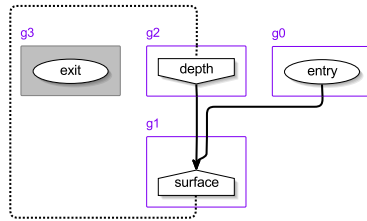
but before being joined again, a modified DFS is used: the Joined Depth-First Search (JDFS). The algorithm, shown in Listing 5.2.2, does not follow the control-flow to the join node after the exit node of a threads has been processed. Instead, the fork will continue the search with the join node after all threads reached their exit nodes.

Example Figure 5.2.9 shows the SB partitioning of the SCG of the ABO program. Assignments which can be executed together without interruption from the scheduler are included in one SB, as can be seen in `g0` and `g3`. `g8` is split due to the incoming dependency edge, which can also be determined immediately while the BBs are constructed.

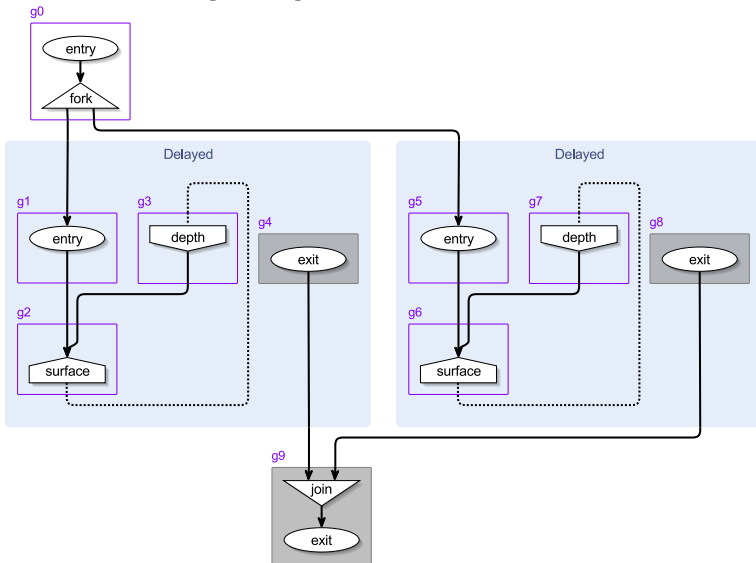
JDFS always terminates in a well-formed SCG. If threads do not terminate, the subsequent nodes are not reachable through the control-flow and the blocks are considered *dead*. The processor can create dead BBs for visualization reasons by running JDFS on the set of nodes which are not already contained in other BBs, which can be seen in Figure 5.2.10a. The *deadness* property propagates. If JDFS continues at a join node and none of the incoming exit nodes is alive, the BB of the join is also considered dead, as depicted in Figure 5.2.10b.

Basic Block Expressions In every tick instance, the guard of a BB may be *active* or *inactive*. The activity state of a BB depends on the preceding BBs.

5.2. Netlist-Based Compilation



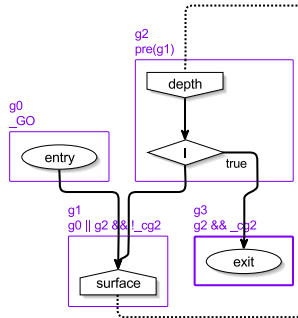
(a) Simple example of a dead BB, here g3



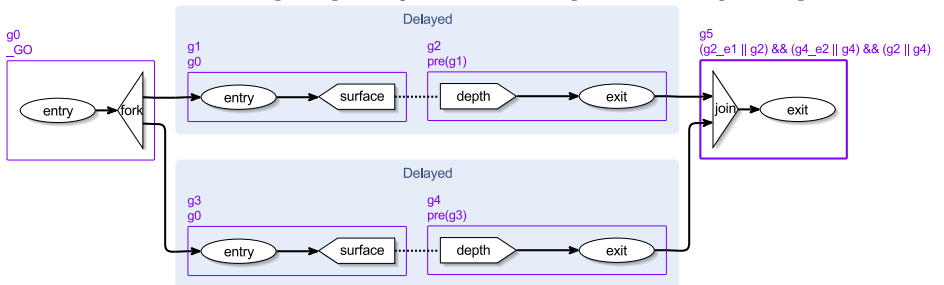
(b) Example of a dead join BB, g9

Figure 5.2.10. Two different examples of dead BBs

5. Interactive Compilation for SCCharts



(a) BBs in a conditional example depending on an external input I with their guard expressions



(b) BBs generated by concurrent pausing threads with their guard expressions

Figure 5.2.11. Two different examples of BB guard expressions

Definition A *Simple Guard* depends solely on its direct predecessors and, in the case of an incoming control-flow from a conditional statement, on potentially external further expressions, such as inputs from the environment. The guard of the first BB in an SCL program depends on the GO start signal of the environment usually emitted at the initialization or reset of the program. Guards of preceding basic blocks which contribute to the guard expression are called *activators*. The *guard expression* is the disjunction of all preceding activators.

Generally speaking, a BB is active if and only if at least one of its predecessors is active in the same tick.

5.2. Netlist-Based Compilation

Figure 5.2.11 illustrates the conditional and fork programs from Figure 5.2.7 with their guard expressions. In Figure 5.2.11a one can see that g_0 depends on the GO signal. The guard becomes active when the program is started. BB g_1 has two incoming control-flows and, therefore, depends on g_0 and g_2 . Its guard becomes active if g_0 is active **or** if g_2 is active and the conditional inside g_2 evaluates to false. Thus, the guard expression of g_1 is $g_0 \parallel g_2 \ \&\& \ !_cg_2$ with the *logical and* having a higher precedence than the *logical or*. BB g_2 starts with the depth of a pause statement. The activation state of its guard depends on the state of the guard of the BB of the corresponding surface. The guard expression is $\text{pre}(g_1)$, meaning that it is only true if the guard of the surface BB was true in the last tick. Finally, guard g_3 becomes true if g_2 is true and the conditional's condition evaluates to true. *Example*

Expression parts prefixed with $_c$ denote condition evaluations. It is important to save the results temporarily, because their state may change during the execution of the true or else branch of the conditional, but before the other branch was reached inside the netlist. The condition's state must be unique for each conditional.

Forked threads must be joined at some point in time in SCL programs if the joining BB is not dead. The join statement will not proceed unless each thread has finished. The guard of a BB which contains a join statement cannot be expressed by simple guards.

A *complex guard* is the conjunction of a simple guard and a set of further conditions to become true. *Definition*

To join concurrent threads in SCL and set the associated guard to true, it is checked if all threads have terminated and at least one must have been terminated in this tick. Otherwise, the control-flow is not permitted to proceed from the join. Each thread status is signalled by an *empty flag* which describes whether or not a thread is inactive. All empty flags are combined in a conjunction together with a combination of exit codes which signal whether at least one thread terminated in this tick instance. BBs which are responsible for joining threads are also called *synchronizers*. The set of empty flags form the further conditions of the complex guard of a synchronizer. They are the only complex guards in SCL. The construction of the synchronizer described here is similar to the synchronizer circuit explained in by Berry [Ber92].

5. Interactive Compilation for SCCharts

```
Algorithm simpleGuardExpression(bb) {
    expr = ∅
    for each (predecessor in bb.predecessors) {
        expr = expr || guardedBy(predecessor, bb)
    }
}

guardedBy(bb, successor) {
    expr = switch type(bb) {
        case surface: createPre(bb.guard)
        case conditional: bb.guard &&
            if bb.trueBranch == successor then
                bb.condition
            else
                not(bb.condition)
        else: bb.guard
    }
}
```

Listing 5.2.3. Simple guard expression algorithm

Example Figure 5.2.11b shows the complex guard expression of the BB with the join node in g5. The expression reads $(g2_e1 \parallel g2) \ \&\& \ (g4_e2 \parallel g4) \ \&\& \ (g2 \parallel g4)$ with the parts suffixed by $_en$ holding the empty flags of the corresponding threads. The guard names of the exit nodes of the threads are used for the empty flags to gain some similarity to the terminating guards. $(g2 \parallel g4)$ stand for the *simple* part of the complex guard and reflect the incoming control-flow, meaning that at least one of the threads must exit in this tick to activate the BB. $(g2_e1 \parallel g2)$ and $(g4_e2 \parallel g4)$ say that the threads in question must have been empty, i. e. must have already terminated previously, or is terminating momentarily. If all conditions are met, the guard evaluates to true.

The guard expressions can be calculated statically by visiting each BB once, depicted in the algorithm in Listing 5.2.3. Different kinds of synchronizer for the complex expressions are shown in Section 5.2.2.

Example The guard expressions generated for the ABO example are shown in Fig. 5.2.12. While constructed in the same way as in the previous examples, one can now see SBs containing multiple assignments in g0, g3, and g10. g8 is split due to the incoming dependency edge and g10 holds the join node.

5.2. Netlist-Based Compilation

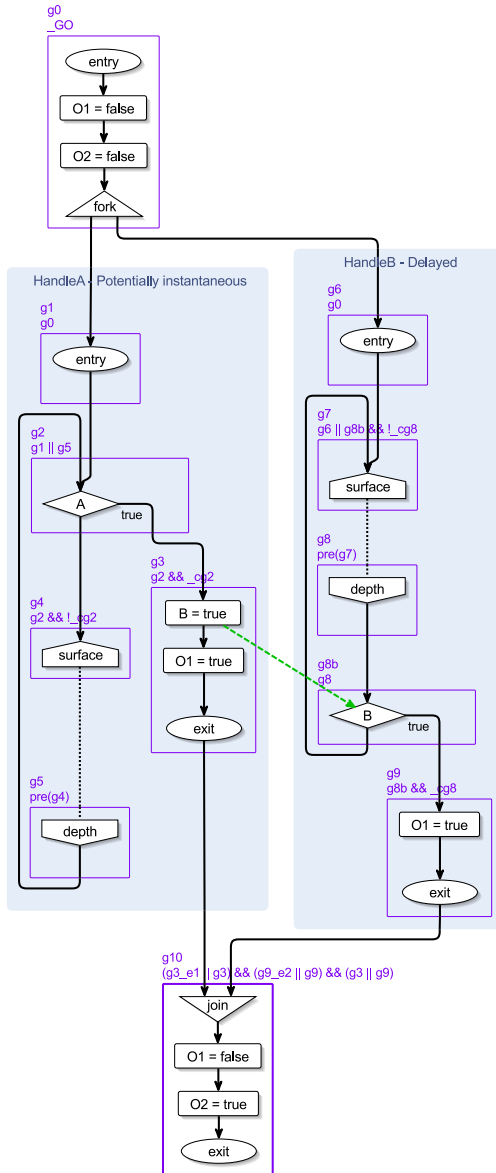


Figure 5.2.12. SB partitioning of ABO with guard expressions

5. Interactive Compilation for SCCharts

Remark Not all BBs need a dedicated own guard. At times, their activation can be purely determined statically at compile-time if they solely depend on preceding BBs without further inputs from the outside. KIELER uses standard compiler techniques, such as *copy propagation* [ASU86], to reduce the amount of needed guards, which is discussed further in Section 5.2.8.

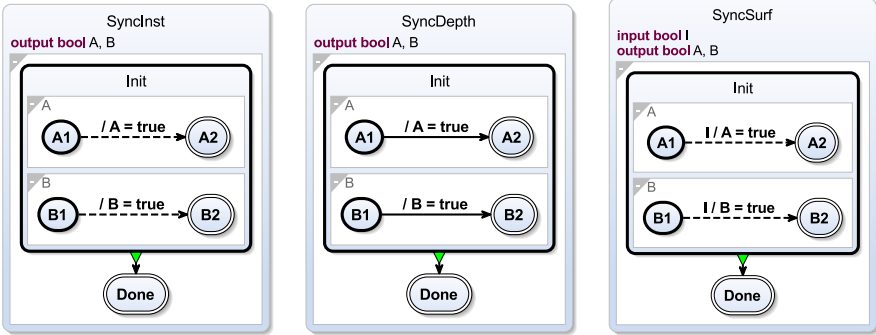
The Synchronizer Synchronizers are required to join threads. They generate the guard expression for BBs that comprise a join node. Since non-trivial joins depend on the actual activity states of different threads, their BBs are activated by a complex guard. Trivial joins, such as joins that join a single thread, can be optimized by eliminating the fork and join nodes, as is shown in Section 5.2.8. I here propose three different synchronizers for joining threads, an instantaneous, a surface and a depth synchronizer. The list can be expanded modularly as required.

A synchronizer guard expression evaluates to true if all threads have reached the end of their control-flow, i. e. their exit node. This can happen in the current tick instance, or for a true subset of the joining threads also in previous ticks. A basic formula for this was already shown in the concept matrix in Figure 5.1.2 on page 121. A join BB becomes active if all threads are *empty* and at least one thread exits in this tick. Empty means that no pause statement within the thread is no longer active.

Instantaneous synchronizers: The synchronizer guard is relatively trivial for instantaneous threads. If all threads are instantaneous, as in e. g. Figure 5.2.13a/d, the join node will be active in the same tick as the fork node. The subsequent scheduler only needs to respect data dependencies to make sure that sequential variable accesses happen in the correct order.

Depth synchronizers: If the threads are delayed, as in e. g. Figure 5.2.13b/e, a depth synchronizer collects the *empty flags* for each thread. In this case, an empty flag is the conjunction of all negated depth guards. They are not directly depicted in the BB view, but they will be shown similar to all other assignments in the generated netlist view in Section 5.2.4. A thread is terminated if the empty flag is true or it was exited in this tick. If all threads are terminated and at least one was left in the current tick, the join guard expression evaluates to true.

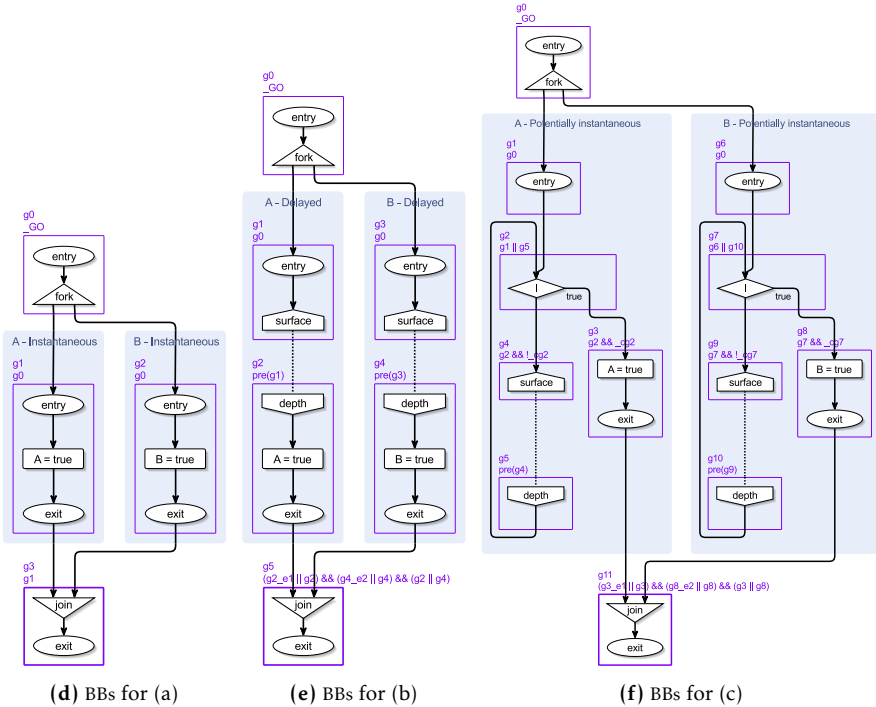
5.2. Netlist-Based Compilation



(a) OSM of an instantaneous synchronizer

(b) OSM of a depth synchronizer

(c) OSM of a surface synchronizer



(d) BBs for (a)

(e) BBs for (b)

(f) BBs for (c)

Figure 5.2.13. Different synchronizers

5. Interactive Compilation for SCCharts

Surface synchronizers: If the threads are potentially instantaneous as in e.g. Figure 5.2.13c/f, it is not sufficient to look at the depths of the pause statements, because if one thread exits in the first tick and the other enters its pause, the guard evaluates to true since one threads exits and the other will still be recognized as empty. In this case, the synchronizer can look at the surfaces instead of the depths of the pauses statements.

For two threads t_1 and t_2 , the synchronizer expressions can be summarized as follows, which completes the transformation matrix in Figure 5.1.2:

$$\begin{aligned}g_{inst}(join) & ::= g(join.fork) \\g_{depth}(join(t_1, t_2)) & ::= (empty_{depth}(t_1) \parallel g(t_1.exit)) \&\& \\ & \quad (empty_{depth}(t_1) \parallel g(t_1.exit)) \&\& \\ & \quad (g(t_1.exit) \parallel g(t_2.exit)) \\g_{surf}(join(t_1, t_2)) & ::= (empty_{surf}(t_1) \parallel g(t_1.exit)) \&\& \\ & \quad (empty_{surf}(t_1) \parallel g(t_1.exit)) \&\& \\ & \quad (g(t_1.exit) \parallel g(t_2.exit))\end{aligned}$$

5.2.3 Curing Schizophrenia

Schizophrenia is a phenomenon in synchronous languages where statements (or scopes) are executed (resp. entered) more than once within the same tick instance. While this is not an issue in pure software synthesis, hardware approaches must cope with this phenomenon with care, because netlists (and circuits) can only have uniquely defined values for their guards (resp. wires) within a tick.

Example The model depicted in Figure 5.2.14a comprises two regions. Region A is delayed and region B is potentially instantaneous depending on the input l . The schizophrenic behaviour becomes apparent if considering the second and third tick of the model simulation. The tickline is shown in Figure 5.2.14d. In the second tick in Figure 5.2.14b, region A terminates because of the delayed transition while region B still remains in its pause, because l is still false. Hence, output A is now 1 and B is still 0. If l now becomes true in the third tick, as depicted in Figure 5.2.14c, B++ is executed

5.2. Netlist-Based Compilation

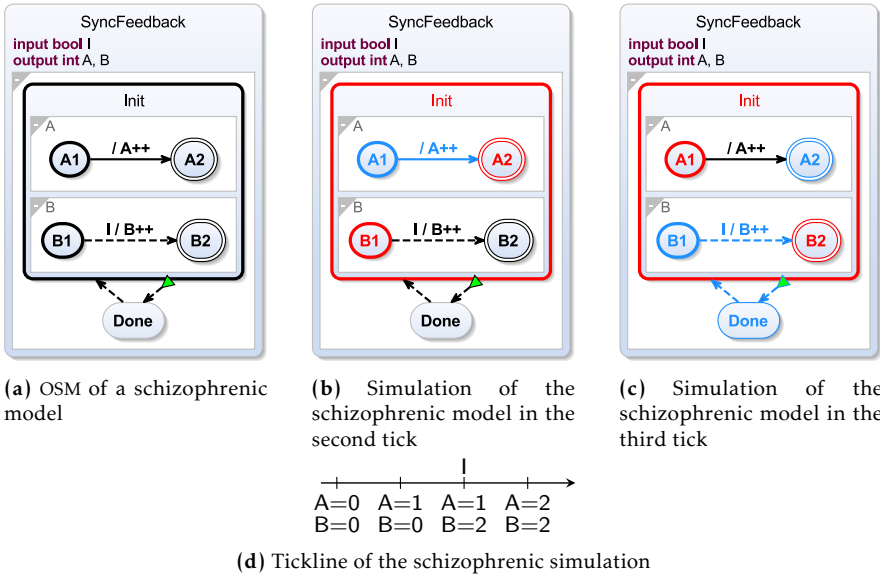
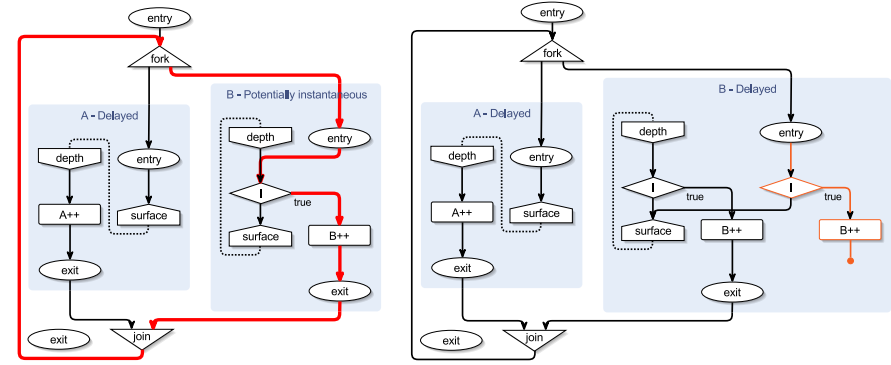


Figure 5.2.14. Schizophrenia in SCCharts

and the region also terminated. Due to the immediate feedback, the `Init` state is entered again and `B++` is executed a second time, because the transition from `B1` to `B2` is immediate and `I` is still true. Thus, the single action `B++` must be executed twice: `A` is still 1 but `B` is 2 at the end of the third tick. In the fourth tick, `A` is also incremented to 2.

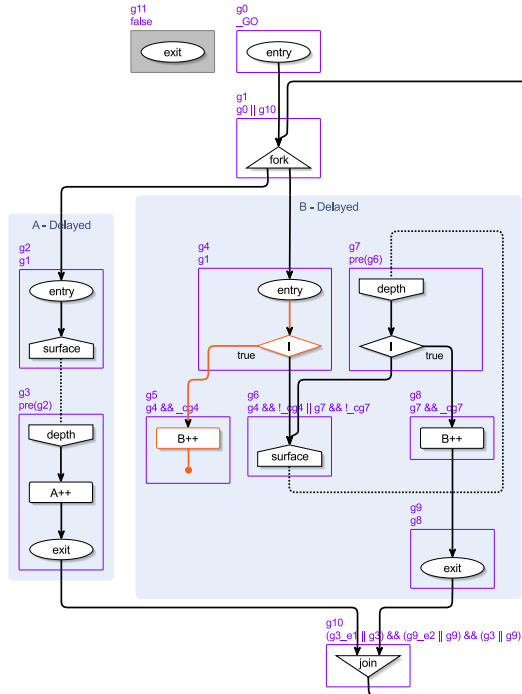
The surface synchronizer, which was used for potentially instantaneous threads in Section 5.2.2, is not viable when handling *immediate feedback* since it would create a circular netlist: The BBs of the surfaces would depend on the fork node BB, the join BB would depend on the surfaces, and due to the feedback the fork node would be a successor of the join. However, as hardware compilation strategies, such as the netlist-based approach, prohibit immediate instantaneous cycles, one can assume that at least one concurrent region must be delayed and the depth synchronizer can be used.

5. Interactive Compilation for SCCharts



(a) Detected instantaneous loop within the schizophrenic model

(b) Surface duplication via SDJ



(c) BBs are applied as usual after the SDJ

Figure 5.2.15. Curing schizophrenia via a Structural Depth Join (SDJ)

5.2. Netlist-Based Compilation

Figure 5.2.15a shows the instantaneous loop. The conditional check of l and the increment of B are both on the *critical path*, which is the path in the surface area of thread B which immediately connects the entry and the exit node. All statements on this path are potentially executed twice. There are two scenarios to consider: (1) l is true when entering B . Since one assumes that at least one thread must be delayed, the thread will not contribute to the termination, because the pause in the other thread will stop the execution. Therefore, thread B can simply execute its surface and terminate silently. (2) l is false, in which case the thread will enter its depth and proceed as if it was delayed itself. However, if l will become true in the future, the depth and afterwards scenario (1) will be executed in the same tick as explained previously. *Example*

I propose to solve this schizophrenia problem with a Structural Depth Join (SDJ) processor, which duplicates the nodes on the critical path without connecting the path to the exit node as depicted in Figure 5.2.15b. Therefore, the critical surface path does not contribute to the guard expression of the join BB although the thread instance still exists. Both the l conditional and the B increment are duplicated and can thus be executed twice per tick in the netlist (resp. circuit). Thereby, the thread cannot be exited instantaneously any more and gets marked as delayed for the synchronizer. In the critical scenario order (2) and then (1), the control-flow will rest at the (black) conditional in the depth. As soon as l becomes true, (black) $B++$ is executed and the thread is left. It is then entered again via the feedback loop and (orange) l is checked again. Still true, (orange) $B++$ is executed and the thread dies, because the overall fork/join will be stopped by the concurrent thread anyhow. Since all threads are considered delayed after the application of the SDJ processor, the BB s can be generated as before using the depth synchronizer for the join. They are depicted in Figure 5.2.15c for this example.

As shown in the following sections, the order of instructions in the netlist resembles their potential execution order during a tick. Therefore, as peculiarity w.r.t. SDJ schizophrenia handling, the statements in the surface of schizophrenic thread will appear at the end of the netlist, as they will be executed after the depth statements, which will occur at the top of the netlist.

5. Interactive Compilation for SCCharts

Remark It is also possible to apply the schizophrenia handling after the usual BB generation purely on expression level. The explained order was chosen to facilitate understanding since the single steps can be depicted via KiCo. It also fits naturally into the existing compilation chain. Preceding and succeeding processors remain unaffected and hence, SDJ can be enabled if needed.

Remark The SDJ is a light-weight concept for solving schizophrenia, which fits nicely into the atomic compilation steps of the proposed approach. It manages the most prominent schizophrenia cases in SCCharts' netlist-based compilation. However, the *delayed thread* assumption is relatively strong. Duplication on circuit level [PH95; Ber02] or program rewriting [SW01; TS03] accept a broader class of schizophrenic programs, i. e. programs with multiple reincarnations in the same tick.

5.2.4 The Netlist

With all guard expressions generated, the compiler can now generate the netlist. I propose to use a sequentially constructive version of the Program Dependency Graph (PDG) [FOW87], or SCPDG in short, for this. The SCPDG is a program dependency graph, which also includes tick boundaries of pause statements, and encodes different types of dependencies according to the established notations to facilitate understandability.

Figure 5.2.16 shows an annotated excerpt example SCPDG, which contains all possible dependencies. Direct *expression dependencies* are shown as brown, solid directed edges. They represent dependencies between the guard expressions of the BBs. A few *control dependencies* remain in the graph, depicted as blue, dotted edges. These usually represent the sequential operator in effects and also make sure that the conditional variables are checked at the position of the corresponding BB in the netlist. While sequential data dependencies could be managed by the dependency analysis, as explained in Section 5.2.1, I propose to use control dependencies, because the model could include host code calls in effects and conditions, which may be beyond the *whitebox* dependency analysis. Hence, effects are executed in the order of their appearance in the model. The *tick boundary dependency* is depicted as a dotted, grey edge and can be toggled on/off in the KIELER view. It is not a dependency which influences the execution order within a tick but indicates

5.2. Netlist-Based Compilation

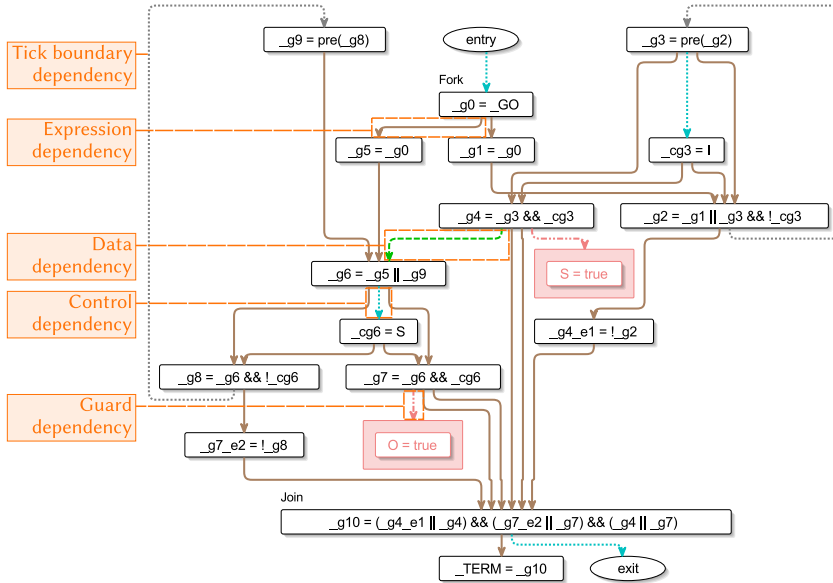


Figure 5.2.16. SCPDG excerpt with different kinds of dependencies (annotations added manually)

that the predecessor guards must be evaluated, because their values are required in the next tick. However, it is not required to be evaluated before the pre expression in this tick. Effects which are guarded, i. e. included in the SB of that guard, are illustrated within red frames. They are connected to the corresponding guard via a *guard dependency*, meaning that the effects guarded by this guard shall be executed if the guard evaluated to true. Their order is retained by control dependencies, when multiple effects are guarded. Common *data dependencies* between the SBs retain in the SCPDG and are depicted in the usual syntax: green, dashed dependencies indicate a write-before-read relationship; red edges indicate conflicts or immediate loops. Collectively, the depicted directed dependency edges form all possible schedules of the program, and thus, maximal parallelism.

5. Interactive Compilation for SCCharts

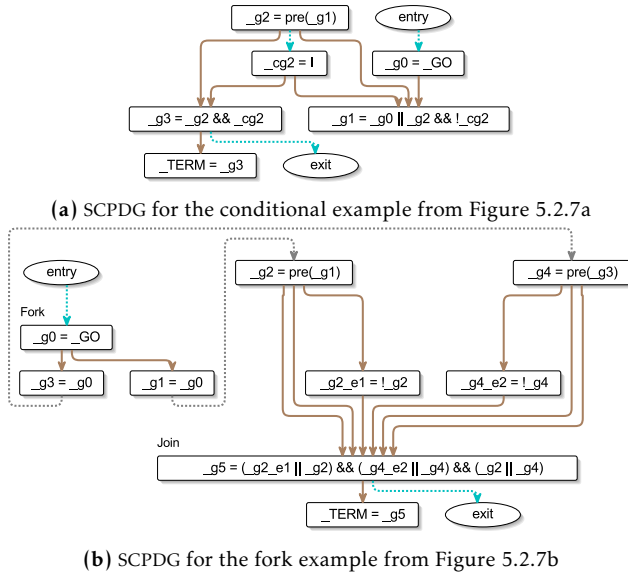


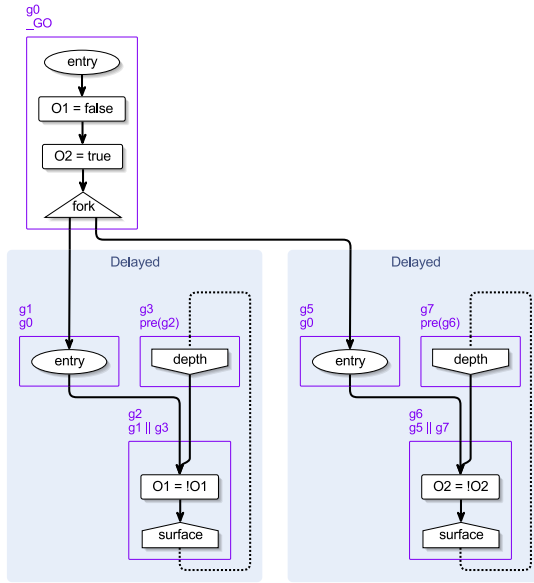
Figure 5.2.17. Two examples of the SCPDG

Example

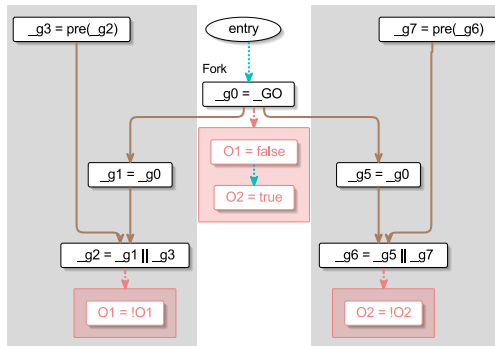
Figure 5.2.17 shows the two SCPDGs for the conditional and the fork examples. Basically, a new SCG containing the whole netlist is generated during transformation. Every guard expression is represented by an own assignment. An example for an expression dependency can be seen in the conditional example in Figure 5.2.17a between $_g2$ and $_g1$. A control dependency can be seen between $_g2$ and $_cg2$. $_g3$ is the predecessor of the $_TERM$ variable. Analogous to $_GO$, $_TERM$ signals to the environment which the program has terminated.

Figure 5.2.17b shows the netlist of the fork example from Figure 5.2.7b. There is no direct dependency between the surfaces and the depths of the pauses in the two threads, because the activation of the guards depends on the status of the predecessor blocks of the last tick. Furthermore, as with the conditional flag $_cg2$ in Figure 5.2.17a, the empty flags, here $_g2_e1$ and $_g4_e2$, are also visible in the SCPDG. Together with the simple activators $_g2$ and $_g4$ they form the immediate predecessors of the guard expression of the join BB.

5.2. Netlist-Based Compilation



(a) SBs and guard expressions of a sustain program



(b) SCPDG of the sustain program (grey areas for the threads are edited manually)

Figure 5.2.18. The sustain example

5. Interactive Compilation for SCCharts

Example Another example for the creation of the SCPDG is shown in Figure 5.2.18. The sustain program forks two threads and alternates two output variables O1 and O2 indefinitely. The SBs and the guard expressions of sustain are shown in Figure 5.2.18a. The control-flow in the threads loop continuously, and therefore, a join cannot be reached. Unreachable blocks can be hidden in the view. The SCPDG is depicted in Figure 5.2.18b.

After the first SB, which initializes O1 and O2, the SCPDG splits into two parts; one for each thread. Both threads immediately begin to toggle their variable and continue to do so in every following tick. The guards belonging to the same thread are grouped together in the figure for clarity reasons. Since there are no further inter-thread dependencies, both thread could run on separate cores after the fork. A corresponding processor would be able to — prepare the SCPDG accordingly.

The SCPDG of ABO is depicted in Figure 5.2.19a. Besides containing the previously mentioned expression, control and guard dependencies, ABO also contains the data dependency induced by the SCMOC. Due to this dependency, one can see that the guard potentially setting B to true, must be evaluated before the check of B in the other thread, represented by `_cg8`, potentially leading to the joining of the threads.

5.2.5 Scheduling

Generating a schedule for a constructed SCPDG can be made in a straightforward manner. Therefore, all nodes of the SCPDG are sorted topologically. Figure 5.2.19b shows a possible schedule of ABO. The path is coloured in purple. In comparison to the SCPDG in Figure 5.2.19a, the top-down execution now becomes visible in Figure 5.2.19b.

Note that there might exist multiple possible paths through the graph. The scheduler must determine an order in which the statements are executed, and therefore, reducing the parallelism of the SCPDG. Other schedulers can manage the task differently. For example, a scheduler which is able to manage disconnected components could initialize the netlist of the sustain example (Figure 5.2.18), but then fork off both independent threads with own schedules for true parallelism on, e.g. different cores. A tick-aware scheduler could guard effects together which are only potentially active in

5.2. Netlist-Based Compilation

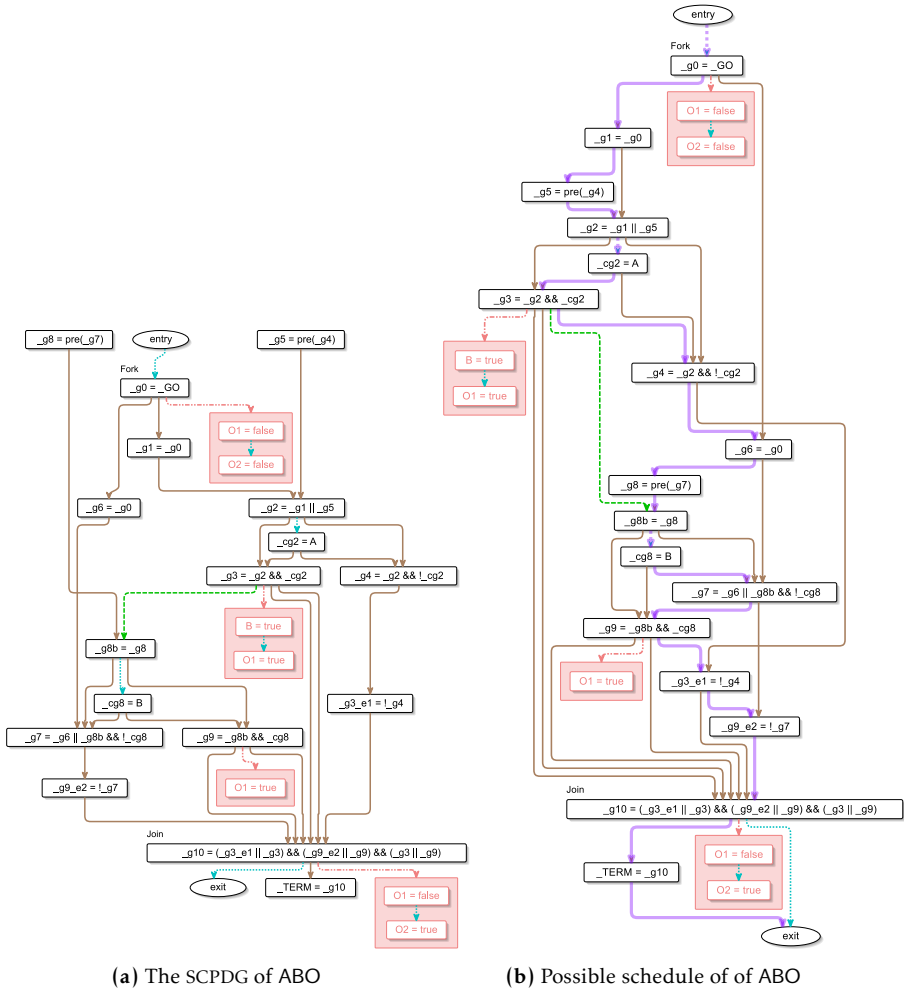


Figure 5.2.19. Scheduling of the ABO netlist

5. Interactive Compilation for SCCharts

certain ticks, and a thread-aware schedule could guard whole threads so that entire parts of the netlist are ignored if not active at the moment similar to GRC. Furthermore, the processor could also calculate all possible schedules in the scheduling step and then determine the best schedule by sending all to the optimizations and using a meter to judge the outcomes similar to the model-based measuring model in Section 3.3.6 on page 88.

5.2.6 SC+

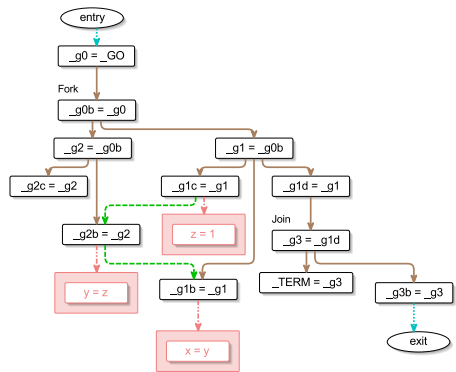
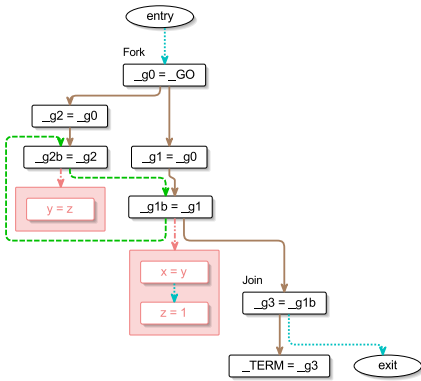
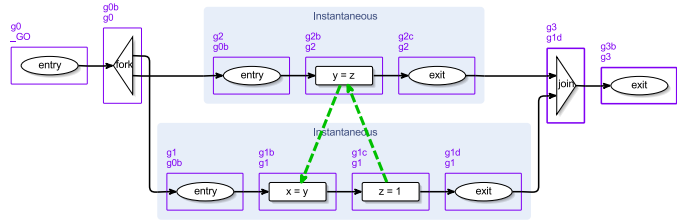
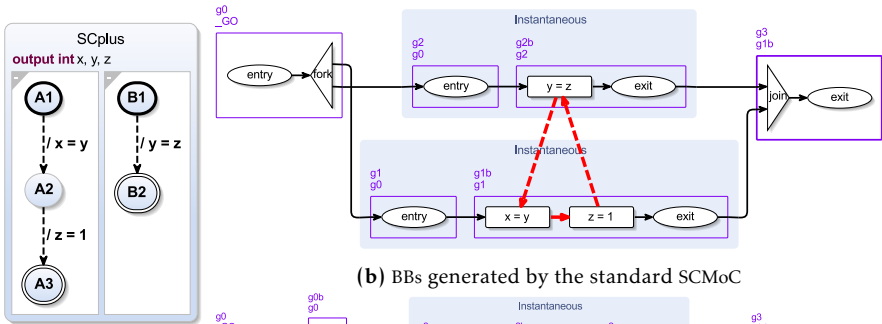
It is quite common to execute statements *out-of-order* if the underlying MoC allows it. The interactive compilation approach makes it easy to adjust the SCMoC of SCCharts.

Example Consider the SCplus model depicted in Figure 5.2.20a. In the model, x is first set to y and then, 1 is assigned to z , while concurrently z is assigned to y . The BBs according to the SCMoC are shown in Figure 5.2.20b. As depicted in red, the dependency analysis (see Section 5.2.1) will detect a dependency cycle following the IURP. The cycle is also visible in the SCPDG between `_g2b` and `_g1b` in Figure 5.2.20d and prevents scheduling.

The culprit in this example is the control-flow dependency between the assignments $x = y$ and $z = 1$. These assignments clearly commute and, therefore, there is no real reason to impose a particular order on them. A compiler may choose to reorder these statements at machine code level, or at an even lower level a processor may choose to execute these out of order. Thus, I here propose a conservative relaxation of the SCMoC, which I will refer to as SC+, which disregards such spurious control-flow dependencies. This allows to accept some programs, such as the example above, that would be rejected under the standard SCMoC.

SC+ is now implemented in KIELER and can be switched on, i. e. replacing the standard processor for BBs, if desired. In fact, the alteration only needs two additional lines of code. When toggled, SBs are also introduced on outgoing dependency edges and control dependencies within the BBs are ignored. Therefore, as shown in Figure 5.2.20c the two assignments, $x = y$ and $z = 1$, do not reside in the same SB anylonger and the control-flow between them is discarded when creating the SCPDG, which is illustrated in Figure 5.2.20e. Here, the dependency cycle is broken and the SCPDG shows

5.2. Netlist-Based Compilation



(d) Generated SCPDG for the standard SCMoC (e) Schedulable SCPDG following the rules of the extended SC+ semantics with depicted dependency cycle

Figure 5.2.20. Illustration of the SC+ semantics

5. Interactive Compilation for SCCharts

```

Algorithm Sequentialize(schedule) {
  newSCG = new SCG
  foreach (node in schedule) {
    newSCG.nodes.append(node)
    if (hasGuardDependencies(node)) {
      newSCG.nodes.append(createConditionalBranch(node))
    }
  }
  return newSCG
}

```

Listing 5.2.4. Sequentialization algorithm

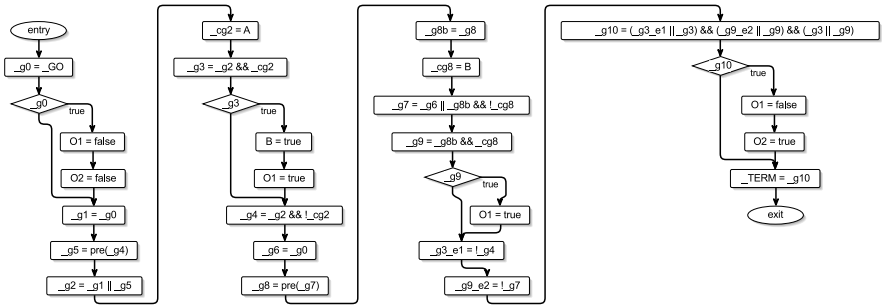


Figure 5.2.21. Sequentialized version of ABO

that z should be scheduled before $y = z$, which should be scheduled before $x = y$.

The SC+ semantics has similarities to dataflow semantics where the order of the control-flow does not matter. Concurrent assignments are ordered according to the dataflow and any sequentiality is absent. Another model-level dataflow approach will be discussed in Section 6.2.

5.2.7 Sequentialization

Once a schedule is determined for an SCPDG, the program can be sequentialized accordingly. *Sequentialization* takes the predetermined schedule and creates a new program, which is in this case represented as a new SCG. The construction rules, shown in Listing 5.2.4, are relatively simple. Look at ev-

5.2. Netlist-Based Compilation

Structural Optimization	LoC	Semantic Optimization	LoC
Copy Propagation	106	Superfluous Fork Remover	46
Conditional Merger	118	Superfluous Thread Remover	24
Register Relocation	171	Halt State Remover	101
Partial Assignment Evaluation	153	Persistent State Optimizer	76
		_GO MoC Optimization	86
		!_GO MoC Optimization	44

Table 5.2.1. KIELER netlist-based optimizations and their code sizes

ery node on the schedule once and place it in scheduling order sequentially in the new program. If the node has additional guard dependencies, add a condition for this guard and execute the guarded effects if the condition evaluated to true. Since each node on the schedule is only visited once, the new SCG can be constructed in linear time.

ABO, sequentialized according to the schedule depicted in Figure 5.2.19, *Example* can be seen in Figure 5.2.21. The sequentialization forms a list of statements which represents the tick function. The whole list will be executed in every tick. Effects are guarded by the corresponding guard. For example, at program start, O1 and O2 are set to true. Hence, `_g0`, which is equivalent to `_GO`, guards these statements at the beginning of the SCG. In the second column, B and O1 are set to true if `_g3` evaluated to true. `_g3` depends on the input A. Finally, if the synchronizer guard `_g10` becomes true, O1 is set to false and O2 is set to true. —

5.2.8 Optimization

When inspecting the straightforwardly constructed sequentialized version of ABO in Figure 5.2.21, one can see statements which seem to be redundant. This section introduces some optimizations that can be applied to reduce the amount of redundant statements. Depending on the level of traceability, these optimizations can be applied at various steps during the compilation process. To facilitate understandability and simplicity, many of the following optimizations are carried out on the sequentialized SCG. Nonetheless, it is also a valid goal to keep a program smallest at the earliest possible state. Therefore, common optimization techniques, such as the

5. Interactive Compilation for SCCharts

Sparse Conditional Constant Propagation (SCCP) [WZ91], demonstrated for SCCharts elsewhere [SSH18d], can be applied immediately after the SCG is constructed. Both optimization approaches should be made at model level, so that the developer can inspect the optimizations in a reasonable way using the possibilities presented in Chapter 4, as shown in the following.

Model-level optimizations fit well into the compiler infrastructure presented in Chapter 3. They can be easily added modularly as (post-)processors and switched on or off as required. Most of the optimizations presented in this section are only an approximate page full of code. Table 5.2.1 shows the KIELER optimizations and the LoC of the process function of the processors. They are categorized into *structural optimizations*, which target specific structures of a program and which are independent of the program semantics, and *semantic optimizations*, which also consider program semantics.

Note that while it is entirely possible to rely only on external optimizations, e. g. by using the GCC with a high optimization level, it is still conducive to perform the optimizations on model-level. Besides being able to actually see the transformations interactively, the compiler developer usually has more knowledge about the semantics than the external compiler. In case of the SCCharts netlist for example, the model-based optimizations exploit knowledge about the MoC and the cyclic execution behaviour, e. g. by moving instructions across functions and accessing values from previous ticks.

Superfluous Fork Remover

Besides providing means for concurrency, a superstate is also useful for structuring a model. It groups parts of the program which belongs together and allows preemption of the entire group. It is not required to fork off several threads. If a superstate has only one region, the fork node in its SCG has only one outgoing control-flow. As already mentioned in Section 5.2.2, this would not create a new BB and the fork and join nodes are semantically superfluous. To streamline the transformation, the SCCharts-to-SCG processor does not consider this special case separately. These superfluous nodes can be removed easily in a dedicated processor, which can be switched off when necessary, e. g. to retain model-preserving bidirectional transformations. Therefore, all incoming control-flows to the fork must be re-routed

5.2. Netlist-Based Compilation

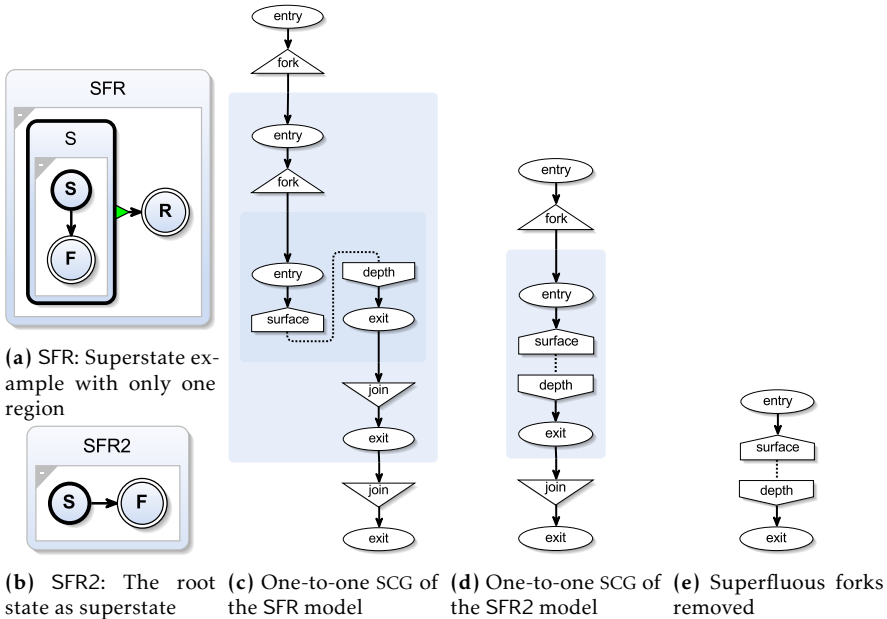


Figure 5.2.22. While structural reasonable, forks with just one outgoing control-flow are semantically superfluous for the downstream compilation.

to the node following the entry node of the enclosed thread. Symmetrically, the control-flows leading to the exit node of the thread must be connected to the node which succeeds the join node. The fork, entry, exit and join nodes are then removed.

Figure 5.2.22 shows the optimization. Figure 5.2.22a depicts a superstate, *Example* whose inner behaviour delays the execution by one tick. Its unoptimized SCG is shown in Figure 5.2.22c. Since the root state itself can also have multiple threads, it is also a superstate. Therefore, the unoptimized SCG shows two forks with each only having one outgoing control flow. The innermost thread holds the surface and the depths of the pause. This is further clarified in the second model in Figure 5.2.22b, which only contains a pause. The corresponding unoptimized SCG is depicted in Figure 5.2.22d. Only

5. Interactive Compilation for SCCharts

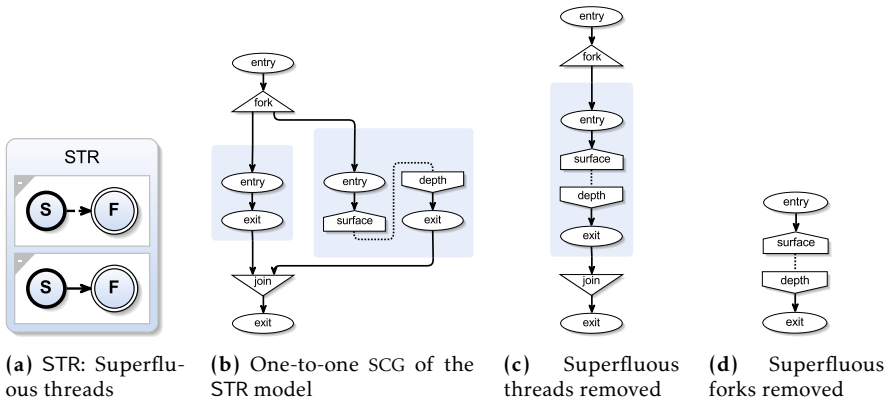


Figure 5.2.23. While semantically valid on model-level, threads with no behaviour are superfluous for the downstream compilation.

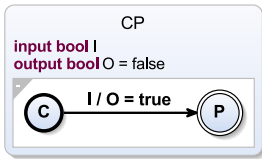
the superfluous fork of the root state is left. After applying the optimization as described above, both SCGs simply comprise the pause, as shown in Figure 5.2.22e. Both models are semantically identical.

Superfluous Thread Remover

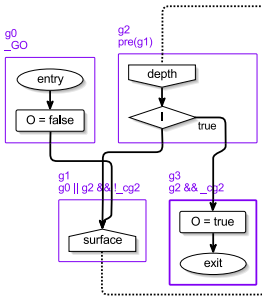
Similarly to the superfluous fork, while semantically valid on model-level, threads can be superfluous, too. Threads with no behaviour could be forbidden on model-level via validator rules. However, as a consequence, every high-level transformation must ensure that this kind of structure is avoided. Another possibility is to add a dedicated processor which ensures that a normalized SCG structure is present. If a thread contains no behaviour, i. e. its control-flow directly traverses from the entry to the exit node, the processor can remove the thread entirely from the fork/join construct.

Example Figure 5.2.23 shows an optimization example for removing superfluous threads. The STR model depicted in Figure 5.2.23a consists of two threads. The first one exists immediately, and therefore, contains no behaviour, while the second one includes a pause. The unoptimized SCG of STR is shown in Figure 5.2.23b. Figure 5.2.23c shows the optimized version of the SCG

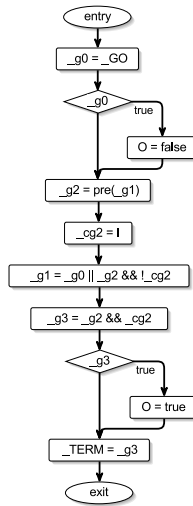
5.2. Netlist-Based Compilation



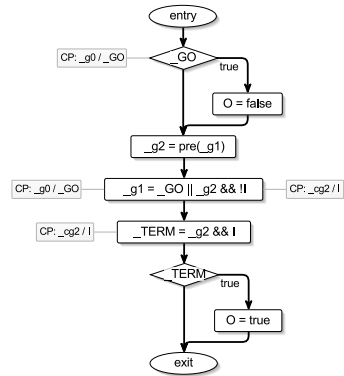
(a) CP: Copy propagation example that emits O as soon as I arrives



(b) BBs of the CP model



(c) Unoptimized sequentialized SCG of the CP model



(d) Optimized sequentialized SCG of the CP model with processor's annotations

Figure 5.2.24. Example result of the copy propagation processor which eliminates copy statements

The optimization created a fork node with only one outgoing control-flow. Therefore, it is reasonable to run the superfluous thread remover before the superfluous fork remover. STR is semantically equivalent to the SFR model presented in Figure 5.2.22. Hence, the same minimal SCG (Figure 5.2.23d) is created after the execution of both optimizations.

Copy Propagation

Statements of the form $f = e$ are called *copy statements* [ASU86]. By propagating e into expressions which use f , the whole assignment can be eliminated. The BB step of the netlist-based compilation is relatively generous when it comes to copy statements. In particular, context switches which introduce

5. Interactive Compilation for SCCharts

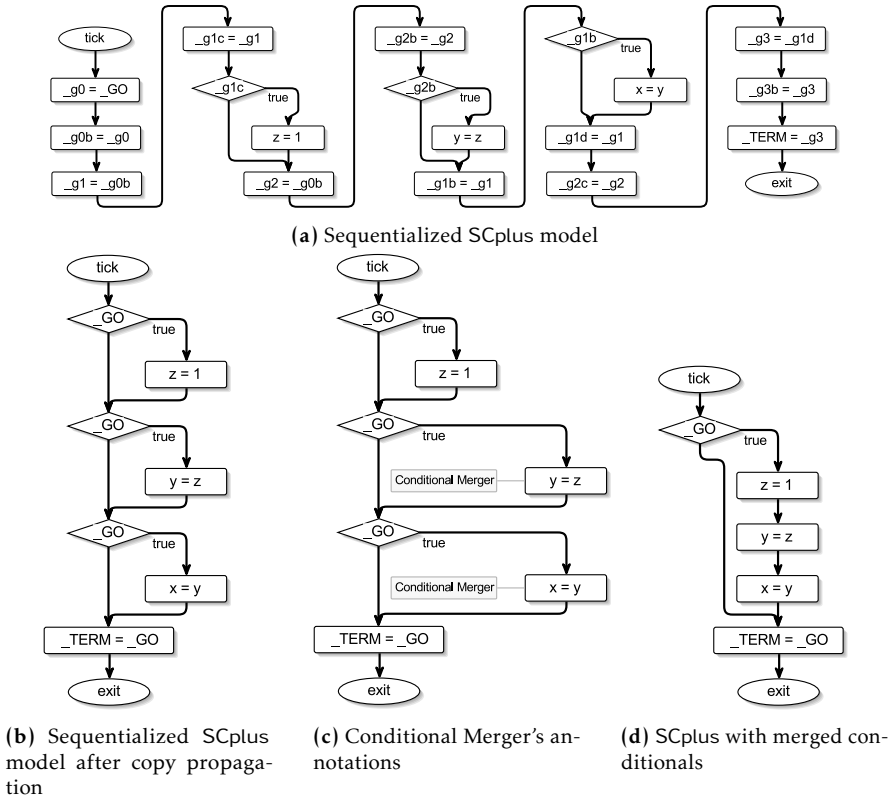


Figure 5.2.25. The Conditional Merger merges control-flows which are executed on the same condition.

multiple SBs use the same guard as their parent BB, which must not be evaluated again. While this generosity may help to trace program parts in the interactive view of the model-based compiler, the final code should not contain superfluous assignments. A copy propagation processor can propagate the expressions and eliminate these statements. Conditional and empty flags of that form may also be optimized if possible.

5.2. Netlist-Based Compilation

Figure 5.2.24 shows an example result of the copy propagation processor. *Example* The CP model waits for an input *I* to arrive, and then sets output *O* to true. The BBs with their guards can be seen in Figure 5.2.24b and the unoptimized sequentialized version of the SCG is depicted in Figure 5.2.24c. There are two simple copy statements, `_g0 = _GO` and `_cg2 = I`, in this version of the graph. The optimizer performs the copy propagation and creates a new version of the SCG as depicted in Figure 5.2.24d. —

The processor adds annotations to the statements which have been altered, so that the developer can track what has happened. Note that `KiCo` allows for different propagation strategies and the degree of the propagation can also be influenced via the `KiCo` environment. By default, only simple copies are replaced and operations, such as `pre` which reads a value from the previous tick, are conservatively ignored. The same is true for expressions which comprise hostcode calls, as discussed in Section 5.2.2. Therefore, if `_cg2` had included a hostcode call, it would not have been propagated.

Conditional Merging

As mentioned in Section 2.3.7, control flow elements in netlist-based compilation approaches which are executed on the same condition can be merged together, which is demonstrated in Figure 5.2.25.

Figure 5.2.25a shows the sequentialized SCG of the SCplus example from Figure 5.2.20. Due to the bi-directional inter-thread communication several SBs are created, which guard the assignments to *x*, *y*, and *z*. Basically all guards can be removed by the copy propagation, which generates an SCG as depicted in Figure 5.2.25b. A dedicated conditional merger processor then identifies assignments which are executed on the same condition. The processor can annotate the model as shown in Figure 5.2.25c to illustrate its behaviour. The detected assignments are then merged as depicted in Figure 5.2.25d. As shown in Section 6.2, this merging can also be made by reordering if the conditional nodes are subsequent to each other and as long as variable dependencies are respected. —

5. Interactive Compilation for SCCharts

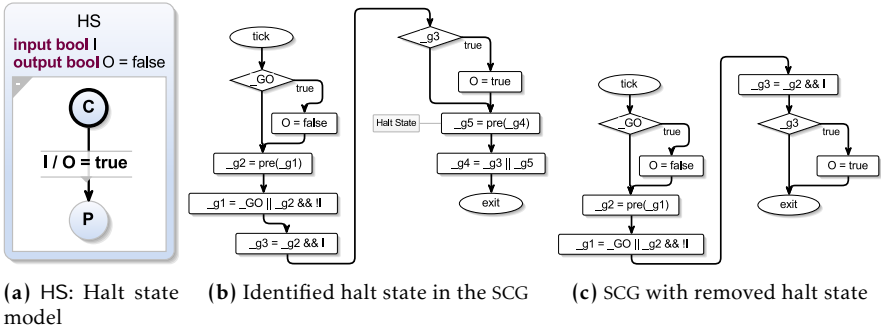


Figure 5.2.26. The halt state optimization removes halt states, which are ineffective.

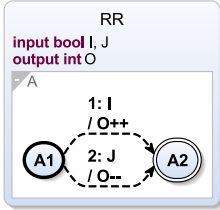
Halt State Remover

Consider the copy propagation example from Figure 5.2.24 but with a halt state instead of the final state, depicted in Figure 5.2.26a. In the copy propagation example, the program terminates as soon as O was emitted. In Figure 5.2.26, the program stays alive after the emission of O . The active states, which are accessed via the pre operator, are stored in registers. However, the program is essentially ineffective after reaching P . Since the final BB of the program is not reachable, there is no `_TERM` flag.

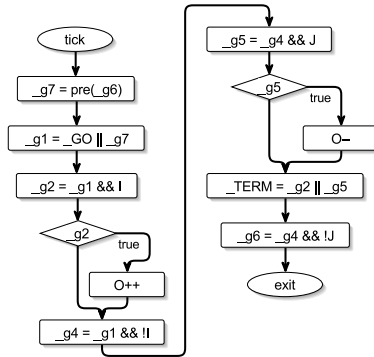
Halt states manifest as two assignments in the netlist which are not referenced further. They calculate the guards of the surface and the depth of that particular state. The first assignment has the form $_gX = \text{pre}(_gY)$, which $_gX$ being the guard of the depth which belongs to the surface of the block guarded by $_gY$. The second assignment has the form $_gY = _gZ \parallel _gX$, which means that the block is active, if was entered in this tick ($_gZ$) or active before ($_gX$). $_gZ$ is an arbitrary predecessor.

If the dedicated processor detects a halt state, it is annotated as before, shown in Figure 5.2.26b. Both assignments are then removed as there is no observable effect from the outside. The resulting SCG is depicted in Figure 5.2.26c.

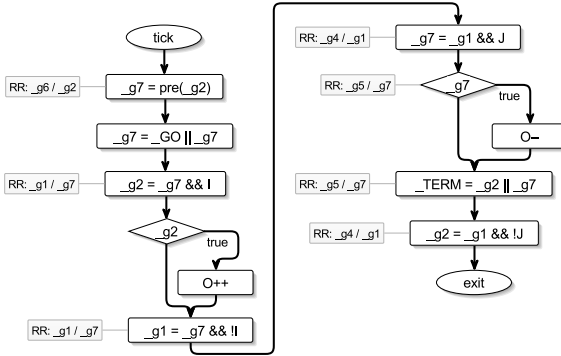
5.2. Netlist-Based Compilation



(a) RR: Register relocation example



(b) Each guard is resembled by a variable



(c) As guards are only used in succeeding guard expressions, they can be recycled after they have been used.

Figure 5.2.27. Register relocation: Used guard variables can be recycled after their life span.

Register Relocation

While *register allocation* is a topic of its own, e.g. see elsewhere [CAC+81; ASU07], the topological sorting of the guards usually only assigns a guard immediately before it is required. Furthermore, it is only assigned once and used in succeeding guard expressions. Therefore, in software code syntheses,

5. Interactive Compilation for SCCharts

the variable used to resemble the guard can be recycled in later assignments after it has been *consumed*, which drastically reduces the set of variables required to form the netlist. This is especially helpful in systems with scarce resources, such as the TinyVM environment used in the practical Mindstorm examples explained later in Section 7.3.1. However, this optimization should not be activated in hardware synthesis because the nature of the netlist already resembles unique *wires*. The variable recycling would require a hardware processor the separate merged variables again, e. g. via SSA, to gain uniquely defined values per tick.

Example Figure 5.2.27 shows an example of guard variable recycling. The OSM, depicted in Figure 5.2.27a, immediately waits for two inputs I and J. Depending on the input, the output O is either incremented or decremented. The corresponding sequentialized SCG after application of the copy propagation is shown in Figure 5.2.27b. `_g7` is assigned in the first assignment and then reused in the second. Afterwards, it is free to be used again and to replace another guard so that its space can be saved. `_g7` replaces `_g1` in the conditional BBs for I. Subsequently, it is free again and can be used to substitute `_g5` in the second conditional. Technically, `_g7` could also replace `_g6`. However, to facilitate other optimizations, the implemented register relocation does not replace guard variables with other guard variables that have been used in assigning pre operations. Hence, `_g6` is replaced by the now free `_g2` instead of `_g7`, because `pre(_g6)` is assigned to `_g7` in the first assignment. Overall, three of the initially six guard variables could be saved in this example.

Persistent State Optimizer

A common pattern, often found in models which make use of immediate during actions and therefore also in the SCCharts dataflow extension, discussed in Section 6.2, is the *persistent state* pattern. The pattern is similar to the halt state pattern from Section 5.2.8 but is initialized by the `_GO` flag and, contrary to halt states, is permitted to be used in other guard expressions. This resembles an action which is performed on every clock tick. Therefore, the computation of the guard evaluates to true in every tick.

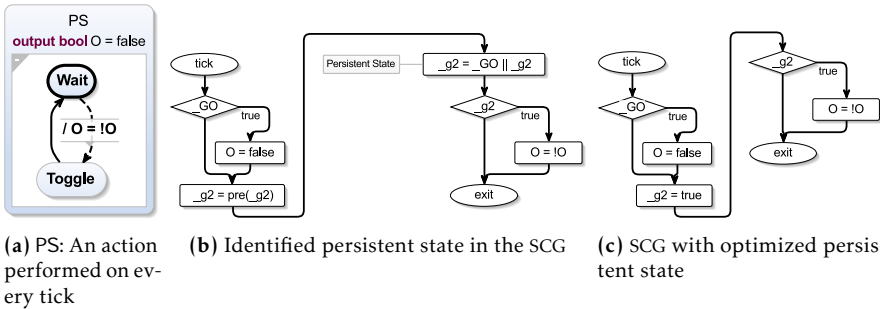


Figure 5.2.28. Persistent state optimization, replaces stateful register patterns with boolean true expressions.

Figure 5.2.28 shows a persistent state example. The model depicted in *Example* Figure 5.2.28a toggles a boolean output O in every tick, which is semantically identical to an immediate during action toggling O . The optimizer recognizes the persistent state in the SCG in Figure 5.2.28b. The persistent state guarded by $_g2$ manifests in the netlist as $_g2 = \text{pre}(_g2)$ and $_g2 = _GO \parallel _g2$. $_g2$ is afterwards referenced in the conditional to toggle O . The assignments are then replaced by an action assigning true to $_g2$ in Figure 5.2.28c.

Clearly, this can be optimized further, which is taken care of by the next optimization.

Partial Assignment Evaluation

Expressions which can be evaluated statically may simplify the SCG in terms of required assignments and concerning the also structure of the graph. The *partial assignment evaluation* optimizer traverses the sequential SCG and gathers variable configurations. These are then applied to the expressions when visiting the nodes. After the execution of the persistent state optimizer, the true literal is assigned to $_g2$ in Figure 5.2.28c. Therefore, the optimizer may insert true in the condition of the second conditional node directly as depicted in Figure 5.2.29a. Statically evaluated conditionals can be resolved immediately. Therefore, the conditional is removed and complete true branch is moved to the main path of the sequentialized SCG. If the condition

5. Interactive Compilation for SCCharts

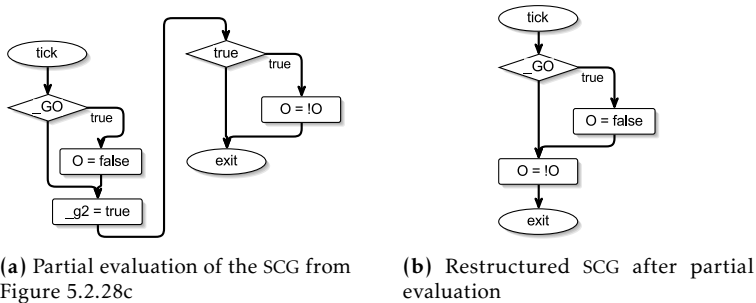


Figure 5.2.29. Partial Assignment Evaluation scans the SCG sequentially and applies configurations to variables.

were false, the true branch would have been discarded. Since `_g2` is also no longer needed, it is also removed. The final result of the optimization is shown in Figure 5.2.29b.

_GO MoC Optimization

Some optimizations can take advantage of the underlying MoC. Since the MoC used in SCCharts dictates a periodic execution with a mandatory reset at the beginning or restart of the program, the code generation can move initialization parts from the logic into the reset function.

Example Consider the example model in Figure 5.2.30a, which initializes `O` with false. After waiting for one tick, `l` is checked in every tick and once present, `O` is set to true until entering a halt state. Due to the nature of the initialization feature in extended SCCharts, the initialization of `O` is transformed into core constructs. The intermediate core SCChart is depicted in Figure 5.2.30b. The resulting sequentialized SCG is shown in Figure 5.2.30c. The first conditional with the `_GO` as condition takes care of the initialization. However, the condition is checked in every tick, even though it is only executed once during its lifetime. An optimizer can move these parts into the dedicated reset function and can discard the conditional check. The result is shown in Figure 5.2.30d.

The SCG now shows two sequentialized SCGs, one for the reset and one

5.2. Netlist-Based Compilation

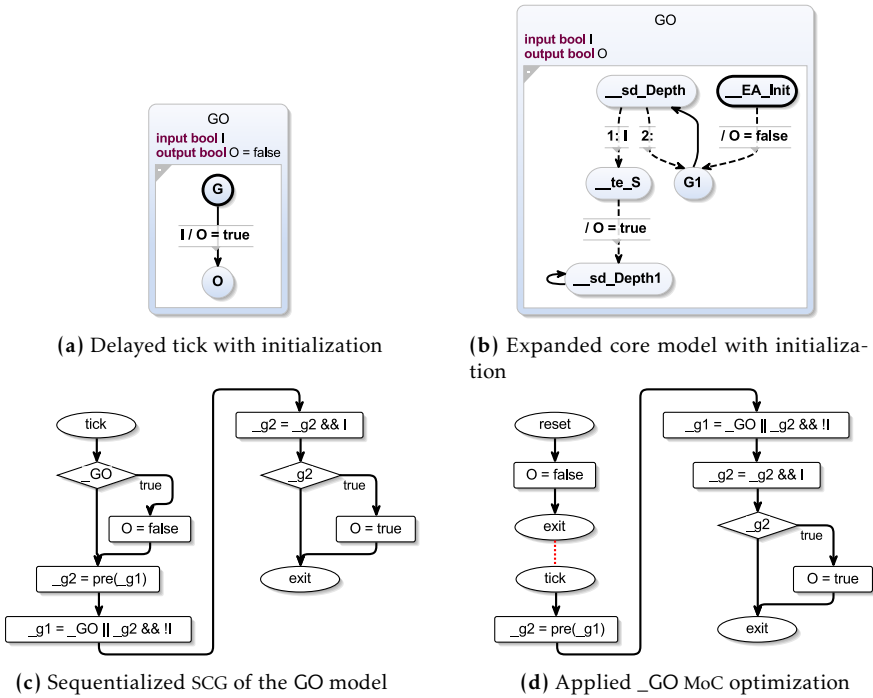


Figure 5.2.30. `_GO` MoC optimization example

for the logic function. If the optimizer runs after the conditional merger, the optimization must only check for one `_GO` conditional. If one is present, the nested assignments can be moved directly. The same is true for models which set their values in the first tick.

Figure 5.2.31a initializes `O` and `O2` to false but then sets them to true in every tick. While the core model expands to a comparably big model (see Figure 5.2.31c), the optimized code is fairly simple. The final sequentialized SCG is depicted in Figure 5.2.31b. The initializations are moved to the reset function, whereas `O` and `O2` are set to true in every tick in the logic of the program. No further guards are required. *Example*

5. Interactive Compilation for SCCharts

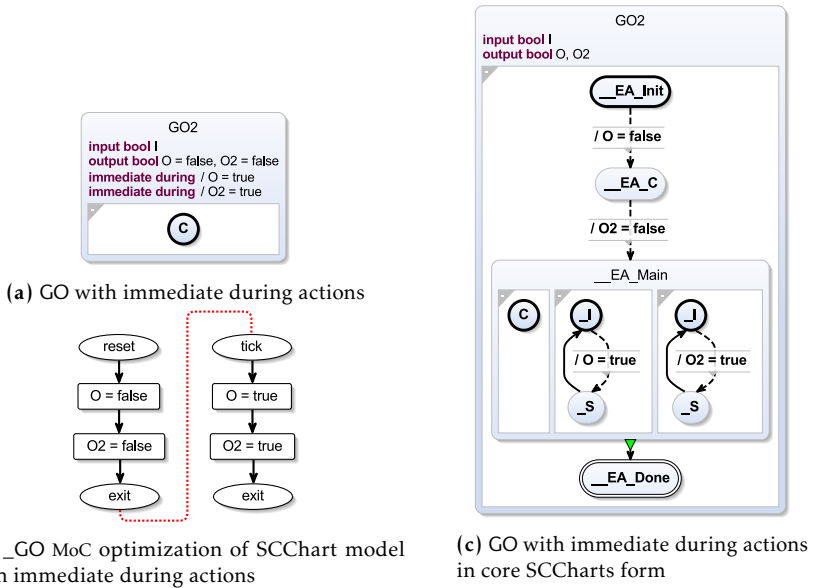


Figure 5.2.31. Second `_GO` MoC optimization example

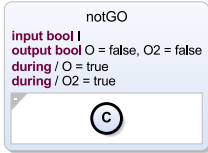
!_GO MoC Optimization

With respect to the `_GO` MoC optimization, things become more difficult when not handling an immediate case. Such effects are guarded, because they must not be executed in the first tick but in subsequent ticks. Therefore, their guard will be set to true in the netlist after their condition was checked.

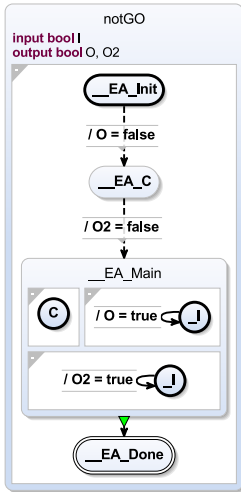
Example

Consider the same model as in Figure 5.2.31 but with delayed during actions, depicted in Figure 5.2.32a. Instead of the two state cycle per during action, the core version now just has a single state with a delayed self loop per during action, shown in Figure 5.2.32b. As the effects do not take place immediately, they have to be guarded in the sequentialized SCG, which is shown in Figure 5.2.32b. However, as the guards solely depend on completion of the first tick, they can be set to true, after the corresponding conditional. This behaviour, dictated by the MoC, can be expressed as `!_GO`, as shown in Figure 5.2.32c.

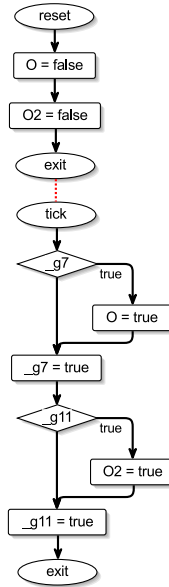
5.2. Netlist-Based Compilation



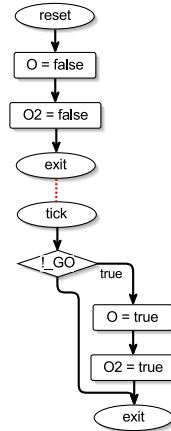
(a) notGO with delayed during actions



(b) notGO with immediate during actions in Core SCCharts



(c) notGO as sequentialized SCG



(d) !_GO MoC optimization of SCChart model with immediate during actions

Figure 5.2.32. !_GO MoC optimization example

5. Interactive Compilation for SCCharts

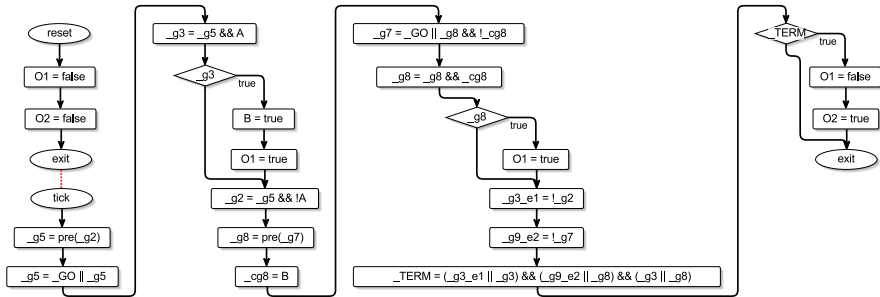


Figure 5.2.33. Sequentialized optimized SCG of ABO

Note that both `_GO` optimizations only save a few conditional checks and assignments which particularly influence the first tick of a program. However, as models can be referenced by other models, these optimizations also may have effects in later ticks if the referenced model is (re-)entered at later tick times. Details about referenced SCCharts follow in Section D.1.

Example—The Optimization of ABO

After discussing the optimizations at small examples, Figure 5.2.33 shows the optimized version of ABO. Instead of 28 nodes present in the logic function in the unoptimized sequentialized SCG in Figure 5.2.21 on page 158, only 19 are needed after applying the optimizations. The interactive model-based compilation approach makes it possible for the modeller to inspect every step in a representation they understand and even alter the outcome if a particular circumstance requires it.

5.2.9 Code Generation

Once the sequentialized SCG has been built, the model can be serialized to code. KiCo directly generates the code for the first two layers, *logic* and *tick*, as discussed in Section 3.2.1 on page 65. Therefore, generally three functions are created. A data structure holds all data of a particular instance of the model. The *logic function* comprises the serialized code of the sequentialized

5.2. Netlist-Based Compilation

SCG without parts in a potential reset SCG. Given an instance of a model including inputs from the environment, the function exactly calculates one tick. The logic function is called by a *tick function*, which additionally handles register saves and the `_GO` signal. Finally, the initialization of the registers, `_GO`, and `_TERM` is handled by a *reset function*. It also comprises the serialized code from a potential reset SCG.

C Code Generation

For C, this can be done straightforwardly: The definitions are stored in the header file; the implementation is located in the source file. As with for every compilation step in the interactive model-based approach, the generated files can be inspected interactively and opened directly in the editor.

The serialized C code of ABO can be seen in Figure 5.2.34 as it appears *Example* interactively in the KiCo compilation chain. The data necessary for the model instance is stored in the TickData struct in the header file on the left. It includes the variables from the interface, namely A, B, O1, and O2, and `_GO` and `_TERM`, which signal the actual state of the instance. The registers, which hold values of guards from the last tick, are stored in the variables prefixed with `_pg`. `_pg2` and `_pg7` for ABO. Further, all temporary guards, prefixed with `_g`, which are evaluated in each tick, are also included in the struct. Hence, the variables of the guards do not have to be declared in the logic function, which is called in every tick. Furthermore, the actual state, including all guard evaluations, can be sent to a simulator for visualization, e. g. highlighting corresponding hot wires during a simulation. It would be also possible to change the state of the model instance here. However, the temporary guards can be discarded if a more compact TickData struct is preferred.

In the C file, one can inspect the three functions. As mentioned before, the logic function contains the serialized assignments and conditionals of the sequentialized SCG, which can be seen in Figure 5.2.33. It is working on a given TickData instance. Subsequently, the reset function resets `_GO`, `_TERM`, and the registers `_pg2` and `_pg7`. The initialization of ABO's outputs O1 and O2 was also moved into the reset function by the `_GO` MoC optimization, discussed in Section 5.2.8. Eventually, the tick function calls logic with

5. Interactive Compilation for SCCharts

```
CODE - ABO.h  
\[Open in Editor\]  
/*  
 * Automatically generated C code by  
 * KIELER SCCharts - The Key to Efficient Modeling  
 *  
 * http://rtsys.informatik.uni-kiel.de/kieler  
 */  
  
typedef struct {  
  char A;  
  char B;  
  char O1;  
  char O2;  
  char _g2;  
  char _g3;  
  char _g5;  
  char _g7;  
  char _g8;  
  char _g0;  
  char _cg2;  
  char _cg8;  
  char _g3_e1;  
  char _g9_e2;  
  char _TERM;  
  char _pg2;  
  char _pg7;  
} TickData;  
  
void reset(TickData* d);  
void logic(TickData* d);  
void tick(TickData* d);
```

```
CODE - ABO.c  
\[Open in Editor\]  
/*  
 * Automatically generated C code by  
 * KIELER SCCharts - The Key to Efficient Modeling  
 *  
 * http://rtsys.informatik.uni-kiel.de/kieler  
 */  
  
#include "ABO.h"  
  
void reset(TickData* d) {  
  d->_GO = 1;  
  d->_TERM = 0;  
  d->_pg2 = 0;  
  d->_pg7 = 0;  
}  
  
void logic(TickData* d) {  
  if (d->_GO) {  
    d->O1 = 0;  
    d->O2 = 0;  
  }  
  d->_g5 = d->_pg2;  
  d->_g5 = d->_GO || d->_g5;  
  d->_g3 = d->_g5 && d->A;  
  if (d->_g3) {  
    d->B = 1;  
    d->O1 = 1;  
  }  
  d->_g2 = d->_g5 && !d->A;  
  d->_g8 = d->_pg7;  
  d->_cg8 = d->B;  
  d->_g7 = d->_GO || d->_g8 && !d->_cg8;  
  d->_g8 = d->_g8 && d->_cg8;  
  if (d->_g8) {  
    d->O1 = 1;  
  }  
  d->_g3_e1 = !d->_g2;  
  d->_g9_e2 = !d->_g7;  
  d->_TERM = (d->_g3_e1 || d->_g3) && (d->_g9_e2 || d->_g8) && (d->_g3 || d->_g8);  
  if (d->_TERM) {  
    d->O1 = 0;  
    d->O2 = 1;  
  }  
}  
  
void tick(TickData* d) {  
  logic(d);  
  
  d->_pg2 = d->_g2;  
  d->_pg7 = d->_g7;  
  d->_GO = 0;  
}
```

Figure 5.2.34. Serialized C code of ABO in the netlist-based approach as it interactively appears in the KiCo compilation chain

5.2. Netlist-Based Compilation

```
1 _tick:
2 LFB2:
3   .cfi_startproc
4   pushl %ebp
5   .cfi_def_cfa_offset 8
6   .cfi_offset 5, -8
7   movl %esp,%ebp
8   .cfi_def_cfa_register 5
9   subl $4,%esp
10  movl 8(%ebp),%eax
11  movl %eax,(%esp)
12  call _logic
13  movl 8(%ebp),%eax
14  movzbl 4(%eax),%edx
15  movl 8(%ebp),%eax
16  movb %dl,15(%eax)
17  movl 8(%ebp),%eax
18  movzbl 7(%eax),%edx
19  movl 8(%ebp),%eax
20  movb %dl,16(%eax)
21  movl 8(%ebp),%eax
22  movb $0,9(%eax)
23  nop
24  leave
25  .cfi_restore 5
26  .cfi_def_cfa 4, 4
27  ret
28  .cfi_endproc
```

(a) Generated assembler code of the ABO tick function with O0 optimization setting

```
1 _tick:
2 LFB2:
3   .cfi_startproc
4   pushl %ebx
5   .cfi_def_cfa_offset 8
6   .cfi_offset 3, -8
7   subl $4,%esp
8   .cfi_def_cfa_offset 12
9   movl 12(%esp),%ebx
10  movl %ebx,(%esp)
11  call _logic
12  movzbl 4(%ebx),%eax
13  movb $0,9(%ebx)
14  movb %al,15(%ebx)
15  movzbl 7(%ebx),%eax
16  movb %al,16(%ebx)
17  addl $4,%esp
18  .cfi_def_cfa_offset 8
19  popl %ebx
20  .cfi_restore 3
21  .cfi_def_cfa_offset 4
22  ret
23  .cfi_endproc
```

(b) Generated assembler code of the ABO tick function with O2 optimization setting

Listing 5.2.5. Assembler code of ABO with different compiler optimizations

the actual instance. Afterwards, the states guards that have corresponding registers are saved. Here, `_g2` and `_g7` are stored in `_pg2` and `_pg7`. `_GO` is set to false.

Note that storing every variable in a dedicated struct instead of using local variables requires two memory accesses per accessed variable. However, modern compilers optimize these accesses so that storing the temporary guards within the `TickData` struct and using them, for e. g. simulations, does not come at much greater cost w.r.t. memory efficiency. Listing 5.2.5 shows the assembler code for ABO's tick function, generated by the measurement system introduced in Section 3.3.7 on page 90 ff. In the unoptimized version on the left side in Listing 5.2.5a, the access to the struct via `movl 8(%ebp)`, can

5. Interactive Compilation for SCCharts

CODE - ABO.java	
<pre>/* * Automatically generated Java code by * KIELER SCCharts - The Key to Efficient Modeling * * http://rtsys.informatik.uni-kiel.de/kieler */ public class ABO { public boolean A; public boolean B; public boolean O1; public boolean O2; public boolean _g2; public boolean _g3; public boolean _g5; public boolean _g7; public boolean _g8; public boolean _GO; public boolean _cg2; public boolean _cg8; public boolean _g3_e1; public boolean _g9_e2; public boolean _TERM; public boolean _pg2; public boolean _pg7; public void reset() { _GO = true; _TERM = false; O1 = false; O2 = false; _pg2 = false; _pg7 = false; } }</pre>	<pre>public void logic() { _g5 = _pg2; _g5 = _GO _g5; _g3 = _g5 && A; if (_g3) { B = true; O1 = true; } _g2 = _g5 && !A; _g8 = _pg7; _cg8 = B; _g7 = _GO _g8 && !_cg8; _g8 = _g8 && _cg8; if (_g8) { O1 = true; } _g3_e1 = !_g2; _g9_e2 = !_g7; _TERM = (_g3_e1 _g3) && (_g9_e2 _g8) && (_g3 _g8); if (_TERM) { O1 = false; O2 = true; } } public void tick() { logic(); _pg2 = _g2; _pg7 = _g7; _GO = false; } }</pre>

Figure 5.2.35. Serialized Java code of the sequentialized SCG of ABO (column break edited manually)

be seen several times. As mentioned before, the function manages the register saves and the assignment of 0 to `_GO`. This is done strictly sequentially in the lines 13–22 after the call of `logic`. The GCC resolves the multiple accesses when using a stronger optimization, which can be seen in Listing 5.2.5b. The same effects as before are made in lines 12–16 without the need to dereference the struct at each access. Also, the assignments are reordered by the optimization. E.g., the assignment to `_GO` is situated in line 13.

Java Code Generation

Since Java is an object-oriented language, the code generation can make use of classes and objects to manage different instances of the model. It is not necessary to create a different data structure for `TickData`, although this would be possible in a dedicated class if preferred. Figure 5.2.35 shows the

5.2. Netlist-Based Compilation

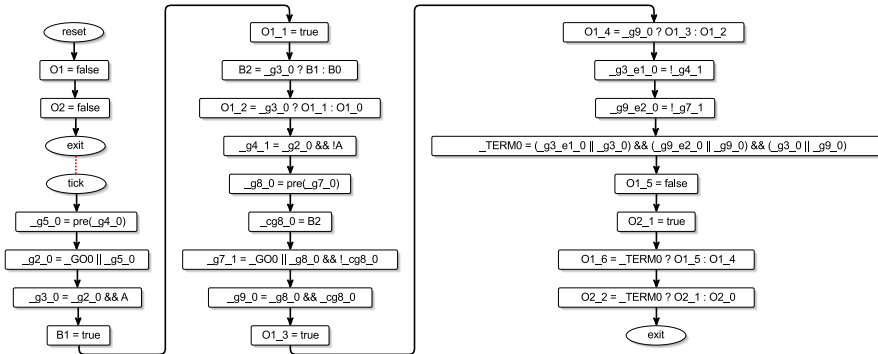


Figure 5.2.36. Sequentialized optimized SSA SCG of ABO

serialized code for ABO in Java. While the serialized elements are the same, they can be included within one class. Objects of this class can then be used to instantiate different versions of the same model. The contents of the logic, tick, and reset functions from before are identically stored in corresponding methods of the class. This kind of serialization is of course also possible for other object-oriented languages.

Circuit Generation

As discussed in Section 3.3, the target of a compilation does not have to be a textual program which can be processed further by another compiler. For example, the netlist-based compilation can be used to directly generate a circuit comprised of logic gates. Physical wires have a defined values for every tick. To guarantee this, the sequentialized SCG must be transformed into a form where sequential assignments to the same variable each become their own incarnation, also known as SSA. Instead of serializing code directly, the sequentialized SCG can be transformed into a sequentialized SSA SCG by a dedicated processor [Sch16][RSM+16]. While the guards are already only written once, the transformation is required so that the remaining variable accesses are also in SSA form.

5. Interactive Compilation for SCCharts

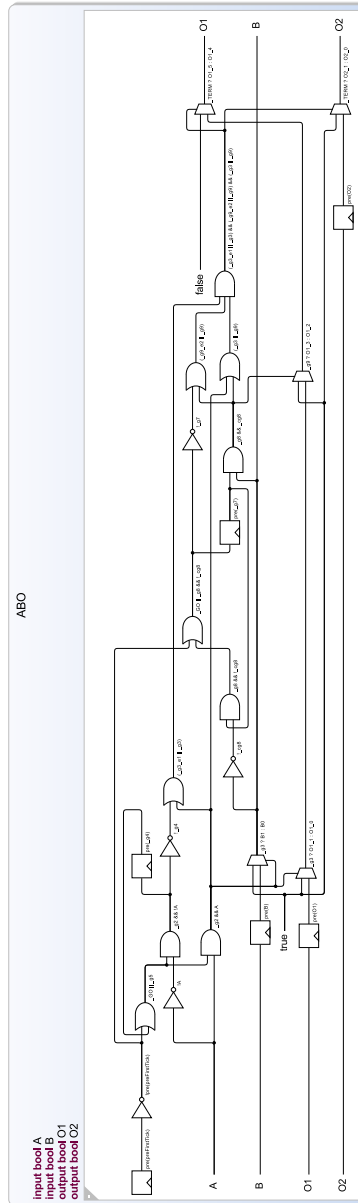


Figure 5.2.37. Synthesized circuit of ABO

5.2. Netlist-Based Compilation

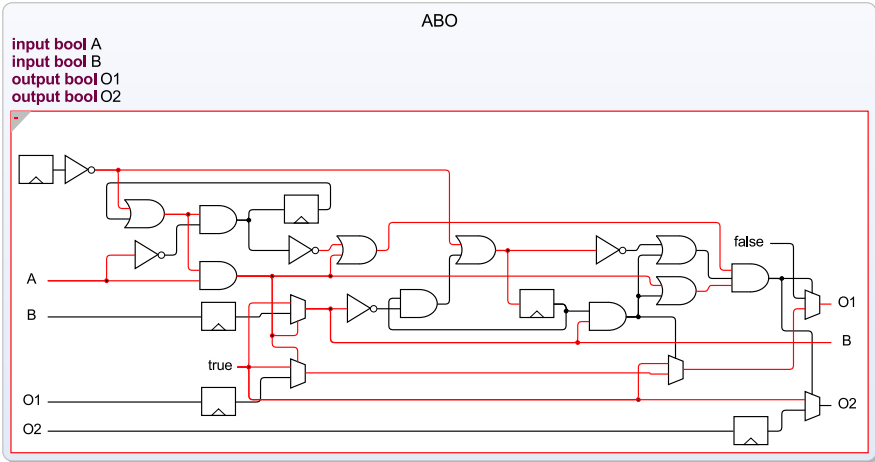


Figure 5.2.38. Synthesized circuit of ABO during simulation. The colour of a wire indicates its voltage (red=high, black=low).

The resulting SCG for ABO is depicted in Figure 5.2.36. For example, the state of B depends on the input of B and A. However, a wire of B cannot be high and low at the same time. Hence, the transformation creates different instances of B suffixed with an integer. The final version of B, B2 is given back to the environment. It is set to either B1 or B0 depending on the value of the guard `_g3_0`, the status of which is controlled by A. By translating each assignment into its logical components, the sequentialized SSA SCG can be synthesized into a circuit. The translation of all elements of the SKP was depicted in Figure 5.1.2 on page 121 ff. For example, a condition is translated into a multiplexer element. The whole circuit for ABO, with wire names enabled to ease the matching to the SCG nodes, can be seen in Figure 5.2.37. *Example*

As the other model views shown in this thesis, this circuit diagram is synthesized automatically with KLightD and ELK, showing the versatility of automatic view synthesis.

As meta-model for the circuit serves the dataflow extension from SCCharts, which is introduced in Section 6.2. The meta-model, the synthesis, and potentially other related processors, which can work on this meta-model,

5. Interactive Compilation for SCCharts

already exist. Therefore, this path of compilation is a further example of the modularity of interactive model-based compilation. From here it is simple to create another processor that transforms the model into VHDL, as has been done by Boysen et al. [BSH20a].

The circuit synthesis also demonstrates the usefulness of the temporary guard variables in the TickData struct. As mentioned before, the tick data struct holds the complete status of a model instance. While communicating with the simulator integrated into KIELER, the complete status is exchanged. Hence, the value of every wire is known and can be visualized.

Example Figure 5.2.38 shows a running instance of the ABO model in its circuit representation. The figure shows ABO after the first tick with A present. While the output B becomes hi (resp. true), the first two instances of B are low (resp. false) as discussed before.

5.3 Priority-Based Compilation

The netlist-based compilation approach presented in the previous section can be used to either generate code for software and hardware. The reaction time of the system is roughly proportional to the size of the model, because the netlist is calculated in each tick. The priority-based compilation approach generates code for software only but solely depends on the components that are active within a tick. Hence, the approach can scale better in very large models. Since hardware is not a concern here, the approach also accepts instantaneous control-flow cycles and solves schizophrenia naturally. Nonetheless, as the netlist-based approach it rejects programs with immediate dependency cycles since no static schedule of priorities can be found.

Remark There exist approaches which reduce the overall amount of guard calculations and only execute parts of a netlist. KIELER also contains an experimental processor that excludes inactive threads from the calculation. A more general approach which produces modular sequential imperative code from synchronous data-flow networks by partitioning it into a minimal number of atomic classes has been studied further by Pouzet and Raymond [PR09].

5.3. Priority-Based Compilation

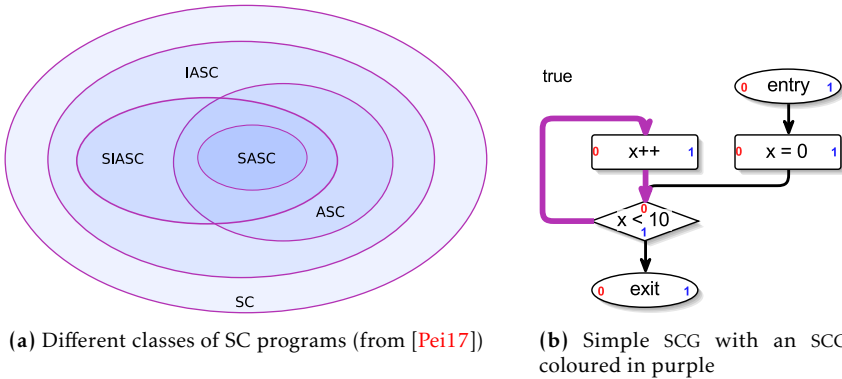


Figure 5.3.1. SC Classes and programs

Figure 5.3.1a gives an overview over different classes of Sequentially Constructive (SC) programs. The most restrictive class, Structurally Acyclic-Schedulable Constructive (SASC), only allows programs in which the SCG does not contain any immediate cycles. Structurally IUR-Acyclic-Schedulable Constructive (SIASC) programs allow immediate cycles in the control-flow of the program as long as the cycle does not contain any data dependencies. Analogously, Acyclic-Schedulable Constructive (ASC) and IUR-Acyclic-Schedulable Constructive (IASC) employ the same rules but for run-time scheduling. There are also SC-constructive programs, which comprise immediate cycles with data dependencies. These programs are situated in the outer SC class. While they are schedulable, they usually tend to be difficult to be handled in a generic way by compilation approaches.

The priority-based compilation approach calculates a static schedule during compilation time. It is capable to handle programs from the SIASC class. Figure 5.3.1b shows a simple SCG with a Strongly Connected Component (SCC) coloured in purple. An SCC is a directed sub-graph in which every node can reach every other node. It corresponds to an immediate cycle in the SCG, which can be handled by the priority-based approach. The red and blue integers in the nodes of the SCG are the assigned priorities, according to the assignment algorithm discussed in the next section. In this case, however,

5. Interactive Compilation for SCCharts

all nodes comprise the same priority since there is no concurrency involved.

To apply a static schedule, the priority-based approach extends SCG nodes with *priorities*. The SCL language defined in Section 5.1 on page 118 is extended by a priority instruction:

$$s ::= x = e \mid s_1 ; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid l : s \mid \text{goto } l \mid \\ \text{fork}_N s_1 \text{ par } \dots \text{ par } s_N \text{ join}_N \mid \text{pause} \mid \text{prio}(id)$$

where x is a *variable*, e an *expression*, l is a *program label*, and id is an integer representing a prioID. As described later, prioIDs are actually computed from a priority, grounded by dependencies, and thread segment IDs. The prioIDs are used at run-time to schedule concurrent threads. Control is not realized with guards for each basic block, as was the case in the netlist-based approach, but instead with continuation points for each thread. This can be achieved with *computed gotos* in the GCC for example. If computed gotos are not available, as e.g. in ANSI C or Java, this can be emulated with a switch-case logic. For C, the whole target language, named SCL_p to avoid confusion, can be realized using C macros [Han09b]. Both variants, C with computed gotos and switch-case logic in Java, are explained in Section 5.3.2. The calculation of the prioIDs is discussed in the following section.

5.3.1 Priorities

Each program consists of a set of *thread segments*, which are delineated by fork/join nodes in the SCG and have a *thread segment ID* ($tsID$). Furthermore, each SCG node has a *priority*. SCL_p dispatches threads based on a *prioID*, which combines the node priority with the $tsID$ in a lexicographic fashion. For an SCL_p program with $tsIDs$ in $N_{<n}$ for some n , prioIDs are computed as $priority \times n + tsID$. This induces an ordering dominated by priorities. Identical priorities are resolved by $tsIDs$.

Implementation

Encoding both $tsIDs$ and priorities into a single scalar permits efficient thread book keeping. For example, in the implementation of the SCL_p operators, the prioID indexes an array which stores for each thread its continuation, and also indexes bit vectors which keep track of which threads are currently active. Dynamic priority changes of a thread allow instantaneous back-and-forth communication between concurrent threads. The encoding

5.3. Priority-Based Compilation

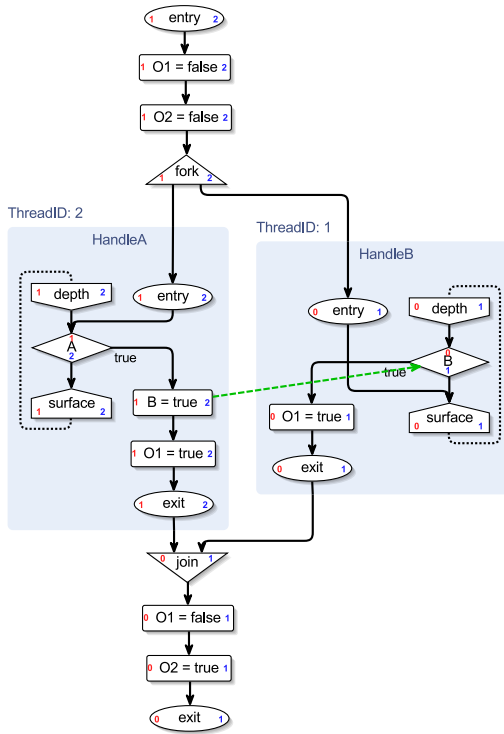


Figure 5.3.2. The SCG of ABO annotated with priorities (red), thread segment IDs and prioIDs (blue)

permits a fast dispatching based solely on which thread has the highest prioID. However, if the number of required priorities exceeds the width of the available bit vector, the bookkeeping must be switched to other data structures, such as arrays.

The priorities and tsIDs must be assigned so that the resulting prioIDs induce a scheduling order which respects the scheduling constraints induced by the SCMoC. This is, for example, achieved with the priority assignment algorithm [HMA+14], which runs in linear time and requires that the program is IUR-acyclic. There is also a `_TickEnd` thread, with priority and tsID

5. Interactive Compilation for SCCharts

```
Algorithm longestWeightedPath(currentSCC) {
  dependencies ← currentSCC.outgoingDependencies
  neighbors ← currentSCC.successors
  prio ← 0
  foreach (d in dependencies) {
    if (!d.scc.visited) {
      d.scc.visited ← true
      longestWeightedPath(d.scc)
    }
  }
  foreach (n in neighbors) {
    if (!n.scc.visited) {
      n.scc.visited ← true
      longestWeightedPath(n.scc)
    }
    prio ← max(n.scc.npr, prio)
  }
  currentSCC.npr ← prio
}
```

Listing 5.3.1. Longest weighted path algorithm (from [Pei17])

both being 0, which is always running and manages the return from the tick function.

The algorithm, shown in Listing 5.3.1, uses a modified version of Tarjan’s Algorithm [Tar72], which recognizes tick edges in the SCG, to find all SCCs. Then, the longest weighted path, with weighted edges 0 for control-flow edges and 1 for dataflow edges, is calculated with a DFS. Fundamentally, a node with an outgoing dependency edge must have a higher node priority than the target of the dependency. The chosen priority is then propagated to the node’s predecessors.

Example In ABO, the concurrent access to shared variable B induces a dependency, which can be handled by assigning priority 1 to the nodes in HandleA and in the initial thread segment of the root thread (up to the fork) and priority 0 to the other nodes. The *root thread* corresponds to the SCChart’s root state and gets started when the program gets started; when the root thread terminates, the whole program terminates.

In ABO there are no concurrent threads of the same priority, therefore the assignment of tsIDs does not influence any scheduling decisions; thus tsID 2 is arbitrarily assigned to HandleA and tsID 1 to HandleB. However,

5.3. Priority-Based Compilation

unnecessary jumps and run-time prioID changes can be avoided by 1) ordering HandleA before HandleB in the fork/join, so that the thread with lowest prioID comes last and gets to execute the join, 2) propagating HandleA's tsID 2 up to the initial segment of the root thread, and 3) propagating HandleB's tsID 1 down to the final segment of the root thread (from the join onwards). Thus, there are $n = 3$ tsIDs in use, and the resulting prioIDs are $1 \times 3 + 2 = 5$ (HandleB/root), $0 \times 3 + 1 = 1$ (HandleA/root), and $0 \times 0 + 0 = 0$ (_TickEnd).

The SCL_p scheduling mechanics implies that the number of required priority changes as well as the number of jumps depends on the assignment of tsIDs to thread segments. Therefore, one may employ a heuristic which tries to minimize priority changes. In ABO, tsIDs have been assigned such that priority changes are not required and gotos have been minimized. Otherwise, a *prio switch* has to be inserted at the incoming dependency to allow for a potential context switch to the other thread. In ABO, even if HandleB starts first, the conditional checking B node must wait for the B assignment in HandleA.

Peiler explored different heuristic node assignment strategies and optimized required prio instructions. Re-scheduling is costly, especially when larger data structures are required to manage the priorities. He achieved a speed-up up to 50% in his experiments [Pei17]. Excerpts from the experimental results are provided in Section 5.5.2. *Remark*

To minimize storage requirements and to maximize the number threads/priorities which can be encoded with only scalar bit vectors, the range of prioIDs is compressed by skipping unused prioIDs.

While the highest determined prioID in ABO is originally 5, IDs 4 and 3 are unused. Thus, the SCL_p program for ABO shown in Fig. 5.3.2 uses prioIDs 2 for HandleA and the top thread segment of the root thread, 1 for HandleB and the bottom thread segment of the root thread, and 0 for the _TickEnd thread which is not shown explicitly here. *Example*

Since the priority-based approach requires an SCG to be in SIASC form, the node priority cannot be raised within one tick. However, this restriction does not apply to the prioID if the node priority stays unchanged.

Figure 5.3.3 shows a variant of ABO with an immediate feedback loop but no direct communication between the two threads. Therefore, all node priorities are 0. As the join node inherits the prioID from the lowest thread, *Example*

5. Interactive Compilation for SCCharts

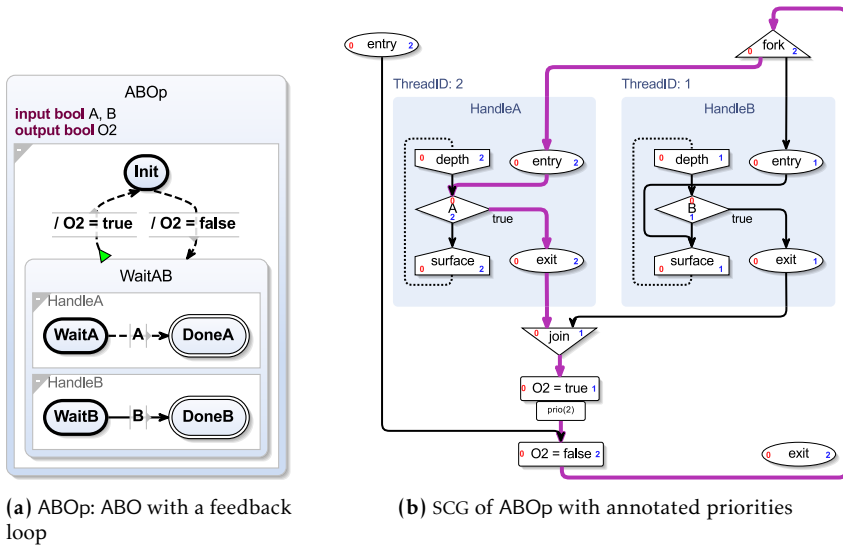


Figure 5.3.3. ABO variant compiled with the priority-based compilation

it is 1 after the join. It can be safely increased to 2 afterwards to realize the immediate feedback, because there is no concurrency involved. Similar to a priority switch in concurrent threads, the prio switch is used to raise the priority here.

5.3.2 Code Generation

After priority assignment, the model can be directly transformed into code. As before, C and Java serve as examples.

C Code Generation

As mentioned before, the code generation makes use of SCL_p macros, which are built upon the Synchronous C language [Han09b] in the C version.

The serialized code generated by the compilation is shown in Figure 5.3.4. The SCL_p macros are $tickstart(p)$, which starts the root thread with prioID p ;

5.3. Priority-Based Compilation

CODE - ABO.h [Open in Editor]	CODE - ABO.c [Open in Editor]
<pre>#ifndef ABO_H #define ABO_H /* * Automatically generated C code by * KIELER SCCharts - The Key to Efficient Modeling * * http://rtsys.informatik.uni-kiel.de/kieler */ #define false 0 #define true 1 typedef int bool; typedef struct { char A; char B; char O1; char O2; } TickData; void reset(TickData* d); int tick(TickData* d); #endif /* !ABO_H */</pre>	<pre>/* * Automatically generated C code by * KIELER SCCharts - The Key to Efficient Modeling * * http://rtsys.informatik.uni-kiel.de/kieler */ #include "ABO.h" #define _SC_NOTRACE #define _SC_NO_SIGNALS2VARS #define _SC_ID_MAX 2 #include "scl.h" #include "sc.h" #include "sc-generic.h" int tick(TickData* d) { tickstart(2); d->O1 = 0; d->O2 = 0; fork1(HandleA, HandleB, 1) { HandleA: label_0: if (d->A) { d->B = 1; d->O1 = 1; } else { pause; goto label_0; } } par { HandleB: label_1: pause; if (d->B) { d->O1 = 1; } else { goto label_1; } } join1(2); d->O1 = 0; d->O2 = 1; return tickreturn(); } void reset(TickData* d) { }</pre>

Figure 5.3.4. C code of ABO generated by the priority-based compilation approach

5. Interactive Compilation for SCCharts

```

1 // Boolean type
2 typedef int bool;
3 #define false 0
4 #define true 1
5
6 // Enable/disable threads with prioID p
7 #define _u2b(u) (1 << u)
8 #define _enable(p) _enabled |= _u2b(p); \
9 active |= _u2b(p)
10 #define _isEnabled(p) ((_enabled & _u2b(p))
11 !=0)
12 #define _disable(p) _enabled &= ~_u2b(p)
13
14 // Set current thread continuation
15 #define _setPC(p, label) _pc[p] = &label
16
17 // Pause, resume at <label>
18 #define _pause(label) _setPC(_cid, label); \
19 goto _L_PAUSE
20
21 // Pause, resume at pause
22 #define _concat_helper(a, b) a ## b
23 #define _concat(a, b) _concat_helper(a, b)
24 #define _label_ _concat(_L, __LINE__)
25 #define pause _pause(_label_); _label_:
26
27 // Fork/join sibling thread with prioID p
28 #define fork1(label, p) _setPC(p, label); \
29 _enable(p);
30 #define join1(p) _label_: if (_isEnabled(p)) \
31 { _pause(_label_); }
32
33 // Terminate thread at "par"
34 #define par goto _L_TERM;

```

```

85 int tick()
86 {
87     if (!_notInitial) { active = enabled; goto
88         _L_DISPATCH; } else { _pc[0] =
89         &&_L_TICKEND; enabled = (1 << 0); active =
90         enabled; _cid = 2; enabled |= (1 << _cid);
91         active |= (1 << _cid); _notInitial = 1; };
92     O1 = 0;
93     O2 = 0;
94     _pc[1] = &&HandleB; enabled |= (1 << 1);
95     active |= (1 << 1); {
96     HandleA:
97     if (IA) {
98         _pc[_cid] = &&_L94; goto _L_PAUSE; _L94;
99         goto HandleA;
100     }
101     B = 1;
102     O1 = 1;
103 } goto _L_TERM; {
104     HandleB:
105     _pc[_cid] = &&_L103; goto _L_PAUSE; _L103;
106     if (IB) {
107         goto HandleB;
108     }
109     O1 = 1;
110 } _L108: if (((enabled & (1 << 2)) != 0)) {
111     _pc[_cid] = &&_L108; goto _L_PAUSE; };
112     O1 = 0;
113     O2 = 1;
114     goto _L_TERM; _L_TICKEND: return (enabled !=
115     (1 << 0)); _L_TERM: enabled &= ~(1 << _cid);
116     _L_PAUSE: active &= ~(1 << _cid);
117     _L_DISPATCH: __asm volatile("bsrl %1,%0\n"
118     : "=r" (_cid) : "r" (active)); goto *_pc[_cid];

```

(a) Selected SCL_p macros. The address of label is obtained with &&label. The concatenation operator ## prevents macro expansion of its arguments, hence we need _concat_helper. __LINE__ expands to the current line number.

(b) ABO SCL_p tick function after macro expansion. The flag _notInitial is initially 0, indicating the initial tick.

Listing 5.3.2. Tick function synthesized for ABO (Figure 5.3.2) with expanded SCL_p macros.

5.3. Priority-Based Compilation

tickreturn, which contains the dispatching logic and returns 1 or 0 depending on whether the root thread is still running or not; pause, which pauses a thread until the next tick starts; fork $n(l_1, p_1, \dots, l_n, p_n)$, which forks off n sibling threads with start labels l_i and prioIDs p_i ; par, which acts as a thread barrier by terminating a thread; and join $n(p_1, \dots, p_n)$, which joins sibling threads with n different prioIDs p_i . Note that the join n is not performed by the parent thread, but by one of its child threads; the parent thread does not get started again until its children have terminated. To catch the termination of sibling threads instantaneously, the thread executing the join n must run at a prioID which is lower than that of these siblings. A further operator, not required in ABO, is prio(p), which allows to change the prioID of a thread. When a parent thread forks off child threads, the child thread which is started immediately after the fork gets to reuse the tsID of the parent thread. Similarly, after the join, the resuming parent thread reuses the tsID of the child thread performing the join. This implies that the forking of a single thread requires neither an extra tsID nor the associated bookkeeping information (resumption address etc.), which is one of the aspects where SCL_p is more efficient than the original Synchronous C.

Listing 5.3.2 shows selected SCL_p macro definitions and the result of macro-expanding ABO (gcc -E). The continuation point for the thread with prioID p is stored in `_pc[p]`. The prioID of the currently running thread is `_cid`. The threads which still have work to do in the current tick are represented in a bit-vector `active`, similarly `enabled` indicates threads which have not terminated yet. Thus the `enabled` bits are the inverse of the m (“empty”) flags computed in the data-flow approach. The macros presented here represent the bit vectors with scalars, thus the word size limits the maximum prioID. There are also alternative macros which use arrays instead and therefore do not have that limitation. As this is code for the x86, the dispatcher can be efficiently implemented as an embedded assembler instruction `bsrl` (bit scan reverse), which computes the highest set bit in a word, see line 112. For an Intel Core i7 (64 bit), gcc produces 626 bytes machine code for the ABO tick function, with no further need for an OS to handle the thread scheduling. For an embedded architecture with a smaller word size, the resulting executables would be even smaller.

5. Interactive Compilation for SCCharts

CODE - ABO.java	
<pre>/* * Automatically generated C code by * KIELER SCCharts - The Key to Efficient Modeling * * http://rtsys.informatik.uni-kiel.de/kieler */ public class ABO extends SJLProgramForPriorities<ABO.State> { public boolean A; public boolean B; public boolean O1; public boolean O2; public static enum State { ABOEntry, HandleA, _l_0, _l_1, _l_2, HandleB, _l_3, _l_4, _l_5, _l_6 } public ABO() { super(State.ABOEntry, 2, 2); } public void reset() {} }</pre>	<pre>@Override public void tick() { setupTick(); while (!isTickDone()) { switch (state()) { case ABOEntry: O1 = false; O2 = false; fork(State.HandleB, 1); gotoB(State.HandleA); if (true) break; case HandleA: if(A){ gotoB(State._l_0); } else { gotoB(State._l_1); } if (true) break; case _l_0: B = true; O1 = true; termB(); if (true) break; case _l_1: pauseB(State._l_2); if (true) break; case _l_2: gotoB(State.HandleA); if (true) break; case HandleB: pauseB(State._l_3); if (true) break; case _l_3: if(B){ gotoB(State._l_4); } else { gotoB(State.HandleB); } if (true) break; case _l_4: O1 = true; gotoB(State._l_5); if (true) break; case _l_5: if (!join(2)) { pauseB(State._l_5); if (true) break; } case _l_6: O1 = false; O2 = true; termB(); if (true) break; } } }</pre>

Figure 5.3.5. Java code of ABO generated by the priority-based compilation approach (column break edited manually)

Java Code Generation

The Java code generation uses the same path as the C code generation. As Java and similar languages do not support computed gotos, which have been

used in the C version, a framework that emulates this features is necessary. Nonetheless, every step before the code serialization can be re-used. The model can be transformed to object-oriented classes similar to the Java code generation of the netlist-based approach. The class generated for a particular model is based on Synchronous Java (SJ) [MHH12]. SJ uses a switch-case pattern (see Section 2.3.7 on page 49). Every potential scheduling target is situated in an own case block. Whenever the scheduler must be called, the switch is exited via a break instruction. The enclosing loop selects a new continuation point and restarts the switch statement. The dispatcher is hidden in the superclass. Since it cannot resort to assembler instructions, such as bit scan reverse, all priority handling is done straightforwardly in standard data structures, such as arrays. The generated code for ABO is listed in Figure 5.3.5.

5.4 State-Based Compilation

The two initially proposed code generation approaches discussed previously target the SCMoC with its synchronous inter-thread communication. However, in some domains, while still important, this is not of primary concern. Providing a rich set of features often is a trade-off w.r.t. simplicity, understandability and readability. The state-based approach, presented now, follows a different route than before and focuses on the latter.

To formulate goals, Section 5.4.1 gives a set of common requirements which can be applied to synchronous languages and statechart dialects in general. Developers must choose for themselves what is of most importance. Afterwards, Section 5.4.2 – Section 5.4.4, discuss three variations of the state-based compilation approach.

5.4.1 The Ten Requirements

As discussed in the related works in Section 2.3, the basic idea of statecharts seems fairly straightforward. However, there is much room for interpretation of what their precise semantics is. While von der Beeck [Bee94] provides an extensive comparison between different statecharts dialects, I here present the minimal and reasonable requirements for the realm of safety-critical

5. Interactive Compilation for SCCharts

systems that the KIELER team agreed upon. The selected requirements are separated into *semantic*, *code* and *compiler requirements*.

Semantic Requirements

Semantic requirements specify the minimal requirements to a language semantics.

Requirement R1. *The semantics shall be deterministic.*

Especially for the synchronous languages and therefore SCCharts, this means that the produced sequence of outputs should be fully determined by the sequence of inputs. More technically, given a certain state of the statechart, calling the tick function with a specific input tuple always results in the same output tuple. In particular, there should be no *race conditions* due to concurrent variable accesses, and there should be no ambiguities due to multiple simultaneously enabled outgoing transitions from a state.

This, for example, is not necessarily fulfilled for UML statecharts, for which the following applies: “if more than one guard evaluates to true, then the model is ill-formed. In such a case, the statecharts semantics stipulate that only one of the transitions will be taken, but one cannot predict which one it will be” [Dou99]. It also rules out the original statecharts semantics, implemented in STATEMATE, where transitions were not prioritized and thus could be enabled simultaneously, which leads to non-deterministic behaviour. In the original implementation in STATEMATE, it could not statically be checked whether this could possibly occur or not. If the statechart was simulated, the simulator detected such cases and asked the user to pick one of the enabled transitions. If the statechart was synthesized, the code generator implicitly prioritized transitions such that one of the enabled transitions gets taken, but this was outside of the control of the modeller.

Requirement R2. *The semantics shall be robust in that it should not depend on details of the visual representation of the model or the naming of model elements.*

The idea is that there should be no “surprises” with changes of semantics due to the visual representation of the model. For example, the positioning of transitions should not affect the semantics.

5.4. State-Based Compilation

This contrasts with, e. g. the *12 o'clock rule* of Simulink/Stateflow, where transitions are implicitly prioritized by their angular position relative to the source state [Ham05]. In Yakindu¹, regions are scanned “either from left to right or from top to bottom” (see also R3), but since regions can generally be arranged two-dimensionally, this does not always provide a clear ordering.

Requirement R3. *The semantics shall include true concurrency.*

This means that not only should the semantics permit concurrent regions that both have their own, independent state, but that these regions should also be able to react concurrently within the same tick, and they should be able to interact with each other. For example, if one region specifies that some light should turn on when a door is opened, and a concurrent region should increment an open-door counter, then both of these actions should take place when the door opens, not just one of them.

This contrasts with, e. g. the *virtual concurrency* implemented in Yakindu Statecharts, where only one of these actions will take place and the choice of action depends on the arrangement of the regions (see R2). In LabVIEW², the “statechart enters orthogonal regions in descending alphabetical order of the region name”, which appears to permit multiple regions to react within a tick, but still precludes back-and-forth communication between concurrent modules within a tick, and also is not robust in the sense of R2. Similarly, it is debatable whether UML statecharts and their run-to-completion execution model that serializes all events is truly concurrent; at least it seems difficult to state whether, e. g. two transitions are simultaneous or not.

Code Requirements

Similar to the semantic requirements, such as those stated in Section 5.4.1, *code requirements* formulate a number of concerns which the synthesized code should fulfil.

Requirement R4. *The code shall be self-explanatory, and it shall be easy to match the generated code with the original model and vice versa.*

¹<https://www.itemis.com/en/yakindu>

²<http://www.ni.com>

5. Interactive Compilation for SCCharts

This implies that the *structure (topology)* of model and code should match, e. g. that for each region or (super) state in the model there should be a corresponding piece of code. Likewise, the *naming* of variables and functions in the code should facilitate the mapping, e. g. a function which implements the behaviour of some region should be named according to the region. Furthermore, the *order* of declarations and code segments should, as far as possible, be consistent with any order present in the model. As discussed for R2, there is not necessarily an obvious order of regions; but if there is an obvious order in the model, that order should be reflected in the code, unless there is a clear reason to change that order, e. g. to efficiently implement a scheduling requirement of the model. Generally, names should be rather speaking and not be overly abbreviated. Similarly, the generated code should include comments which facilitate understanding. Fulfilment of this requirement should also support debugging of the generated code, i. e. when stepping through the code, it should be clear what the code is doing even without referring to the original model.

Requirement R5. *The generated code shall be a safe subset of the target language.*

For example, the C language, which is still one of the most widely used languages in particular in the embedded area, has been designed with efficiency in mind, not robustness. It thus includes a number of features which are considered unsafe and are typically not permitted for safety-critical applications, such as *pointer arithmetic*, *complex macros*, *function pointers*, *break*, *gotos* (also in generated code) or *continue* statements [Hat95; MIS13].

Requirement R6. *The generated code shall not rely on external thread support.*

Conceptually, each statechart region is a *thread* in the sense of a concurrent flow unit. However, that concurrent control-flow should be managed within the tick function without making use of thread libraries, such as POSIX threads³. Similarly, the code should not make use of thread-synchronizing constructs, such as semaphores or monitors. The rationale is that one wants to make the code self-contained and in full control of

³<https://standards.ieee.org>

scheduling decisions. In particular, one wants to avoid non-determinism (considering also R1) as introduced by POSIX threads or, for that matter, Java threads [Lee06].

Compiler Requirements

The last set of requirements targets the compiler framework and the result of its compilations. While the first two in general overlap with the code requirements in providing efficient and predictable code, the last two requirements formulate characteristics of the compiler itself.

Requirement R7. *The program execution shall be predictable with respect to timing and memory usage.*

Besides being deterministic (see R1), safety-critical applications must guarantee specific behaviour towards timing and memory consumption. Therefore, to make sure that a program always operates in a specified range w.r.t. to reaction time and memory usage, usually worst-case analyses are performed. These analyses give upper bounds for the execution time of the tick function and therefore determine the overall frequency of the system. In general, a small jitter in the execution time is preferred to avoid a slowdown of the whole system because even rarely occurring execution time spikes must always be able to execute correctly.

Requirement R8. *The execution shall be efficient with respect to timing and memory usage.*

Reducing the jitter in execution times or memory consumption prevents a slowdown of the system because of single spikes. However, this does not avoid bad scaling w.r.t. to model sizes. In general, the execution time and the memory consumption should not become inefficient with growing model sizes.

Requirement R9. *The compiler shall be easy to maintain.*

While it is entirely possible to write a specialized compiler for every feature in a statecharts dialect, one usually wants to reduce the necessary

5. Interactive Compilation for SCCharts

<p>R1 <i>The semantics shall be deterministic!</i></p> <p>R2 <i>The semantics shall be robust in that it should not depend on details of the visual representation of the model or the naming of model elements!</i></p> <p>R3 <i>The semantics shall include true concurrency!</i></p>	<p>R4 <i>The code shall be self-explanatory, and it shall be easy to match the generated code with the original model and vice versa!</i></p> <p>R5 <i>The code shall be a safe subset of the target language!</i></p> <p>R6 <i>The code shall not rely on external thread support!</i></p>	<p>R7 <i>The execution shall be predictable with respect to timing and memory usage!</i></p> <p>R8 <i>The execution shall be efficient with respect to timing and memory usage!</i></p>
		<p>R9 <i>The compiler shall be easy to maintain.</i></p> <p>R10 <i>The tooling shall provide meaningful model guidance!</i></p>

Figure 5.4.1. Requirements Overview

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Netlist-based Compilation	X	X	XX		X	X	X		X	X
Priority-based Compilation	X	X	XX			X		X	X	X
State-based Compilation	X	X	XX	X	X	X			X	XX
Lean State-based Compilation	X	X	X	XX	X	X		X	X	XX
CS State-based Compilation	X	X	X	XXX	X	X		X		XX

Figure 5.4.2. Requirements checklist for KIELER

maintenance effort. It is common to translate more complex language constructs into a kernel language, which the compiler can handle, or an IR of other compilers that are invoked afterwards. As will be demonstrated in the next sections, there is often a trade-off between maintainability and readability.

Requirement R10. *The tooling shall provide meaningful model guidance and error reporting.*

As discussed in Chapter 4 on page 95 ff., a modelling/compiler framework can guide the modeller significantly. When building tools, it should always be a design concern on how to help the modeller best and how easily this can be achieved.

The Requirements Trade-Off

The requirements are summarized in Figure 5.4.1. As mentioned before, there is rarely a one-fits-all approach. The best approach should be chosen

5.4. State-Based Compilation

according to the actual requirements of a project. The checklist of the compilation approaches within KIELER is depicted in Figure 5.4.2. For the two approaches discussed earlier, the situation is reasonably clear. Both satisfy all semantics requirements, which is no surprise as they were designed with deterministic synchronous semantics. Regarding code requirements, they fulfil R6, but both lack R4 as the code is arguably not mappable except for a compiler expert. However, the netlist-based approach is considered worse in terms of readability. In the priority-based approach the macros expansion and interleaving w.r.t. priorities impede understandability. The non-concurrent flow is relatively readable. Additionally, at least in the C variant, the priority-based approach uses computed gotos, which may not be regarded as safe subset of C and hence the approach also fails R5. As for the compiler requirements, Section 5.5 shows that the netlist-based approach generally produces binaries with low jitter, whereas the priority-based approach scales better with increasing model sizes. Both approaches are maintainable in KIELER due to the modular KiCo framework and provide meaningful model feedback during development, as discussed in Chapter 4 and in Sections 5.2 and 5.3.

The goal for the state-based approach is to meet R4. While it seems that improving the netlist-based approach to reach R4 is a reasonable choice because it already fulfils all other requirements up to a satisfactory level, making the generated netlist self-explanatory seems hardly possible. Hence, the priority-based approach serves as common starting point for all state-based variations, which are explained in the following sections.

Basically, the state-based approach is a modification of the common state code generation pattern, explained in Section 2.3.7 on page 49 ff. The hierarchy of the generated functions follows the topology of the statechart. Switch patterns are used to select the appropriate region and state functions. However, to achieve R4 without sacrificing R5 or R6, interleaving of threads has to be done manually, which sacrifices efficiency. Therefore, two other variants of the state-based approach, namely Lean and Lean Common Set (LCS), which both only support uni-directional communication between regions within a tick, have been implemented and tested. Additionally, the code generator of the latter also directly supports common language features, which increases readability but sacrifices maintainability.

5. Interactive Compilation for SCCharts

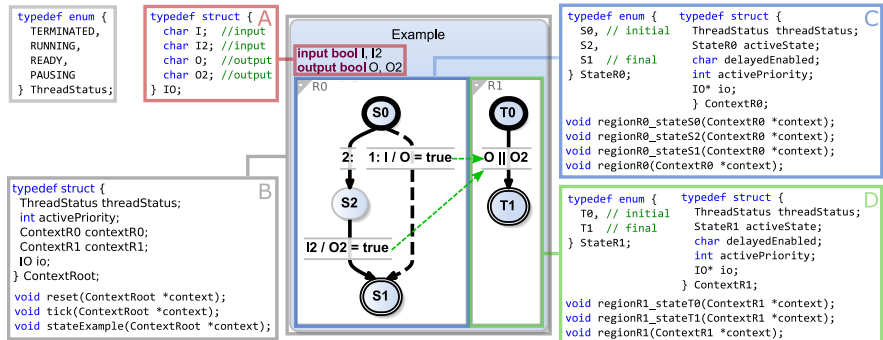


Figure 5.4.3. Topology-preserving code-generation: The SCChart’s interface gets transformed into the interface IO struct (red A); the root state creates the main root context and the functions which the environment should call, namely reset and tick (gray B); the R0 region gets transformed into the ContextR0 struct with its accompanying functions (blue C); similarly, R1 is transformed into the ContextR1 (green D). The green dashed arrows indicate *data dependencies*, which influence the scheduling order.

5.4.2 The General State-Based Compilation Approach

As explained, the state-based compilation approach is driven by the coding requirements from Section 5.4.1. The approach is now part of the KIELER releases and is successfully used in real-life industrial projects.

Priority-Based Concurrency

To implement concurrency (R3) without external thread support (R6), the state-based approach adopts the concept of node *priorities* from the priority-based compilation with an explicit light-weight application-level thread concept. The basic idea is that concurrent regions are scheduled according to a statically computed priority. The generated code includes a *dispatcher*, which, within a tick, determines which regions are eligible for execution. Among those, it dispatches the ones with the highest priority, and keeps doing so until no more region has any work to do in the current tick. If there are multiple eligible regions with highest priority, there should be

5.4. State-Based Compilation

a deterministic ordering for these regions, typically given by their order in the code. Therefore the code generator may use both, priorities and the region ordering in the code, to implement scheduling constraints given by the statechart semantics.

There are various options on how to implement the dispatcher, which affect both code complexity and efficiency. To facilitate R4, the state-based approach uses a straightforward method, which first scans all regions which are ready to execute for their maximum priority, and runs them if they are ready and have that highest priority. This appears to perform reasonably well in most cases (see also Section 5.5.4), but for models with many regions per superstate and many priority changes, alternative schemes with additional data structures might be more efficient. In either case, this dispatching is a comparatively light-weight application-level context switching mechanism, typically more efficient than OS threading.

Priorities can be determined in a number of ways. For example, if one wishes to implement the semantics of LabVIEW statecharts, then this priority could be assigned according to the lexical order of region names (notwithstanding the robustness issue discussed on R2). To implement the concurrent scheduling of Céu [SIL+17] or PRET-C [ARG10], which executes concurrent threads sequentially according to their syntactic order in the program, the priority could be assigned according to that order. Alternatively, for both of these schemes, one could assign all regions the same priority and just generate code such that the regions appear in the correct order. However, to allow back-and-forth interaction of concurrent regions (R3), the state-based approach follows the same algorithm for priority generation as the priority-based approach discussed in Section 5.3.1, meeting R3. In the example of Figure 5.4.3, R0 writes a variable (O2) which is read concurrently by R1. This induces a *data dependency*, indicated with a dashed arrow, and implies that R0 must be executed before R1 and, therefore, receives a higher priority.

Note that in KIELER the state-based approach compiles down to the SCG *Implementation* and re-uses the same priority processor that is used for the priority-based approach. Afterwards, the SCG is reversely transformed into a new SCChart with priority annotations. While this path exploits the modularity of KiCo processors, the SCG transformation could be skipped if a priority processor that works on the SCCharts meta-model is used.

5. Interactive Compilation for SCCharts

Preserving Topology

The basic topology mapping is straightforward. Figure 5.4.3 shows how different parts of the SCCharts model get transformed into C code. The *interface* of the SCChart is translated into a dedicated IO struct (red A). Comments are added to give additional information about inputs and outputs. The root state is transformed into a ContextRoot struct which holds the interface, a ThreadStatus, and *sub-contexts* from enclosed regions (dark gray B). The functions which can be called from the environment are reset and tick. The reset function initializes the state machine and is usually called at start-up. Then, tick can be invoked periodically to calculate exactly one discrete tick of the automaton. The regions inside of the root state are then transformed similarly (blue C and green D). Each region has a set of states, an own context struct and a set of functions which mimic the topology of the statechart. The details of each are described in the following.

Thread Status

A thread can be in one of four different statuses, see Figure 5.4.4. A READY thread is ready and waits for its execution. RUNNING indicates that the thread is currently executing. If a thread has finished its reaction for a tick, it sets itself to PAUSING. If it ceases to exist, it is TERMINATED and can only be re-spawned from a higher hierarchy.

These four states are a flattened and slightly refined encoding of the enabled/active flags of the original SC-Charts proposal [HDM+14]: disabled threads are TERMINATED; enabled and inactive threads are PAUSING; enabled and active threads are READY or RUNNING, depending on whether they are currently dispatched or not. In the code, the thread status is stored in the ThreadStatus member (light grey, Figure 5.4.3).

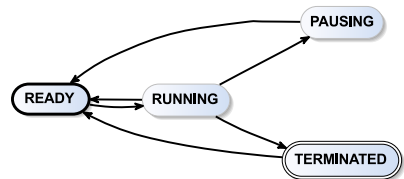


Figure 5.4.4. Thread status in priority-based concurrency

Superstates

As explained earlier, the priorities are calculated statically. The concurrent data dependencies, shown as green, dashed edges in Figure 5.4.3, determine the priorities in this example. Static thread priorities are set when a superstate is entered. Note that the root state is a superstate with an interface for the environment; any superstate inside an SCCharts model is constructed in the same way. For the root state, the priorities are set inside the reset function, which is called when the program is initialized.

Figure 5.4.5a shows the internals of the reset function for the example *Example* from Figure 5.4.3. Besides initializing the thread context, the previously calculated priorities for R0 and R1 are set. As the dependency edges in Figure 5.4.3 indicate, R0 must be executed before R1 and, therefore, receives the higher priority.

Eventually, the root state's function `stateExample` is called, illustrated in Figure 5.4.5d. As with every superstate, its purpose is to determine which contained region is allowed to run in which order and to adjust the priorities accordingly. At the beginning of the function, all threads which are still able to run are set to `RUNNING` and are invoked sequentially (see the red box A in Figure 5.4.5d). All threads which are still ready to run contribute to the new priority, which is set afterwards (blue box B). If all threads finished their execution for the active tick, the priority is set to the maximum of the paused threads for the next tick (green box C). Eventually, all threads have completed their tick and the superstate checks if it must terminate itself (grey box D). The control then returns to the caller. Note that the code can be further optimized if a superstate contains only one region or if the static priorities do not change. —

Regions

Every region inside a superstate gets its own context. For each region, the context struct, an enum with the included states, and a function for every state plus one function for the whole region is created. As before, the context contains a thread status, and a pointer to the interface. It also holds the active state, a flag which signals if delayed transitions are enabled, and an active priority. The functions are named appropriately and called when the

5. Interactive Compilation for SCCharts

```
void reset(ContextRoot *context) {
    context->contextR0.io = &(context->io);
    context->contextR1.io = &(context->io);

    context->contextR0.activeState = S0;
    context->contextR0.delayedEnabled = 0;
    context->contextR0.activePriority = 2;
    context->contextR0.threadStatus = READY;

    context->contextR1.activeState = T0;
    context->contextR1.delayedEnabled = 0;
    context->contextR1.activePriority = 1;
    context->contextR1.threadStatus = READY;

    context->activePriority = 1;
    context->threadStatus = RUNNING;
}
```

(a) Code structure of the reset function

```
void regionR0(ContextR0 *context) {
    /* Cycle through the states of the region as long as this thread
    * is set to RUNNING. */
    while(context->threadStatus == RUNNING) {
        switch(context->activeState) {
            case S0:
                regionR0_stateS0(context);
                break;
            case S2:
                regionR0_stateS2(context);
                break;
            case S1:
                regionR0_stateS1(context);
                break;
        }
    }
}
```

(b) Code structure generated from a region

```
void regionR0_stateS0(ContextR0 *context) {
    /* Transition 0: immediate to final state S1
    * Trigger/Effects: I / O = 1 */
    if (context->io->I) {
        context->io->O = 1;
        context->activeState = S1;
        context->delayedEnabled = 0;
    } else if (context->delayedEnabled) {
        /* Transition 1: delayed to state S2
        * This is the default transition, the trigger is always true. */
        context->activeState = S2;
        context->delayedEnabled = 0;
    } else {
        // Wait for next tick if no transition was taken.
        context->threadStatus = PAUSING;
    }
}
```

(c) Code structure generated from a state

```
void stateExample(ContextRoot *context) {
    int newActivePriority = 0;

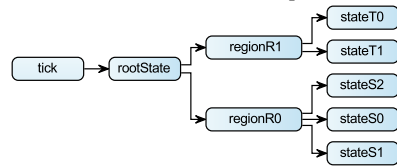
    if (context->contextR0.threadStatus == READY) {
        if (context->contextR0.activePriority == context->activePriority) {
            context->contextR0.threadStatus = RUNNING;
            // Call the logic code of thread R0.
            regionR0(&context->contextR0);
        }
        if (context->contextR0.threadStatus == READY) {
            if (context->contextR0.activePriority > newActivePriority) {
                newActivePriority = context->contextR0.activePriority;
            }
        }
    }
    if (context->contextR1.threadStatus == READY) {
        if (context->contextR1.activePriority == context->activePriority) {
            context->contextR1.threadStatus = RUNNING;
            // Call the logic code of thread R1.
            regionR1(&context->contextR1);
        }
        if (context->contextR1.threadStatus == READY) {
            if (context->contextR1.activePriority > newActivePriority) {
                newActivePriority = context->contextR1.activePriority;
            }
        }
    }

    // Set the new priority.
    context->activePriority = newActivePriority;

    // Calculate the priority for the next tick if necessary.
    if (context->contextR0.threadStatus != READY &&
        context->contextR1.threadStatus != READY) {
        if ((context->contextR0.threadStatus == PAUSING) &&
            (context->contextR0.activePriority > context->activePriority)) {
            context->activePriority = context->contextR0.activePriority;
        }
        if ((context->contextR1.threadStatus == PAUSING) &&
            (context->contextR1.activePriority > context->activePriority)) {
            context->activePriority = context->contextR1.activePriority;
        }
    }

    // Check if the root state must terminate.
    if (context->contextR0.threadStatus == TERMINATED &&
        context->contextR1.threadStatus == TERMINATED) {
        context->threadStatus = TERMINATED;
    }
}
```

(d) Code structure of a superstate



(e) Hierarchical call tree of the generated code

Figure 5.4.5. Generated code of the topology-preserving code generation

corresponding element in the SCChart is active.

Example

The call stack for these functions is constructed hierarchically, as depicted in Figure 5.4.5e. The environment sets the inputs and calls the tick function to compute the reaction for one tick. The tick function then calls

5.4. State-Based Compilation

the root state's function stateExample. As the root state includes the two regions regionR0 and regionR1 in the example, their functions are called next. Each region function is then responsible for calling the active state functions. Note that the order in which the regions are executed depends on their priority. They can also be interleaved.

As stated before, the superstate function calls the functions responsible for the regions in the correct order. Such a function is straightforward, as shown in Figure 5.4.5b. The region code is looped as long as the thread status is RUNNING. The contained switch block selects the correct function for the actual state and calls it. A reaction inside of a state function can set the thread status to READY, PAUSING, or TERMINATED to yield, so that the region while loop will be left. READY means that the thread releases its control so that another thread can continue. This happens when the priority of a thread changes. PAUSING signals that the thread finished its reaction for this tick. Paused threads will be set to READY at the beginning of the next tick. TERMINATED indicates that the region has reached a final state and is terminated until invoked again from a higher hierarchy.

States

The reactions occur inside the state functions.

Figure 5.4.5c illustrates the source code of the state S0. Here, for every outgoing transition it is checked whether or not the transition is eligible to fire. Therefore, the *delay status* and the *trigger* of all transitions are checked in order of their transition priorities. S has two such transitions: One immediate transition with index 1 to S1 with I as trigger, and one delayed transition with index 2 without trigger to S2. According to the transition index, the function first checks if I is true. If so, the reaction of that transition is executed: The output O is set to true and the control is handed over to S1 while setting the delayed flag to false. If the transition cannot fire, it is checked if the delayed transition is eligible to run. If so, the control is transferred to S2. If no transition can be taken, the thread is set to PAUSING to signal the reaction's end for this tick.

Note that extended information of the transitions and their reactions are annotated as comments by the code generator. These comments are

5. Interactive Compilation for SCCharts



Figure 5.4.6. Lean state-based compilation system

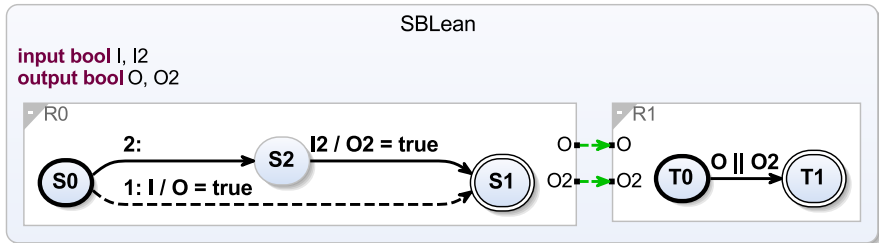


Figure 5.4.7. State-based example SCChart with region dependencies

also added to the header file where the function declarations are stored. While it is still necessary to validate the source code inside the source files, the transition reactions can also be seen inside the header files to facilitate readability and understandability.

Conclusion

The granularity of thread interleaving depends on the topology of the statechart. As described before, control is given back to the parent region if a priority changes. However, the state function only returns after a transition reaction has been completed, which could comprise multiple actions if the SCChart is in core form. The SCChart can be transformed into normalized form to allow a finer interleaving of transition actions.

5.4.3 The Lean State-Based Approach

Much of the superstate code shown in Figure 5.4.5d is required for the bidirectional communication between threads. Threads release their control whenever the priority changes. The control is given back to the parent, which then calculates the new active priority and calls the appropriate function. While the code of the other functions (Figure 5.4.5a – Figure 5.4.5c) is self-explanatory, the priority handling impairs readability. Therefore, the lean state-based approach refrains from bidirectional communication.

Regions are automatically topologically ordered according to their dataflow dependencies. If there is no cycle in the dependencies, the code can be executed statically in that order and no priorities are needed.

```
static inline void SBLean(TickData *context) {
    if (context->SBLean_regionR0.threadStatus != TERMINATED) {
        context->SBLean_regionR0.threadStatus = RUNNING;
        SBLean_regionR0(&context->SBLean_regionR0);
    }

    if (context->SBLean_regionR1.threadStatus != TERMINATED) {
        context->SBLean_regionR1.threadStatus = RUNNING;
        SBLean_regionR1(&context->SBLean_regionR1);
    }

    context->SBLean_regionR0.delayedEnabled = 1;
    context->SBLean_regionR1.delayedEnabled = 1;
    context->threadStatus = READY;
}
```

Figure 5.4.8. Superstate code of the lean state-based approach for the example from Figure 5.4.3

the topology of the source SCChart without the need of any priorities.

For the example from Figure 5.4.3, the intermediate model of the region *Example* dependency processor is shown in Figure 5.4.7. All region dependencies are unidirectional and determine the call order within the code. The superstate code, shown in Figure 5.4.8, is now considerably smaller without the priority handling.

This is listed accordingly in the rating in Figure 5.4.2 on page 198. The lean state-based approach is more readable and efficient than its predecessor at the cost of bidirectional communication. While this seems to be a significant sacrifice, it must be decided which of the listed criteria are more important for a project. In fact, the lean approach was chosen more often in industrial projects as it is easier to understand, more maintainable and bidirectional communication within the same tick instance was not required.

5.4.4 The Lean State-Based Approach of the Common Set

As described in Section 5.1, every SCChart is usually transformed into a normalized SCChart that only comprises SKPs. Depending on the granularity of potential interleaving, the previously mentioned state-based approaches can also manage Core SCCharts directly. However, with the goal of retaining the

5. Interactive Compilation for SCCharts

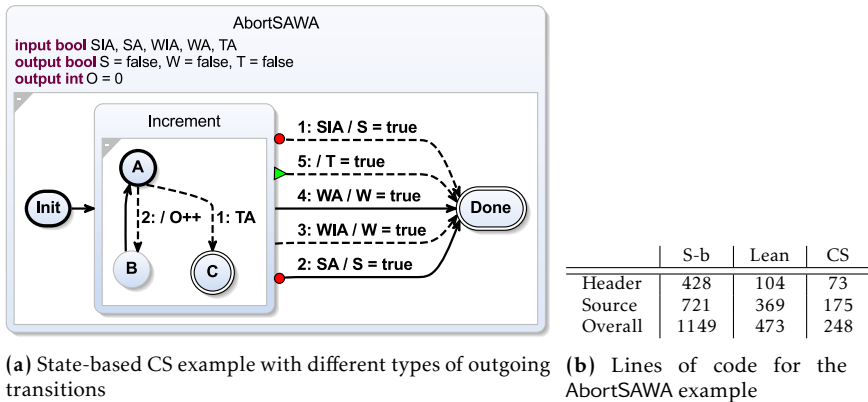


Figure 5.4.9. Leaving superstates: AbortSAWA

topology of the statechart, it is not always preferable to compile all features away. The last variant of the state-based approach implemented in KIELER, *common set*, builds upon the lean approach and preserves the extended features *abort* and *actions* of SCCharts. It specifically targets aborts and actions because these features are commonly used in synchronous languages and their extended transformations in SCCharts obfuscate the readability of the generated code. Similar to the lean approach, it only allows unidirectional communication between regions. Therefore, the code generator needs to be able to serialize code for these features directly, losing generalizability and maintainability but increases readability. The trade-off is also indicated in Figure 5.4.2 on page 198.

Figure 5.4.9a shows an SCChart comprising a superstate with five different kinds of outgoing transitions. By not transforming the abort into more basic core features and serializing it directly to code, only 21% of the original LoC are needed, which is listed in Figure 5.4.9b. The direct serialization for this feature can be done straightforwardly. Strong aborts have to be checked before any functions of the inner behaviour of the superstate is called. Weak aborts afterwards. The termination is only triggered if all inner regions terminated. If the transition is delayed, a check for `delayedEnabled` must be added. The generated code is depicted in Figure 5.4.10.

5.4. State-Based Compilation

```
static inline void AbortSAWA_regionR0_stateIncrement_running(AbortSAWA_regionR0Context *context) {
    if (context->AbortSAWA_regionR0_stateIncrement_regionR1.threadStatus != TERMINATED) {
        context->AbortSAWA_regionR0_stateIncrement_regionR1.threadStatus = RUNNING;
    }

    if (context->iface->SIA) {
        context->iface->S = 1;
        context->xdelayedEnabled = 0;
        context->xactiveState = DONE;
        return;
    } else if (context->xdelayedEnabled && (context->iface->SA)) {
        context->iface->S = 1;
        context->xdelayedEnabled = 0;
        context->xactiveState = DONE;
        return;
    }

    AbortSAWA_regionR0_stateIncrement_regionR1(&context->AbortSAWA_regionR0_stateIncrement_regionR1);

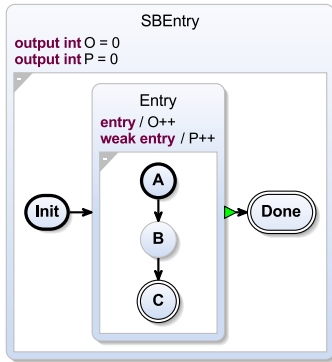
    if (context->iface->MIA) {
        context->iface->W = 1;
        context->xdelayedEnabled = 0;
        context->xactiveState = DONE;
    } else if (context->xdelayedEnabled && (context->iface->WA)) {
        context->iface->W = 1;
        context->xdelayedEnabled = 0;
        context->xactiveState = DONE;
    } else if (context->AbortSAWA_regionR0_stateIncrement_regionR1.threadStatus == TERMINATED) {
        context->iface->T = 1;
        context->xdelayedEnabled = 0;
        context->xactiveState = DONE;
    } else {
        context->AbortSAWA_regionR0_stateIncrement_regionR1.delayedEnabled = 1;
        context->xthreadStatus = READY;
    }
}
```

Figure 5.4.10. Lean CS State-based abort serialization

Similarly, actions can be translated straightforwardly. During actions can be executed just like inner behaviour and exit actions are executed if the superstate is left. Entry actions, however, should only be executed once whenever the state is entered anew. Since the actual state is set by the state that performs the state change, an intermediate step is introduced so that every state change to the superstate with the entry action is treated identically.

Figure 5.4.11a shows an SCCharts with a superstate and two entry actions. *Example* These actions must be executed if the state is entered but not if control starts in the state in a new tick. Therefore, a new superstate level is introduced between the initialization of the superstate and its running function, called entry. This can be implemented as fall-through in the switch block of the region function which is responsible for calling the superstate, which is shown in Figure 5.4.11b. The calling procedure in the states which perform a state change to the superstate remain untouched.

5. Interactive Compilation for SCCharts



(a) State-based CS example with an entry action

```
static inline void SBEntry_regionR0(SBEntry_regionR0Context *context) {
  while (context->threadStatus == RUNNING) {
    switch (context->activeState) {
      case INIT:
        SBEntry_regionR0_stateInit(context);
        break;
      case ENTRY1:
        SBEntry_regionR0_stateEntry(context);
        // Superstate: intended fall-through
      case ENTRYENTRY:
        SBEntry_regionR0_stateEntry_entry(context);
        // Superstate: intended fall-through
      case ENTRY1RUNNING:
        SBEntry_regionR0_stateEntry_running(context);
        break;
      case DONE:
        SBEntry_regionR0_stateDone(context);
        break;
    }
  }
}
```

(b) State-based CS example code for entry actions

Figure 5.4.11. Entry action code for superstates in Lean CS

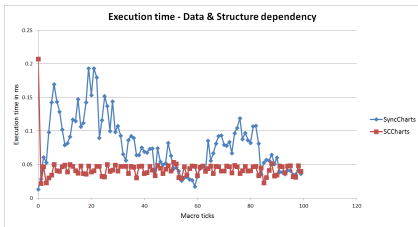
5.5 Evaluation of Compilation Approaches

The efficiency of the generated code w.r.t. to size and execution time was measured during several periods of the KIELER development. This section presents the results from the evaluation of the initial SCCharts contribution [HDM+14] in 2014 (Section 5.5.1), from the priority-based evaluation according to Peiler (Section 5.5.2) in 2017, from the hardware demonstrator evaluation according to Boysen (Section 5.5.3) in 2020 and the final run-time evaluation in this theses (Section 5.5.5) in 2020. The state-based approach was evaluated in two case-studies to verify its usefulness towards manual verification. The results are presented in Section 5.5.4. An in-depth analysis of over 100 questionnaires distributed to users of SCCharts and KIELER, which covers a wide range of questions beyond code efficiency, follows in Section 6.6.

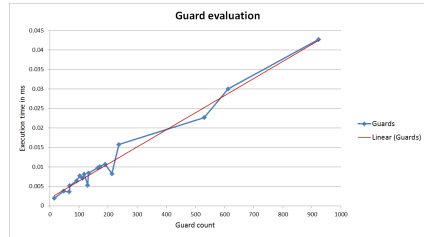
5.5.1 Initial Performance Evaluation

The initial evaluation of the first implementation of the netlist-based (a.k.a. dataflow) approach for SCCharts compared the approach to the SyncCharts

5.5. Evaluation of Compilation Approaches

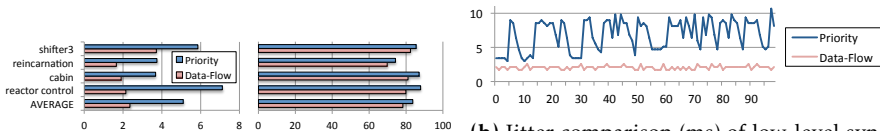


(a) Execution time comparison of models with nested hierarchy levels

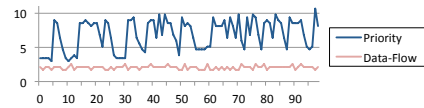


(b) Relation between execution time and generated guard count

Figure 5.5.1. Initial benchmarks of the netlist-based approach (from [Smy13])



(a) Avg. tick times (ms) (left) and exec. size (KB) (right), priority vs. dataflow low-level comp.



(b) Jitter comparison (ms) of low-level synthesis approaches, for cabin running 100 ticks

Figure 5.5.2. Benchmarks of the initial SCCharts contribution (from [HDM+14])

implementation of KIELER, which basically followed the priority-based approach [Smy13]. It has already shown two peculiarities, which are illustrated in Figure 5.5.1. Firstly, the execution time *jitter* is steadier when using the netlist-based approach, as the whole netlist is calculated in every tick. This is shown for one test of that benchmark in Figure 5.5.1a. The approach also performed better for small models on average. However, as the execution times in priority-based approach depend on the structure of the generated SCG, this is not a necessity. Secondly, the execution times of the netlist-based approach are bound linearly to the amount of guards generated during the approach, which is shown in Figure 5.5.1b.

The first comparison between the compilation approaches of SCCharts was published alongside the initial language contribution in 2014 [HDM+14] and confirm the initial benchmark results. The results are illustrated in Figure 5.5.2. The execution time benchmark compared tick cycles and code

5. Interactive Compilation for SCCharts

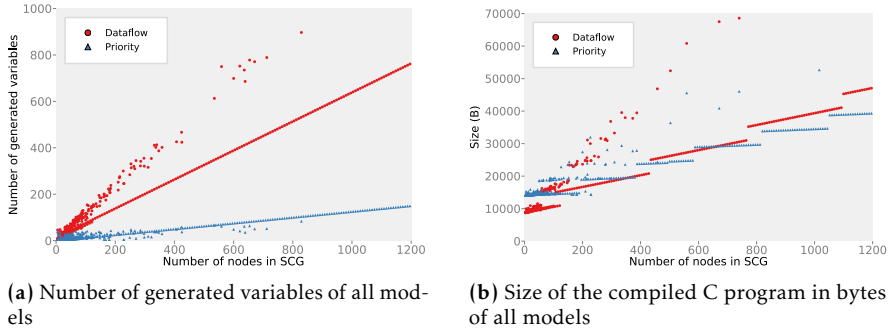


Figure 5.5.3. Comparison of code sizes according to Peiler (from [Pei17])

sizes between the netlist-based and priority-based compilation approach of small-to-mid size models. Figure 5.5.2a shows that the netlist-based approach performed better w.r.t. execution time for small models. While the netlist-based approach has to calculate the whole netlist anew in every cycle, the linear control-flow structure allows a fast execution. Figure 5.5.2b illustrates for one of the benchmarks the execution times per reaction for a sequence of reactions. Not surprisingly, the execution times of the priority approach show a variance, i. e. a high jitter. The data-flow approach has a much steadier response time as there is basically no internal control-flow which depends on the inputs and the internal state. The size differences between the two approaches are less significant for these model sizes.

5.5.2 Comparison According to Peiler

While the netlist-based approach is more efficient w.r.t. to execution times for small models, the priority-based approach scales better as the model size increases, which was investigated further by Peiler [Pei17].

Figure 5.5.3 shows the sizes of the generated code w.r.t. increasing model sizes measured in SCG nodes. As explained in Section 5.2, the netlist-based approach needs a guard for every BB. Hence, the amount of variables needed increases almost linearly with the nodes in the SCG, which is depicted in Figure 5.5.3a. On the contrary, the priority-based approach needs only to

5.5. Evaluation of Compilation Approaches

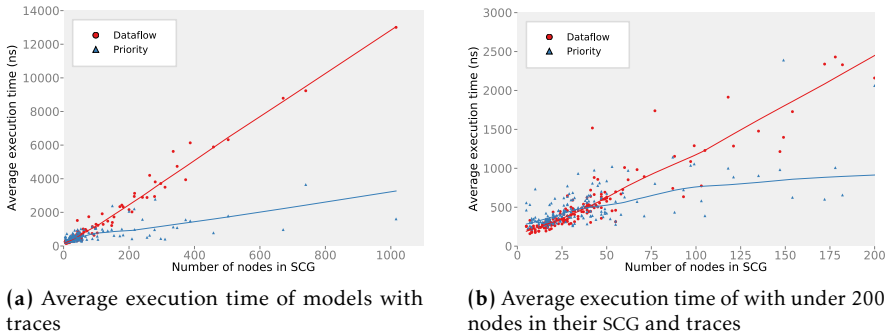


Figure 5.5.4. Comparison of execution times according to Peiler (from [Pei17])

generate variables for inputs and outputs. The macros used in the C version of the approach comprise a fixed amount of up to 50 additional variables. Peiler measured that the latter approach needs less variables on average than the former one in programs with more than 100 nodes in the SCG. Figure 5.5.3b shows the sizes of compiled C programs. As discussed earlier, the size of the code in the netlist-based approach is bound linearly to the number of nodes in the SCG of the netlist of the model, whereas the priority-based approach follows the structure of one-to-one translation of the model to the SCG. Once the overhead produced due the macros of SCL_p is overcome, the latter performs better w.r.t. to code sizes. Peiler measured that this point is reached at around 400 SCG nodes.

With regards to execution time, Peiler measured the tick times of models with execution traces in the KIELER models repository, which range from models with up to 1000 SCG nodes. As shown in the initial SCCharts benchmark, the execution times of the netlist-based approach are bound linearly to the size of the SCG in terms of nodes. The execution times of the priority-based approach depend on the structure of the SCG. While this results in a more erratic jitter, the approach scales better for larger models. Figure 5.5.4 shows the results. Figure 5.5.4a shows the benchmarks for all used models. To take a closer look to the turning point of the two approaches, Figure 5.5.4b zooms in on the small-to-mid size models. As shown in the figure, Peiler measured that the priority-based approach performs better than the netlist-based

5. Interactive Compilation for SCCharts

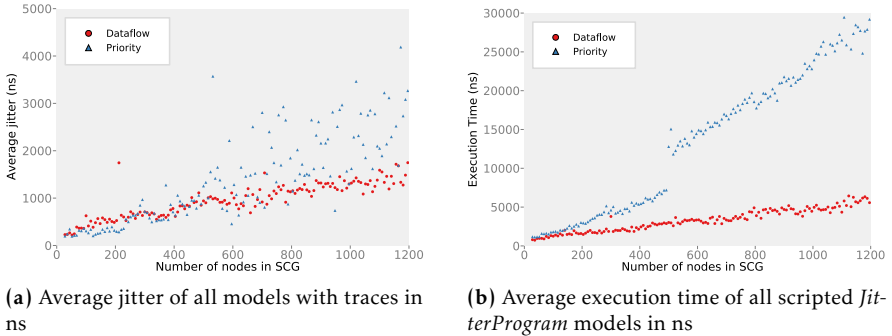


Figure 5.5.5. Jitter comparison according to Peiler (from [Pei17])

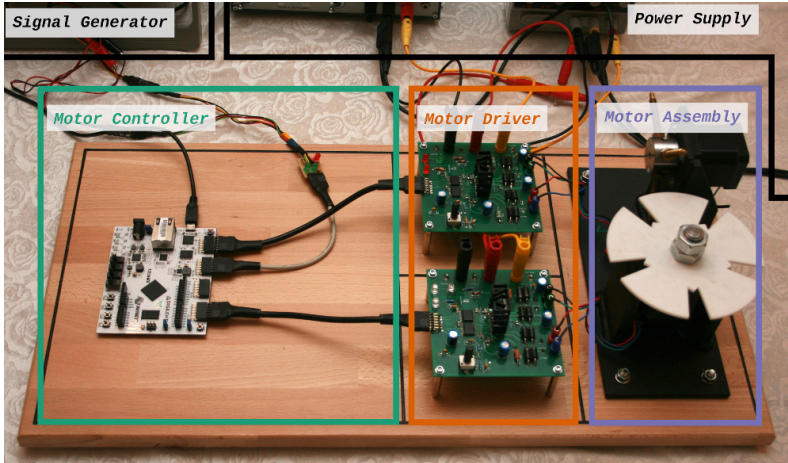
approach w.r.t. execution times at model sizes at around 40 SCG nodes.

Figure 5.5.5 shows the jitter comparison. The results over all models, see Figure 5.5.5a, confirm the results of the initial contribution: the netlist-based approach leads to a steadier execution time jitter. Peiler also performed a benchmark on a set of models which focus on the fork and join overhead of the priority-based approach. These models, named *JitterPrograms*, comprise a high amount of fork and join nodes, which introduces overhead in the approach. Figure 5.5.5b shows that these models performed strictly worse than in the netlist-based approach. Additionally, one can see a significant increase of roughly $5\mu\text{s}$ in the execution time at around 500 SCG nodes. This is the point where the macros of the priority-based approach for C code switches from scalar-based priority bookkeeping to array-based.

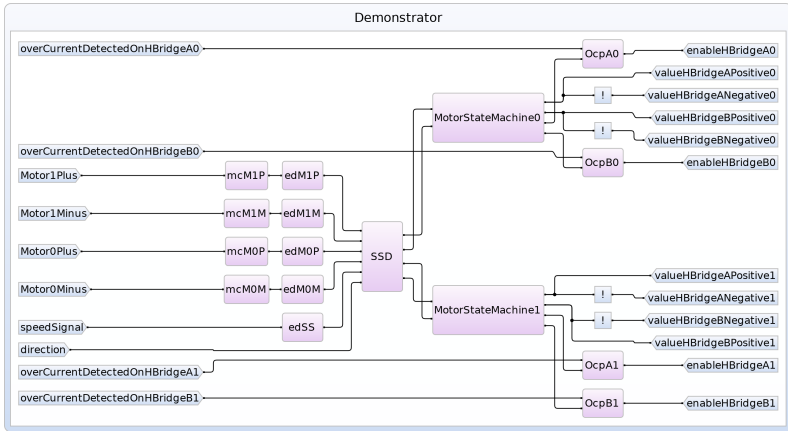
5.5.3 Comparison According to Boysen

To investigate and demonstrate Timed SCCharts [SHM+18a], Boysen built a Disk-and-Sticks demonstrator [BSH20b], which uses two stepper motors. The first rotates a disk with slots and the second rotates three sticks mounted on a shaft. Both motor assemblies are arranged perpendicular towards each other. The rotation ratio of the motors must exactly be 3-to-5 in order to the sticks can pass through the slots of the disk. The timing of the motors is paramount. Any deviation would cause a collision between the assemblies.

5.5. Evaluation of Compilation Approaches



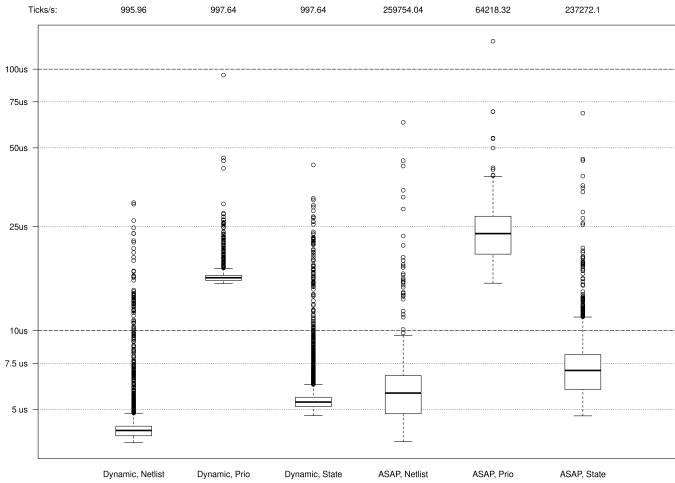
(a) Disk-and-Sticks demonstrator



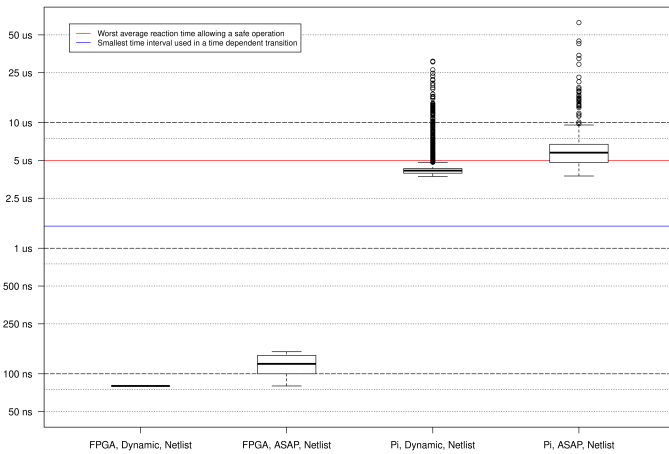
(b) Top-level SCChart controller connecting multiple SCCharts modules handling sub tasks

Figure 5.5.6. The Disk-and-Sticks demonstrator (from [BSH20b])

5. Interactive Compilation for SCCharts



(a) Comparison between different compilation approaches and environment types on a Raspberry Pi (logarithmic scale)



(b) Comparison between FPGA and Raspberry Pi (logarithmic scale)

Figure 5.5.7. Evaluation of the Disk-and-Sticks demonstrator (from [BSH20b])

5.5. Evaluation of Compilation Approaches

The demonstrator is depicted in Figure 5.5.6a. On the top level, the controller, shown in Figure 5.5.6b, was modelled in SCCharts using the dataflow extension, which is discussed in Section 6.2.

While the main goal for Boysen's evaluation was a benchmark for the dynamic tick environment of SCCharts, which is not part of this work, the evaluation also included a comparison between the different compilation approaches when generating code for an actual hardware platform. Two different controller boards for the motors were used for this comparison. The first design was based on a Raspberry Pi, which is capable to run any C code produced by the three SCCharts compilation approaches. The second design was based on an Field Programmable Gate Array (FPGA) board and uses Very High Speed Integrated Circuit Hardware Description Language (VHDL) code generated from the netlist-based approach. Figure 5.5.7a contains the boxplots and the reaction times regarding the performance of Raspberry Pi based controller. The reaction times are measured on the actual target hardware. The experiment includes dynamic tick execution [SHM+18b], which means that the tick function is only executed at the next mandatory motor controller calculation. By calling the tick function as soon as possible, the calculation may not be completed on the next required step w.r.t. motor control.

The expected number of dynamic ticks per second for this setup is approximately 1000, which is reached by all controllers. There is a significant performance gap between the approaches. The netlist-based approach performs better for models of this size and is also a match for code that will be deployed to hardware. However, the state-based lean approach did also perform comparably well. In comparison, Figure 5.5.7b shows the measurements of the netlist-based approach deployed to the FPGA. The outliers, which mostly were contributed to interferences of the Pi kernel, are gone. Furthermore, dynamic tick time execution eliminates the jitter and gives predictable execution times on hardware. With the FPGA-based controller the demonstrator operated successfully with up to 100 sticks/disks crossings per second, which corresponds to 1200 RPM for the disk and 80,000 ticks per second.

5. Interactive Compilation for SCCharts

Lines of code	Netlist	Prio	State w/o	State w/
Header	34	39	55	129
Source	52	104	154	226
Overall	86	143	209	355

Table 5.5.1. Lines of code for the different trials: netlist-based, priority-based, state-based without comments and state-based with comments

5.5.4 Evaluation of the State-Based Approach

The goal of the state-based approach, presented in Section 5.4, was to create a well-readable code, which should ease manual verification. To validate this, two user studies were conducted which compare the code generation to the other approaches. In the first study, the participants should reconstruct the state machine from the generated codes, explained further in Section 5.5.4. Results of the study are presented afterwards followed by executable sizes and execution times. In the second study, a faulty code, generated for a model of a steam-boiler control system, should be fixed. The results are shown in Section 5.5.4.

Manual Reverse Engineering of the State-based Approach

24 students participated in the study. All students were given a short 5 min introduction to the semantics of SCCharts but without information on how the different code generators work or what kind of source they produce. For every code generation approach, the participants should inspect the automatically generated source code without having seen the original source model. They should then draw the model from which the model was generated. The students were asked to draw as many characteristics of the SCCharts as possible, which were states, regions, transitions, the interface, and the labels of the states and regions. There was a time limit of 20 min per trial, although a trial could be finished prematurely. For each trial, the SCCharts were similar to the example shown in Figure 5.4.3. There were four trials which were presented in different orderings to each participant: netlist-based compilation, priority-based compilation, state-based compilation without comments, and state-based compilation with

5.5. Evaluation of Compilation Approaches



(a) Mean time (min) of the state-based trials (b) State-based mean confidence (1=low, 5=high)

Figure 5.5.8. Mean time and confidence of the state-based compilation trials with vs. without comments regardless of trial ordering

auto-generated comments. The different sizes in lines of code, including comments, between the trials can be seen in Table 5.5.1. The overall program code which needed to be understood by the study's participants grows with each trial with 86 lines for the netlist-based, 143 lines for the priority-based, and 209 and resp. 355 lines for the state-based approach. The drawn SCCharts were checked for their correctness. The participant's time needed for the tasks was also measured. The participants were asked to give a confidence rating between 1 (very unsure) to 5 (very sure) for their answer for every trial.

Reverse Engineering Results As Figure 5.5.8 shows, the mean time (Figure 5.5.8a) and the mean confidence (Figure 5.5.8b) of the two state-based trials are nearly identical, even though the participants got the two trials in different order. Hence, for the comparison of the different compilation approaches, it is not differentiated between the state-based approach with or without comments but whichever the participant worked on first (indicated by State I and State II respectively).

Time & Confidence Figure 5.5.9 depicts the overall time and confidence results of all trials. Despite shorter programs w.r.t. lines of code, as shown in Table 5.5.1, the trials with the netlist-based and priority-based generated codes almost always needed the full amount of time, as depicted in Figure 5.5.9a. Even the first state-based trial presented to the participants only needs 14 minutes on average although the code is up to four times

5. Interactive Compilation for SCCharts

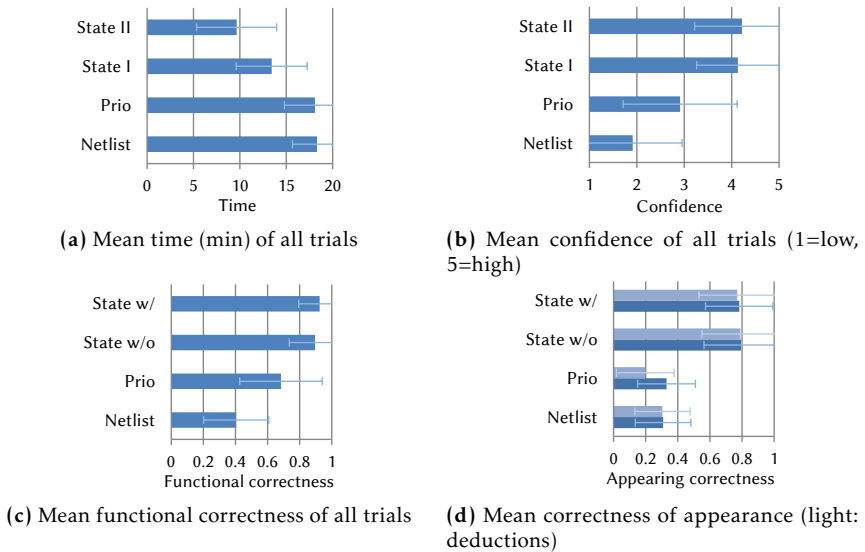


Figure 5.5.9. Mean time and confidence of all trials.

larger. Once accustomed to the structure of the code, the second trial of the state-based code-generation can be done in under 10 minutes mean. It is of course also possible to get a training effect with the netlist- or priority-based compilation, but arguably it is inherently more difficult and with larger models or in some other cases, e. g. , strongly optimized netlists, perhaps even impossible to reach the same result in time and confidence compared to the state-based compilation.

Analogous to the time ratings, the confidence rises with the state-based approach. While the netlist-based code seems to be difficult to understand, the priority-based compilation, being closer bound to the structure of the program, scores better. The state-based compilation, which maps the topology of the statechart nearly one-to-one, always scores a *sure* or better rating on average, even though the participants did not see that kind of code before.

5.5. Evaluation of Compilation Approaches

Correctness As stated at the beginning of this section, the correctness of the reverse engineered SCCharts was also checked. The results are shown in Figure 5.5.9. The ratings are split into *functional correctness* (Figure 5.5.9c), which says if the model behaves semantically correct, and *correctness of appearance* (Figure 5.5.9d), which rates if the model’s graphical appearance resembles the original model. In the case study, this mainly concerns state and region labels. While superfluous syntax, e. g. , transient states, is not per se incorrect, it can impair the overview of the model. Hence, a second rating with deductions for superfluous syntax was added to the results in Figure 5.5.9d.

The functional correctness results almost mirror the confidence ratings. The participant’s models for the netlist-based compilation were 40% correct. For the priority-based compilation they reached 70%, and the state-based compilations were with 90% almost correct without prior knowledge of the SCChart.

The correctness of appearance results are similar. Many of the graphical elements, which include for example labels of regions, states, and variables, are transformed and partially lost during the netlist-based and priority-based compilation. In the second evaluation with the point deductions, the priority-based approach score worse, because the participants drew many transient states which mimic the program control flow which are not necessary and did not resemble the original model even if the model is semantically correct. The topology-preserving state-based compilation keeps most of these elements with their names in the final code and also uses them to auto-generate appropriate comments.

The result suggests that there is some room for improvement even with the state-based approach. The participant’s unfamiliarity to the subject may play a role here, because a commented header is arguably enough to score a 100% correctness of appearance rating.

Code Comments In the trial on the state-based approach with comments, we further asked the participants to rate the influence of the comments towards their drawings (see Figure 5.5.10) from 1 (“I looked only at the comments”) to 5 (“I looked only at the source code”). The results are separated into two groups, depending which state-based study was worked on

5. Interactive Compilation for SCCharts

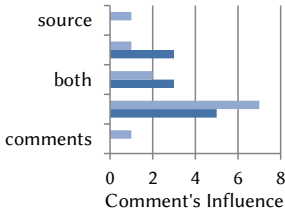


Figure 5.5.10. Comment's influence rating (light: SB with comments first; dark: SB without comments first)

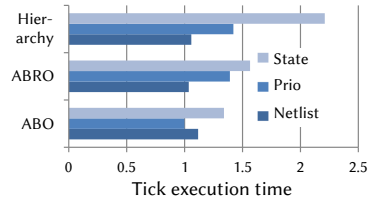


Figure 5.5.11. Mean tick execution time (μs)

first. The group which got the state-based pattern without comments first are depicted in dark blue, whereas the group with the comments first are shown in light blue. While there is a peak in both groups at the “I looked more on the comments” rating, there is not enough data to support that claim yet, especially, as the results in Figure 5.5.8 and Figure 5.5.9 do not show significant distinctions between the two state-based trials. Overall, the study with the comments got a slightly better confidence and functional correctness rating, whereas the appearance rating is slightly worse. However, the survey sample size is too small to conclude from this slight differences.

Run-Time Results Figure 5.5.11 shows a preliminary run-time evaluation for small models. While the state-based approach is only marginally slower at models with little concurrent communication, such as ABO and ABRO, a slow down is measurable for models with more concurrency, which was measured in Hierarchy with five communicating concurrent regions, due to the traversal through the call stack. This slow-down, however, is only present in the state-based version with bi-directional communication, because the control must given back to the parent hierarchy at priority change. State-based lean does not suffer from this phenomenon but only supports uni-directional thread communication.

Manual Verification of the State-based Approach

The steam boiler example is a well known model for cyber-physical systems, which has been implemented in several different languages [Abr96; BW96]. An SCCharts version of the core steam boiler part of that specification can be seen in Figure 5.5.12. The inner behaviour of the normal, degraded, and rescue state is depicted in Figure 5.5.13a.

The steam boiler has four pumps which provide it with water. Each pump can be operated independently. In this example however, they are always activated or deactivated at the same time to simplify the case-study. A water measuring system, which is not modelled here, calculates the quantity q of water which is currently in the steam boiler. The amount of water should be between the minimal normal quantity $N1$ and the maximal normal quantity $N2$. The water level has to be between the minimal limit quantity $M1$ and the maximal limit quantity $M2$ to prevent damage to the steam boiler. $M1 < N1 < N2 < M2$ must hold. To further simplify the generated code, boolean variables are used as signals to notify the system's components about external state changes, instead of broadcasting signals as in the original specification.

The behaviour of the steam boiler can be expressed in five different execution modes:

- ▷ The initialization mode ensure that the steam boiler is filled with an amount of water q which is within its normal quantity ($N1 \leq q \leq N2$). In this mode, the boiler is filled or water is released by opening the valve when necessary. If all pumps work correctly, the initialization phase transitions in the normal execution phase. If they do not, the degraded state is activated.
- ▷ During normal execution, the steam boiler should only have a water level q between its normal, non safety-critical boundaries $N1$ and $N2$. The initial state in this mode assumes a valid water level (see initialization mode). Depending whether the water level falls below or rises above this range, pumps are activated or deactivated and a corresponding inner state is entered. For example, we take the transition from `manageWaterLevel` to `tooLittleWater` if $q < N1$. The inner state is left again if the water-level measured by the quantity q is within the normal range. The normal mode

5. Interactive Compilation for SCCharts

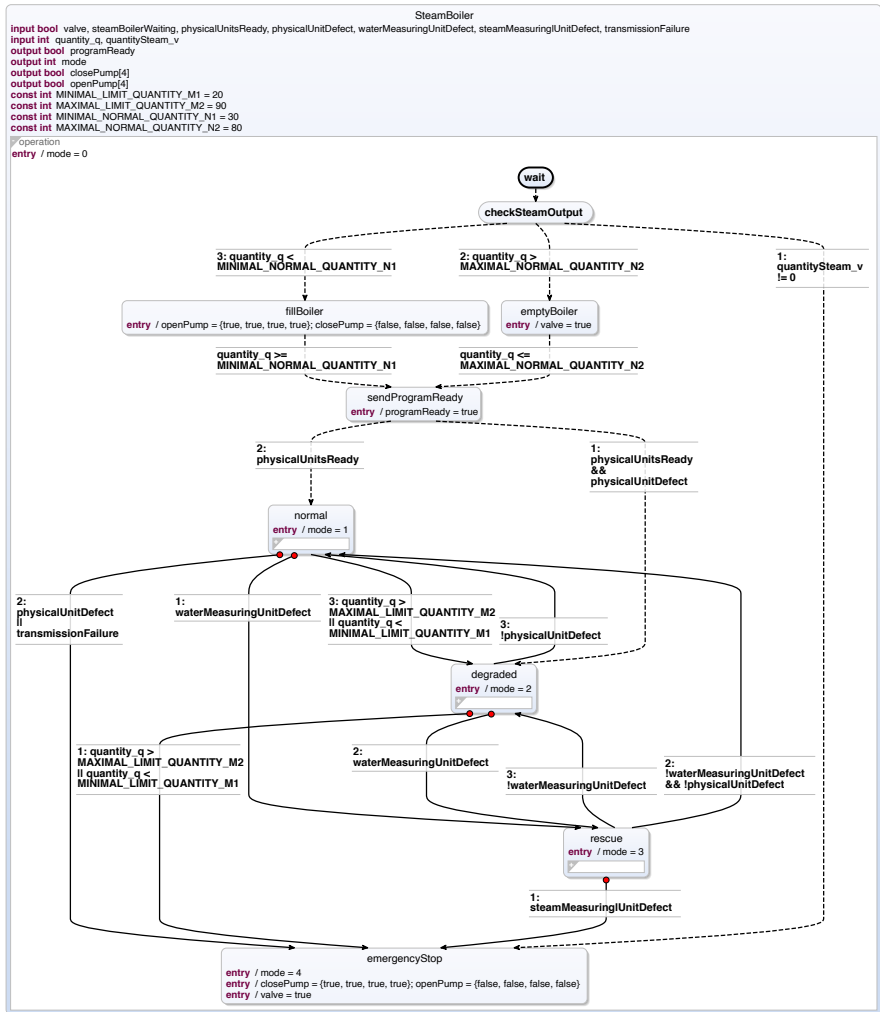


Figure 5.5.12. Steam boiler core model based on the specification of Abriall [Abr96] implemented in SCCharts

5.5. Evaluation of Compilation Approaches

is left if parts of the pumping-system fail (degraded), the water measuring unit is defect (rescue), or the water level is no longer within its critical boundaries (emergency stop).

- ▷ In the degraded mode the system has the same behavior as in the normal mode as originally specified. The system enters the degraded state if a physical unit, for example the pumping-system, fails. The degraded state is left if the failed parts are repaired (normal), a water measuring unit defect occurs (rescue), or the critical boundaries $M1$ and $M2$ are violated (emergency stop).
- ▷ During rescue mode only the calculation of q changes. The inner behavior is the same as in the normal and the degraded mode. In the rescue state the quantity q is calculated using v , the quantity of produced steam. This state is left if the corresponding failed systems are repaired (normal, degraded) or a defect of the steam measuring unit is detected (emergency stop).
- ▷ The emergency stop mode stops the total system if it is no longer rescueable. The original paper specifies that the physical environment has to take appropriate actions to shut down the system. We model this by closing the pumps and releasing all water from the boiler by opening the valve.

The inner behaviour of the normal, degraded, and rescue modes are modelled identically to make the task of identifying specific inner states non-trivial. This is also stated by the original specification.

Case-Study Results The steam boiler case-study verifies if the state-based code generation approach makes it easier to comprehend a program. The diagram of the steam boiler model, as it is depicted in Figure 5.5.12, was given to the participants of the study in digital form as an SVG and also as A4 print-out. They received a five minute introduction to the power plant set-up and a general explanation on the netlist-based, priority-based and state-based code generation approach. Afterwards, the participants were asked to find a structural error within the generated C codes. A structural error means that there is either

- ▷ an additional erroneous or missing state,

5. Interactive Compilation for SCCharts

- ▷ an additional erroneous or missing transition,
- ▷ or a transition with wrong source or target state.

They were further informed that the wrong behaviour had been located within the degraded state. The degraded state was marked in red to signal that the issue with the boiler was located inside this superstate. It was not necessary to check outgoing transitions of the degraded superstate. The participants had 12 minutes to find the difference of the generated C code and the steam boiler model for each code generation approach. However, if they could not make any sense of the generated code, they were allowed to end the approach prematurely. To stay within the tight time constraints, each approach comprises exactly one structural error. The included errors are depicted as SCCharts in Figure 5.5.13. Figure 5.5.13a shows the original inner behaviour of the degraded state. The participants faced erroneous code of the following model alterations: In the first structurally erroneous part (1) of the diagram, the transition from `manageWaterLevel` to `tooLittleWater` is missing, as seen in Figure 5.5.13b. In the second faulty model part (2) in Figure 5.5.13c, the transition from `tooLittleWater` to `manageWaterLevel` has the wrong target `tooLittleWater`. The third erroneous diagram (3), shown in Figure 5.5.13d, has a new state `pumpsOff`. The `pumpsOff` state is always reached from `manageWaterLevel` since the transition to it has the highest priority and it is not guarded. Note that depending on the code generation approach, code optimization might throw away unreachable code. For example, error (1) (see Figure 5.5.13b) results in a missing `tooLittleWater` state in the netlist-based code generation approach.

The participants were put into six different groups regarding the code generation approach they worked on first, as seen in Table 5.5.2, to mitigate learning effects. Each group should include a reasonable number of people.

	I	II	III
Group 1	netlist	prio	state-based
Group 2	prio	netlist	state-based
Group 3	state-based	netlist	prio
Group 4	prio	state-based	netlist
Group 5	state-based	prio	netlist
Group 6	netlist	state-based	prio

Table 5.5.2. Order of code generation approaches for each group

5.5. Evaluation of Compilation Approaches

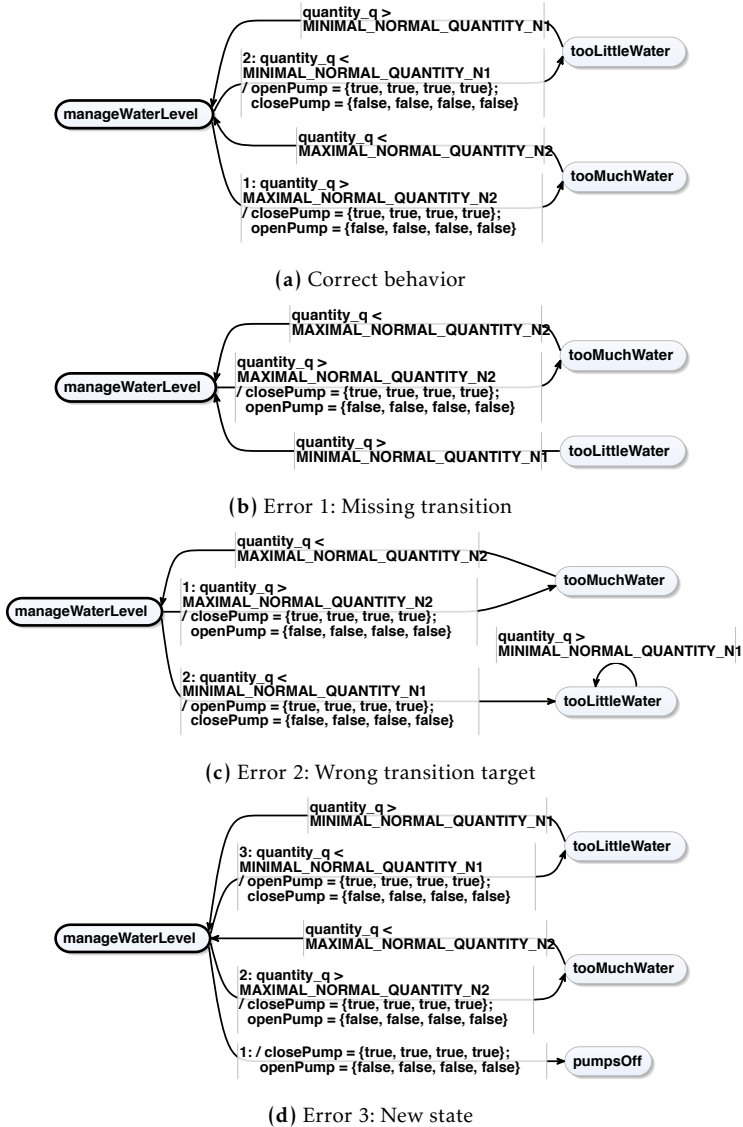


Figure 5.5.13. Inner behaviour of the degraded state

5. Interactive Compilation for SCCharts

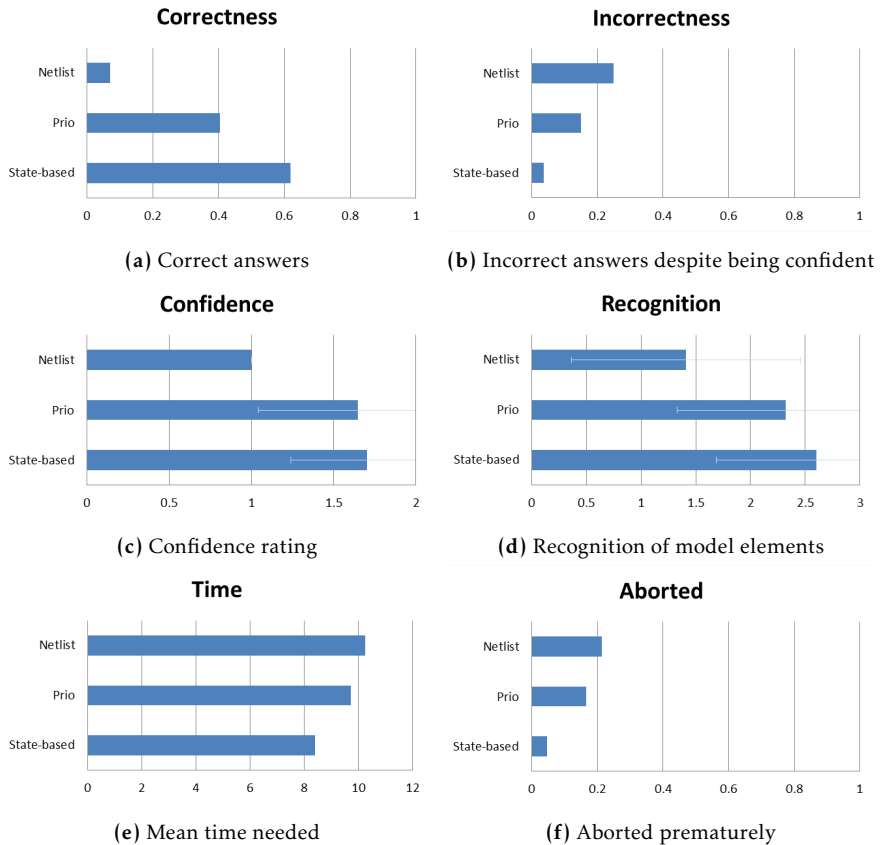


Figure 5.5.14. Steam boiler case-study results

42 students participated; all of them computer science students, either in the final terms of their bachelor's or in the master's degree programme. While having limited experience with SCCharts from attended lectures, they had only sparse knowledge about the different code generation approaches. Figure 5.5.14 shows the results of the case-study in the categories *correctness*, *confidence*, and *time*.

5.5. Evaluation of Compilation Approaches

Correctness From the 42 participants, 61.9% found the issue in the state-based approach, 40.4% in the priority-based approach, and 7.1% in the netlist-based approach, which is depicted in Figure 5.5.14a. Moreover, only 4, 20, and 27 participants named an issue within the netlist-, priority-, and state-based approaches respectively. In this order, 1, 3, and 1 answers were wrong, which is shown in Figure 5.5.14b normalized to the answers given.

Confidence The participants should rate how confident they are with their answer on a scale from 0 (not confident) to 2 (confident). From the correct answers, the mean result of the confidence rating is depicted in Figure 5.5.14c. While both, priority and state-based, are settled around a mean value of 1.7, the confidence in the netlist approach is lower at around 1.

If the participants did not find the issue, they were asked if they recognize any relevant parts of the diagram in the code, namely states, regions, or transitions relevant for the degraded state. The mean values of these answers are shown in Figure 5.5.14d, counting the types of parts they recognized from 0 to 3. The results are 2.6, 2.32, and 1.41 for state-based, prio, and netlist respectively.

Time The mean time values for the participants which gave correct answers are shown in Figure 5.5.14e, which are 10:15 minutes for netlist, 09:44 minutes for prio, and 08:24 minutes for state-based. In this order of approaches, 9, 7, and 2 participants ended their try prematurely, because they could not make any sense of the generated code, which is shown in Figure 5.5.14f normalized to the total number of participants.

Freeform Feedback All participants who gave feedback to the netlist approach state that it is not readable and not made to be read by humans. The priority approach was criticized since one has to read all code leading to the degraded state to find it. Moreover, the presence of many conditionals in this approach impairs the readability. The state-based approach was seen as the most readable. However, one has to focus on the code relevant for the task, because of the increased code size. The state-based approach result in 1271 lines of generated code compared to 390 and 316 for priority and netlist (without header). The long variable names were also criticized: They

5. Interactive Compilation for SCCharts

proved a challenge for one participant with dyslexia, because they often have seemingly redundant names as a result of the transformation, such as "...regionoperation_stateoperation_regionoperation...".

Results Overall, the results confirm the trends from the first case-study. The code generated by the state-based approach is more readable and mappable to the original model. This facilitates manual verification of the generated code, increasing the rate of issues found in less time. While it is arguably not surprising that manually verifying netlists is difficult, the state-based approach also scores better in this field compared to the priority-based approach despite the fact which both generate structures that resemble the original model. The priority-based approach uses nested macros for concurrent regions and jumps to labels to model transitions and states. The state-based approach follows the object-oriented state pattern but creates dedicated functions for regions and states, which mimic the hierarchy of an SCChart. The current state is stored in a context object and is updated at state transitions. Also, the priority-based approach sometimes gave a false sense of confidence, which was less of a problem with the state-based approach.

5.5.5 Run-Time Evaluation of All Approaches

The data of the previously presented evaluations were gathered since 2013. Unsurprisingly, the means of gathering changed over time from hard-coded time measurements inside the generated code to more generic benchmark templates later on. In the final run-time evaluation of all presented approaches, the latest version of KiCo was available. The results are a benchmark of the compilation approaches but serve also as demonstrator for the potential of the interactive model-based compilation approach and its KiCo implementation: the complete benchmark can be configured on-demand.

Figure 5.5.15 shows the KiCo workflow for the benchmark suite. Different KiCo contexts are surrounded by a dotted box. The complete benchmark is configured as a system which runs a series of processors (blue boxes). The root context receives the benchmark configuration and then collects model files, e. g. from a model repository. The benchmark generator then

5.5. Evaluation of Compilation Approaches

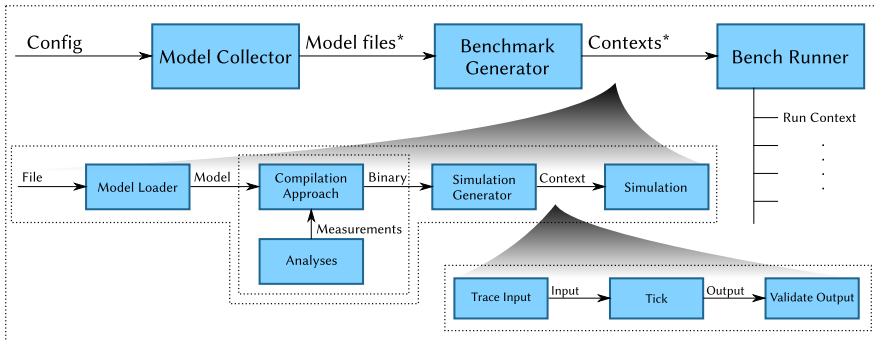


Figure 5.5.15. Schematic of the test suite benchmark compilation system

creates benchmark compilation contexts for each model file according to its configuration. Finally, the contexts are executed and the results are gathered in a generic format, e. g. JSON. Therefore, in this evaluation, the KiCo system requires a JSON configuration as input and returns the complete benchmark of all configured runs also in JSON format.

The distinct benchmark contexts proceed as follows. They retrieve the model file from which the actual model is parsed. The model is fed into the system for the selected compilation chain. The chain can be enriched by the analysers required for the concrete benchmark. As usual, a binary is created, i. e. by sending the generated model code to an external compiler. The binary is then used to drive the simulation in KIELER as an own new context, which is generated by a dedicated processor. The data gathered during simulation is then written back into the environments of the parent context, which are saved as part as the benchmark results. During simulation, the trace files are checked automatically and mismatches are reported in the results.

The simulation context consists of three processors. The first one reads the inputs from the current trace file and sends them to the model. In the second processor, a single tick of the model is then executed. The third processor checks the outputs of the model against the trace file. The results are written into the environment.

5. Interactive Compilation for SCCharts

```
1 {"de.cau.cs.kieler.simulation.testing.benchmark.runner.json.verbose": true,
2  "de.cau.cs.kieler.simulation.testing.suite.config": {
3    "all": {
4      "env": {
5        "de.cau.cs.kieler.simulation.testing.runs": 10,
6        "de.cau.cs.kieler.kicool.analyzers.objects.classes": "State,Region,Transition"
7      },
8      "filter": {
9        "model": "sccharts && !(\\\"known-to-fail\\\" || \\\"must-fail\\\")",
10       "trace": "true"
11     }
12   },
13   "sccharts-netlist-c-simulation": {
14     "system": "de.cau.cs.kieler.sccharts.simulation.netlist.c",
15     "analyzer": {
16       "de.cau.cs.kieler.kicool.analyzers.objects": "post:de.cau.cs.kieler.sccharts.processors.reference",
17       "de.cau.cs.kieler.kicool.analyzers.loc": "post:de.cau.cs.kieler.scg.processors.codegen.c"
18     }
19   },
20   "sccharts-prio-c-simulation": {
21     "system": "de.cau.cs.kieler.sccharts.simulation.priority.c",
22     "analyzer": {
23       "de.cau.cs.kieler.kicool.analyzers.objects": "post:de.cau.cs.kieler.sccharts.processors.reference",
24       "de.cau.cs.kieler.kicool.analyzers.loc": "post:de.cau.cs.kieler.scg.processors.codegen.prio.c"
25     }
26   },
27   "sccharts-statebased-lean-c-simulation": {
28     "system": "de.cau.cs.kieler.sccharts.simulation.statebased.lean.c",
29     "analyzer": {
30       "de.cau.cs.kieler.kicool.analyzers.objects": "post:de.cau.cs.kieler.sccharts.processors.reference",
31       "de.cau.cs.kieler.kicool.analyzers.loc":
32         "post:de.cau.cs.kieler.sccharts.processors.codegen.statebased.lean.c"
33     }
34   },
35   "sccharts-statebased-lean-cs-c-simulation": {
36     "system": "de.cau.cs.kieler.sccharts.simulation.statebased.lean.c",
37     "analyzer": {
38       "de.cau.cs.kieler.kicool.analyzers.objects": "post:de.cau.cs.kieler.sccharts.processors.reference",
39       "de.cau.cs.kieler.kicool.analyzers.loc":
40         "post:de.cau.cs.kieler.sccharts.processors.codegen.statebased.lean.c"
41     }
42   },
43   "sccharts-statebased-c-simulation": {
44     "system": "de.cau.cs.kieler.sccharts.simulation.statebased.lean.c",
45     "analyzer": {
46       "de.cau.cs.kieler.kicool.analyzers.objects": "post:de.cau.cs.kieler.sccharts.processors.reference",
47       "de.cau.cs.kieler.kicool.analyzers.loc":
48         "post:de.cau.cs.kieler.sccharts.processors.codegen.statebased.lean.c"
49     }
50   }
51 }
52 }
```

Listing 5.5.1. Complete benchmark configuration for the final run-time benchmarks

5.5. Evaluation of Compilation Approaches

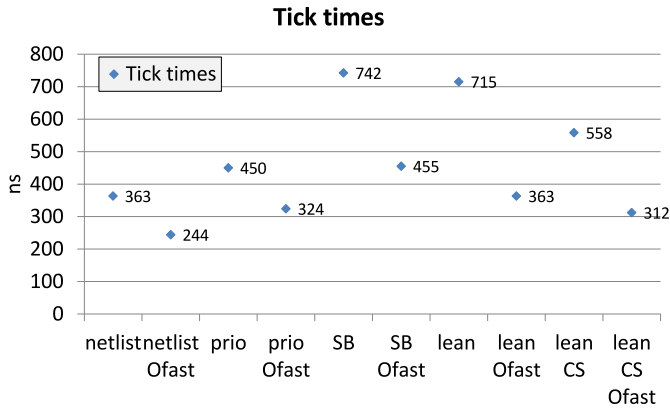
The workflow serves as test suite and is in fact the procedure to test against the KIELER models repository in the current version of KIELER.

The complete benchmark configuration for all five compilation approaches is listed in Listing 5.5.1 in JSON. It is 50 lines long. Environment settings for all benchmarks are situated in the `all` object. The benchmark is configured to test each model 10 times and the object count analyser should search specifically for the classes `State`, `Region` and `Transition`. Afterwards, specific test runs are configured; one for each compilation system. In this benchmark, all systems are configured in the same way, but this is not mandatory. Each run specifies the used compilation system and 2 analysers, which count model objects and lines of generated code. The simulation automatically provides tick times and reports tick mismatches in case of an error. The results are also written back in the JSON format to facilitate further processing. They can be parsed by simple scripts, e. g. in Python, or be converted into formats for common spreadsheet software. The two analysers in this benchmark are generic and can be re-used in other settings. However, customized analysers can be added relatively simple as KiCo processors. Both analysers used here comprise less than 70 LoCs. The tick times have been recorded for two GCC optimization levels, `-O0` and `-Ofast`. The benchmark ran on an Intel(R) Xeon(R) with 16 CPUs (8 cores) 2.53 GHz and 48 GByte RAM running an Ubuntu 18.04.4 LTS – 64 bit.

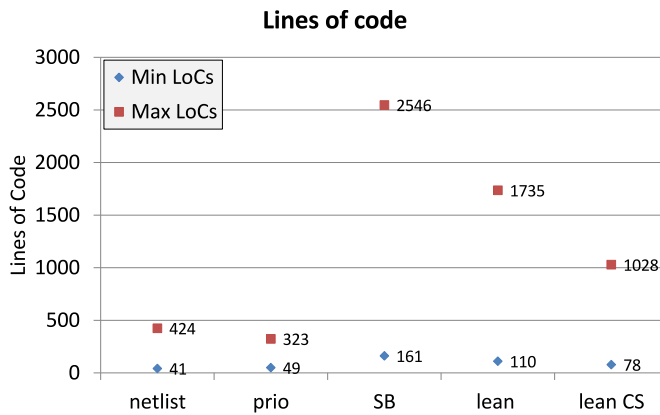
The KIELER models repository contains 115 models which were compatible to all five approaches. The model sizes range from models with 9 to 336 objects. The expanded models contain up to 61 states and up to 24 regions.

Figure 5.5.16a shows the average tick time and the LoCs results of the comparison. The average tick times in Figure 5.5.16a confirm the previous trends for small scale models. The netlist-based compilation performs best, followed by the priority-based approach. As expected, the *fast* optimization decreases the tick times, but the performance gain is larger in the state-based approaches. Here, the lean CS and priority approaches nearly perform equally. However, as explained earlier, the priority approach also supports bidirectional communication and the lean CS approach is optimized towards readability. Regarding size, the state-based approach require significantly more LoCs. For larger models, the priority-based approach needs less lines than the netlist-based approach, which also confirms the previous results.

5. Interactive Compilation for SCCharts



(a) Tick times comparison

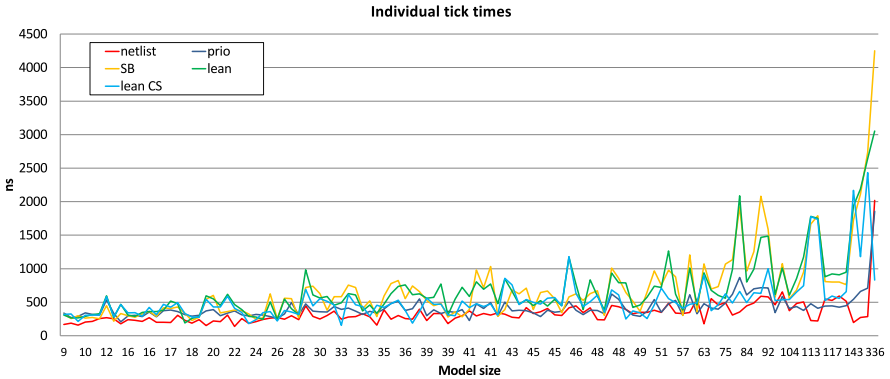


(b) LoCs comparison

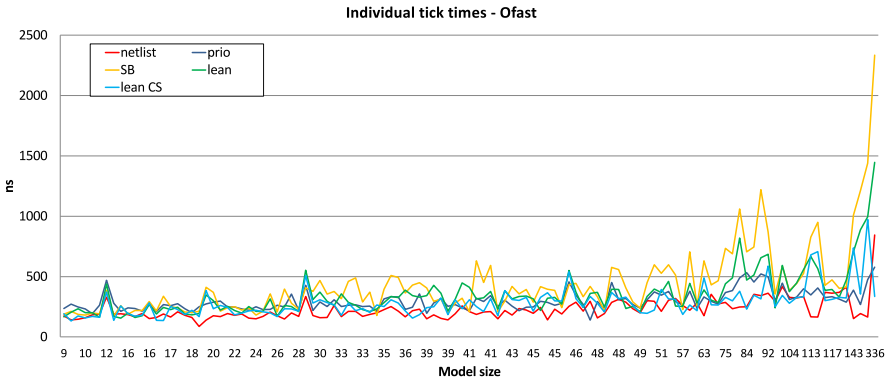
Figure 5.5.16. Comparison between all presented compilation approaches

Figure 5.5.17 shows the timing results for each individual model with (a) and without (b) *fast* optimization. The results also confirm the previous trends. The tick times for netlist-based approach grow steadily w.r.t. to the model size. While the priority-based approach performs a bit worse compared to the netlist-based approach when it comes to small models, it

5.5. Evaluation of Compilation Approaches



(a) Individual tick times comparison



(b) Individual optimized tick times comparison

Figure 5.5.17. Comparison of individual tick times

scales better as the model size grows. Depending on the model structure, it is sometimes faster than then the netlist-based approach.

The state-based approach usually performs a bit worse compared to the former two approaches. In particular, the three spikes at model sizes around (i) 75 – 92, (ii) 113 and (iii) 336 are interesting. The models at (i) include hierarchy and (weak) aborts. Unsurprisingly the lean CS approach performs better than the other two state-based approaches, because the abort

5. Interactive Compilation for SCCharts

feature is supported naturally. It nearly performs as good as the netlist-based compilation.

The (ii) models make use of the SCCharts dataflow extension. The optimizations in the netlist-based compilation approach facilitate immediate during actions used in the SCCharts dataflow extensions, whereas the feature has to be expanded normally in the other approaches. Therefore, the approach performs best. More on the dataflow extension follows in Section 6.2.

Finally, the model in (iii) makes use of *Timed SCCharts*, which are compiled to many concurrent *during actions*. Here, the lean CS approach, which can naturally handle aborts and actions, performs even better than the netlist-based and priority-based approach. Similar to RISC vs. CISC architectures, a specialized instruction set should perform better at tasks it was designed for. The question is if the extra effort justifies the higher compiler complexity. As described earlier in Section 5.4, the abort and action trade-off seems reasonable.

Rapid Prototyping

*We can only see a short distance ahead,
but we can see plenty there that needs to be done.*
— Alan Turing

This chapter presents how the concepts from Part I also facilitate the rapid development of various extensions to the SCCharts language, its underlying MoC and beyond. Section 6.1 introduces the high-level concept of scheduling directives, which influence the scheduling regime of models as first class citizen of the language. They can be seen as a generalization of the IURP, which can be influenced by the modeller. Section 6.2 adds a dataflow extension to the otherwise control-flow-oriented SCCharts flavour. Section 6.3 explains symmetrical entry and exit actions. To conclude the extensions added to SCCharts, Section 6.6 presents the results of a series of SCCharts surveys, which were gathered over a span of five years. Finally, to show possibilities beyond SCCharts, Section 6.4 and Section 6.5 sketch out how to use the interactive model-based compilation approach to enrich classical Esterel by the SCM_oC, which results in the new dialect SCEest, and how to extract arbitrary models out of C programs.

6.1 Scheduling Directives

To reconcile concurrency and determinism for programming reactive systems, synchronous languages follow their strictly defined MoCs. Execution is separated into discrete *ticks*, where (sensor) inputs are read and (actuator)

6. Rapid Prototyping

outputs are produced. Within each tick, concurrent threads of execution proceed according to certain scheduling rules which guarantee determinism defined by the MoC. As previously discussed, the prominent *write-before-read* principle, employed in languages such as Esterel and in dataflow languages, such as Lustre, demands that a write to some variable x must be scheduled before a concurrent read of x .

The write-before-read principle clearly guarantees determinism, but like other scheduling rules, comes at the price that a compiler may reject a program because it cannot find a viable schedule for it, e. g. due to cyclic write-read dependencies. It is then reported that the program is *not causal* and it is the programmer's task to find a solution for the scheduling issue. This, in practice, is often easier said than done for different reasons: 1) Some synchronous MoCs are restrictive in ways that the average programmer may not expect; 2) the compiler's analysis and scheduling abilities may be limited and conservatively reject programs which would indeed be schedulable; and 3), the feedback provided by the compiler may be too restrictive to be helpful to the programmer, in particular when models become more complex. Issues 1) and 2) not only matter for the human developer but also when transforming a model as part of a compilation; restrictive scheduling regimes defined by the MoC may make M2MTs which compile advanced language features into simpler ones more complex than one might hope.

Besides giving the modeller means to understand what is happening and guidance on how to solve the problem, as discussed in Section 4.1, Scheduling Directives (SDs) are a powerful tool to make causality handling more practical. SDs form Flexible Schedules (FSs) for synchronous languages. These should not replace existing scheduling regimes but rather augment them either to change the default scheduling or to make program schedulable (causal) in the first place. This approach should not replace existing solutions for solving causality issues, such as *pre*, but adds another tool to the repertoire of the modeller. M2MTs transformations can benefit from SDs in the same way without the modeller having to interact.

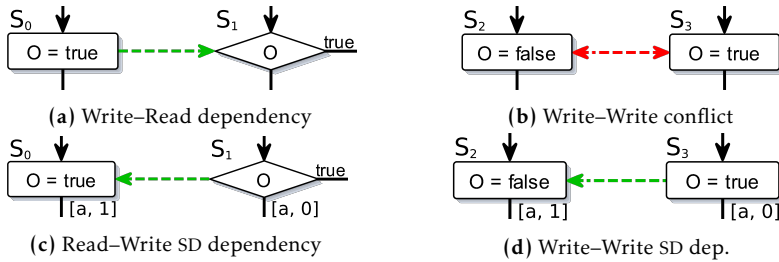


Figure 6.1.1. Dependencies induced by either a MoC or an SD

6.1.1 Flexible Schedules

Accesses to variables are usually categorized into *writers* and *readers*. A possible control-flow graph representation, as depicted in Figure 6.1.1, shows assignment statements (rectangle nodes) and conditional statements (diamond nodes). A *schedule* is a static order of all nodes in a control-flow graph, meaning the order is determined at compile-time and fixed during run-time. The particular ordering is governed by the used MoC. Usually, it is determined by the *control* and/or (concurrent) *data dependencies*.

Figure 6.1.1 shows four examples of concurrent data dependencies. In Figure 6.1.1a, as usual a write-before-read dependency is depicted as green dashed arrow. An exemplary relation for these statements is $s_0 \rightarrow_{moc} s_1$, with \rightarrow_{moc} being an order relation which implements the rules of the underlying MoC (s_0 before s_1). Figure 6.1.1b shows two conflicting write accesses. The dependency conflict is depicted as red dashed double arrow.

A *scheduling directive* (SD) associates a *scheduling unit* with a *named schedule* and an *index*.

The scheduling unit may be for example a single statement, or a coarser unit of execution, such as a thread. For a named schedule s , the scheduling units associated with s must be scheduled according to their index, lowest index first.

For example, considering Figure 6.1.1c, an SD may be added to each of the scheduling units (statements) s_0 and s_1 which associates them with schedule a and indices 1 and 0, respectively. This induces a scheduling order $s_1 \rightarrow_{sd} s_0$.

6. Rapid Prototyping

The value of O is now read from in s_1 , before written to in s_0 . Analogously, the write–write conflict is resolved in Figure 6.1.1d by giving statement s_3 a lower index than statement s_2 . If an SD exists for two statements, the SD order (\rightarrow_{sd}) is used. Otherwise, the MoC determines the order (\rightarrow_{moc}).

Definition A *flexible schedule* (FS) is a schedule which takes all SDs of the model into account.

Remark The term *priority* is avoided to avert confusion with the priorities of priority-based scheduling, where the highest priority is executed first.

When a conflict occurs which leads to an incomplete model, the modeller can complete it with SDs. They can be used *directly* on different levels of detail or *indirectly* via M2MTs. Both are explained in the following.

Conclusion A model containing scheduling conflicts is not considered to be causally wrong per se but merely incomplete.

Example To exemplify modelling with SDs in the SCCharts, Figure 6.1.2b shows a diagram of an SCCharts model, named Counter Reset. The textual source program is shown in Figure 6.1.2a. The scheduling protocol, i. e. the IURP, states that concurrent accesses within one tick can only set, update and read variables in this particular order as indicated by the coloured, dashed dependencies in Figure 6.1.3. The example contains the dependency cycle

$$\text{counter} = 0 \rightarrow_{moc} \text{counter}++ \rightarrow_{moc} \text{counter} \geq 10 \rightarrow_{moc} \text{counter} = 0.$$

Therefore, under the used MoC, similar to other synchronous MoCs, this model is considered not causal and rejected.

Example Despite the default MoC of SCCharts, the modeller can use SDs to model named schedules to achieve the intended behaviour. To illustrate, consider Figure 6.1.4a, which is the Counter Reset example from Figure 6.1.2 enriched with SDs. First, a named schedule `_auto` is declared, in line 3. Named schedules, which are of the form $\langle \text{scheduling unit} \rangle \text{ schedule } \langle \text{schedule name} \rangle \langle \text{index} \rangle$. In Figure 6.1.4a, the SDs in lines 8 and 19 resolve the cycle by incrementing the counter before the test and reset.

It may be difficult for a modeller to obtain an overview over all conflicts and subsequent potential cures for these conflicts. Therefore, instead of letting the modeller add SDs textually, interacting directly with the diagram is sometimes a more efficient approach to define SDs. An example is

6.1. Scheduling Directives

```

1 scchart CounterReset {
2   output int counter = 0
3
4   region Increment:
5     initial state Wait
6     do counter++
7     go to Increment
8
9   state Increment
10    immediate go to Wait

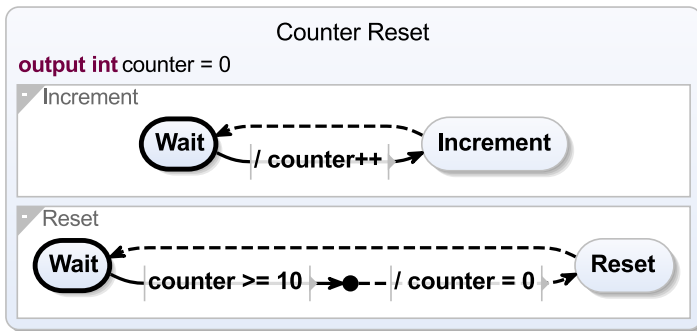
```

```

10 region Reset:
11  initial state Wait
12  if counter >= 10 go to Do
13
14  connector state Do
15  immediate do counter = 0
16  go to Reset
17
18  state Reset
19  immediate go to Wait
20 }

```

(a) Textual representation of Counter Reset



(b) Automatically generated graphical representation of Counter Reset

Figure 6.1.2. Concurrent Counter Reset program in SCCharts

the dependency view discussed in Section 4.1. The modeller can click on the blue dependency edge in the example to reverse its direction. The SD overriding the MoC is added to the model and therefore also to the textual description without the modeller having to find and edit the corresponding text manually.

Scheduling Transitivity Since it is possible to declare arbitrary many schedules, it is also possible to declare a schedule for any number of concurrent expressions. However, when considering user-defined schedules, it is sufficient to use exactly one schedule per superstate due to the transitive

6. Rapid Prototyping

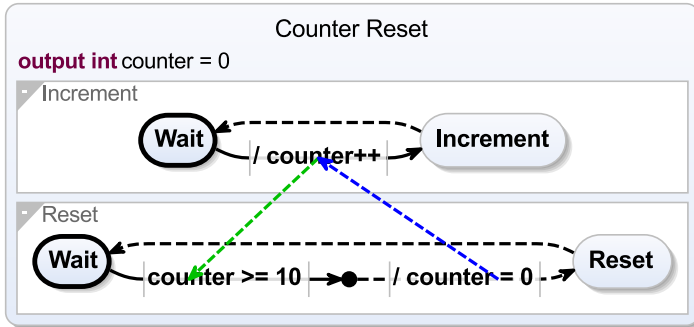


Figure 6.1.3. Data dependencies are visualized as coloured, dashed edges between the regions.

nature of the scheduling.

Example Figure 6.1.5 shows a more complex model with the three concurrent regions T, F, and Q. Let $_autoTF$, $_autoTQ$, and $_autoFQ$ be different, named schedules between the regions T, F, and Q. The scheduling constraints are then given by

$$_autoTQ : T0 \rightarrow Q0 \rightarrow Q1$$

$$_autoTF : T0 \rightarrow F0$$

$$_autoFQ : Q0 \rightarrow F0 \rightarrow Q1.$$

However, following the transitive principle, the modeller can enforce the schedule $T0 \rightarrow Q0 \rightarrow F0 \rightarrow Q1$ with one scheduling definition, which is named $_auto$ in the example.

When looking at individual dependencies, it is still possible to create a *dependency cycle*, which is not schedulable, e. g.

$$T0 \rightarrow Q0 \rightarrow F0 \rightarrow T0.$$

However, as concurrent conflicts are solved by prioritizing conflicting expressions within a single schedule, the previously mentioned scenario cannot be expressed in the SCL. It is still possible to create an unschedulable program if the control flow disallows a dependency schedule. For example, the schedule

$$Q1 \rightarrow T0 \rightarrow F0 \rightarrow Q0$$

6.1. Scheduling Directives

```

1 scchart CounterReset {
2   output int counter = 0
3   schedule _auto
4
5   region Increment:
6   initial state Wait
7   do counter++
8   schedule _auto 0
9   go to Increment
10
11  state Increment
12  immediate go to Wait

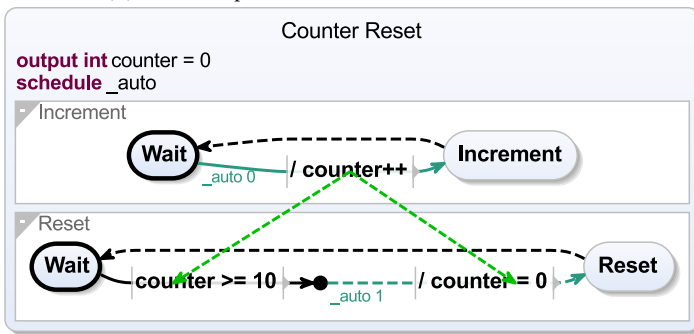
```

```

13 region Reset:
14 initial state Wait
15 if counter >= 10 go to Do
16
17 connector state Do
18 immediate do counter = 0
19 schedule _auto 1
20 go to Reset
21
22 state Reset
23 immediate go to Wait
24 }

```

(a) Textual representation of Counter Reset with SDs



(b) Automatically generated graphical representation of Counter Reset; the dependency edges are now influenced by the SDs.

Figure 6.1.4. Counter Reset example with SDs

is expressible but not schedulable since the control flow from Q0 to Q1 makes the schedule infeasible.

Scheduling Directives on Coarser Granularities It is often sufficient to define SDs on a coarser granularity than the statement level. Especially when the language supports a distinction between core (resp. kernel) and extended features, coarser granularities are implemented easily. If statement-level SDs are available in the core language, coarse granularity SDs can be implemented

6. Rapid Prototyping

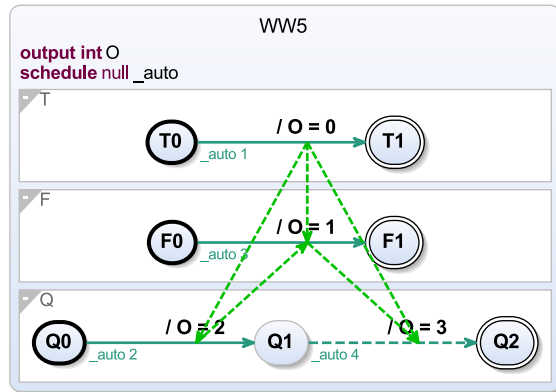


Figure 6.1.5. Example WW5 depicts a strict order across three regions.

as extended features, which can be transformed automatically to statement-level SDs via model-based compilation.

Example For example, using a schedule on a *regions* sets the directive for all statements in that region. Therefore, a modeller can relatively easy define that a specific region should be scheduled before another one.

Example Similarly, using a schedule on a transition or an action assigns the SD to all statements of the transition/action. Note that this precludes interleaved execution of the transitions.

6.1.2 Scheduling Directives in Transformations

As discussed in Part I, consecutively executed M2MTs are the core of a model-based compiler. Even if the modeller does not use SDs directly, they can improve these transformations w.r.t. complexity and efficiency.

One M2MT in the SCCharts compiler transforms the *count delay* feature into simpler constructs. In a graphical syn-

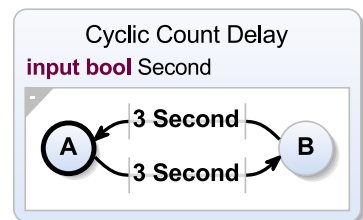


Figure 6.1.6. Cyclic count delay

6.1. Scheduling Directives

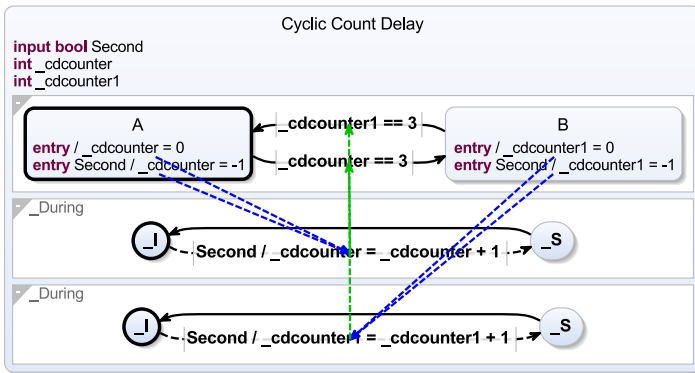


Figure 6.1.7. Expanded cyclic count delay

tax, count delay is depicted as an integer n in front of a transition trigger. Such a transition is only taken if it was eligible to run n times without the count delay. An example of two alternating count delays can be seen in Figure 6.1.6.

A straightforward transformation which simply counts the occurrences as implemented by Motika [Mot17] adds a counter per count delay and waits until n is reached. This works for simple count delays. However, if two count delays are called in a cyclic manner as in Figure 6.1.6, this simple approach fails because of cyclic dependencies which are introduced by the M2MT, see Figure 6.1.7, similar to the pattern shown in Section 6.1.1.

The current version of the SCCharts compiler solves this problem by using a more sophisticated transformation which uses *pre* operators to look at values of from the previous ticks, which is a common way for solving causality problems in synchronous languages. However, since the increments should always be performed before the test and reset, this transformation can be made more efficiently with SDs similar to the counter example presented in Section 6.1.1. It is sufficient to set the scheduling index of the counting regions to a lower value than the index of the main region. As a result, the SDs ensure that the increments happen before the checks and potential resets of the counters, see Figure 6.1.8. Additionally, an arguably unintuitive reset

6. Rapid Prototyping

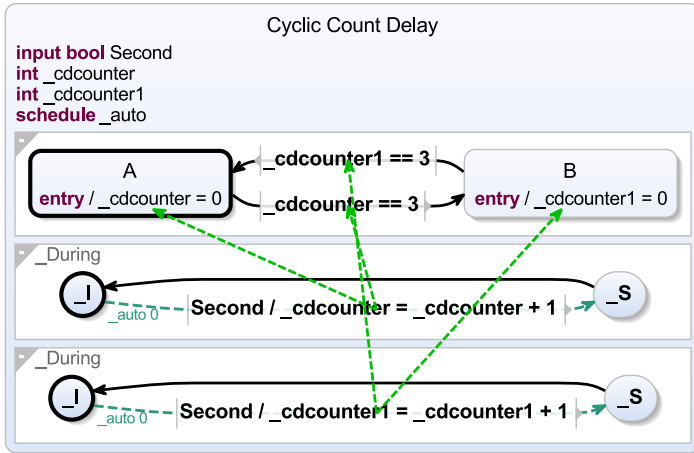


Figure 6.1.8. Cured expanded Cyclic Count delay

	Simple	with Pre	with SDs
Schedulable	No	Yes	Yes
Schizophrenia	-	Yes	No
Variables	2	8	2
States	22	44	18
Regions	5	10	5
Binary Size (b)	-	2702	1337

Table 6.1.1. Results of the different count delay approaches in SCCharts when compiling the Cyclic Count Delay model in Figure 6.1.6

to -1 in Figure 6.1.7, which was previously necessary to handle the case of a reset and a subsequent increment in the same tick, can be omitted.

Table 6.1.1 compares the three different implementations of SCCharts' count delay transformation when compiling the Cyclic Count Delay model in Figure 6.1.6. While the simple approach is unable to handle two cyclic count delays, the pre variant needs more variables, states, and regions than the SDs approach. Furthermore, the pre transformation also creates schizophrenic models. Handling schizophrenia does not come trivially [TS04]. The SD solution avoids schizophrenia.

Similarly, Schulz-Rosengarten uses SDs to consolidate clock increments in Timed SCCharts [SHM+18a] and to enable blackbox scheduling in object-oriented modelling [SSM19], which is sketched out in Section 8.2.3. *Run-Time enforcement* [PRS+17a], discussed in Section 7.2, is another practical example for SDs.

6.2 SCCharts Dataflow

As described in Section 2.3.6, the two main ways with which a program flow can be described are by its control-flow or by its dataflow. While one form can be described in its counter-part, languages and specific solutions often tend towards the one or the other. Especially the behaviour of computation-heavy controllers are more naturally described in dataflow, and modelling it in control-flow-oriented languages becomes cumbersome. While SCCharts is a control-flow-oriented language, it is relatively easy to use the model-based compilation approach to extend the language by dataflow modelling features. The developer is free to chose which concrete dataflow flavour is implemented.

In SCCharts, a *dataflow region* (DFR) is a region which comprises a list of assignments, which are evaluated in every tick when the enclosing state is active. *Definition*

Conceptionally, each assignment in a DFR resembles an immediate during action of the parent state. However, they can be ordered sequentially under the SCM_{oC}, as is shown in the following.

6.2.1 Syntax and Semantics

In contrast to control-flow regions (CFRs), the scheduling order of the contents DFRs is determined by the flow of data. While DFRs are usually comprised of mathematical equations, as e. g. in Lustre, they are assignments in SCCharts because of the SCM_{oC}. As established earlier, SCCharts' SCM_{oC} has two distinct properties which distinguishes it from classical synchronous languages: *variable persistence* and the IURP. (1) In SCCharts, all communication is made via variables and variables store their value even across ticks. (2) The IURP states that all concurrent region accesses to a variable are

6. Rapid Prototyping

```
1 x = pre(x) + (if b then 1 else 0)
2           + (if c then 1 else 0)
3           + 1
```

(a) Multiple conditions in Lustre

```
1 x = x + 1
2 x = b ? x + 1 : x
3 x = c ? x + 1 : x
```

(b) Multiple conditions
in SCCharts

```
1 x = x + 1
2 x = b ? x + 1
3 x = c ? x + 1
```

(c) Multiple conditions
in SCCharts using the
set operator

Listing 6.2.1. Dataflow extension syntax

ordered as discussed in Section 2.1.3. These principles are also applicable to the dataflow domain. All assignments in DFRs are inherently concurrent unless their dataflow dictates a relationship. Therefore, the IURP, while still applicable, might be just a means to order equations according to three priorities: initialize, update and read. Furthermore, the dataflow extension can use the sequential order of the equations when writes occur to the same variable. This enables the modeller to experiment further with the SCMoc using the standard SCCharts expression language.

Example The SCL serves as expression language for the dataflow assignments in KIELER. Control-flow instructions are forbidden. Listing 6.2.1 compares the syntax of the SCCharts dataflow extension to Lustre. Listing 6.2.1a shows a Lustre node block which increments a variable x by 3 if both inputs, b and c , are present. It is only incremented by 2 if one of the two inputs is present otherwise by 1. Using the previously mentioned principles the same program can be expressed as depicted in Listing 6.2.1b in SCCharts assuming that there is no other sequentially preceding assignment to x in the same tick. The variable persistence (1) ensures that the previous incarnation if x is used and the extended IURP (2) ensures the correct scheduling of the different assignments.

6.2.2 Sequentially Constructive Set

Definition The binary $?$ -operator, called *set operator* ($x = c ? e$), is a short-hand form of the common ternary conditional operator. If the condition is true, the variable value is updated to the new value. Otherwise, it keeps the stored value, which is possible because of the variable persistence (1).

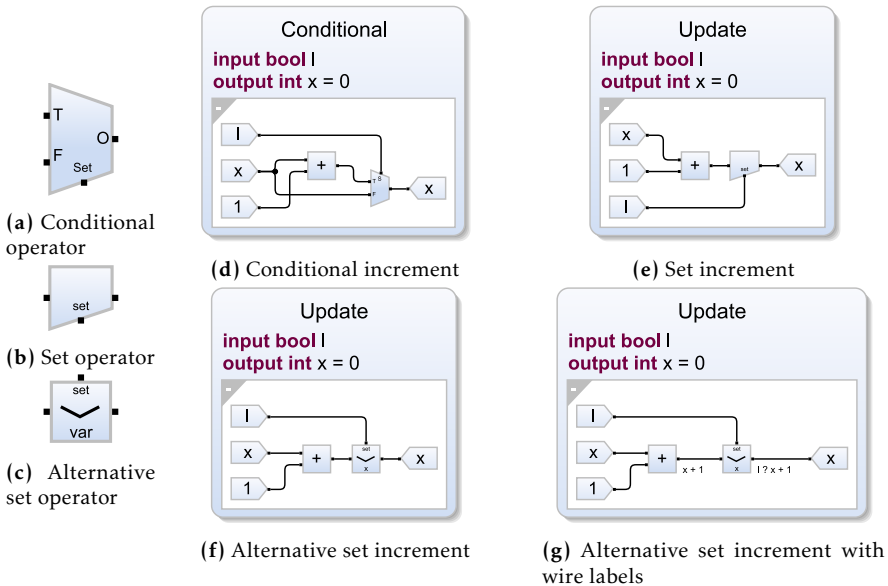


Figure 6.2.1. Sequentially constructive write operators in the SCCharts dataflow extension

Thus, the syntax example can be written even shorter in SCCharts, as shown in Listing 6.2.1c.

The common ternary conditional operator is graphically often depicted as a multiplexer, similar to the depiction in Figure 6.2.1a. T and F mark the two data inputs for the true and the false branches. Depending on the conditional input, a value is written to the output O. As a set does not have an else branch, the multiplexer can be split in half. Figure 6.2.1b shows a set. The variable receives the new value via O if the condition becomes true. Otherwise, the stored value remains valid.

I propose a third depiction for the set operator to help the modeller to keep an overview; particularly in large diagrams. ?? shows an alternative icon for set. The new value, which is fed to the actor from the left, is written to var, which is depicted on the bottom of the actor, if the set condition,

6. Rapid Prototyping

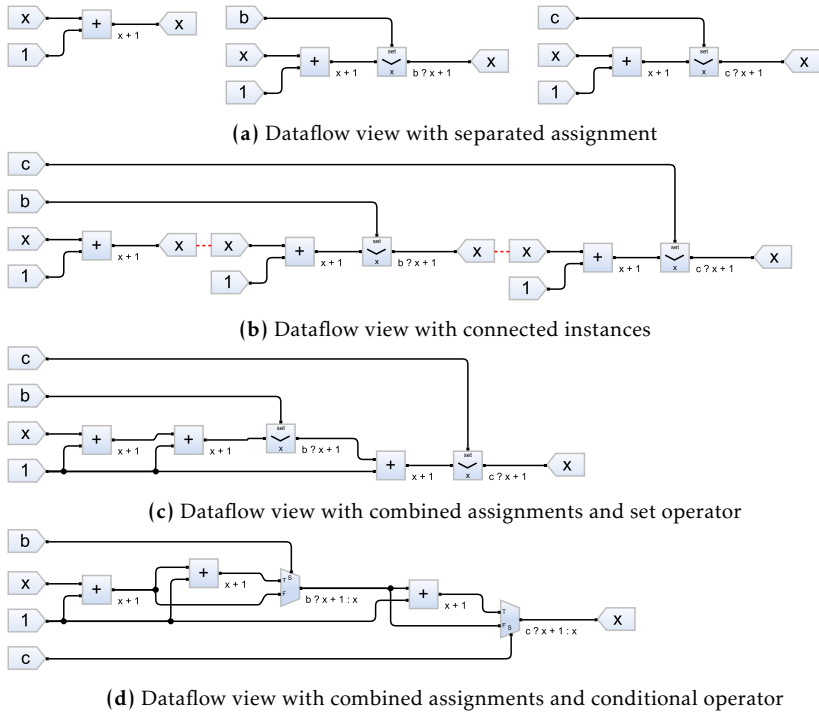


Figure 6.2.2. Sequentially constructive dataflow example of the counter in Listing 6.2.1; different views in a, b, c and d can be toggled interactively in KIELER.

connected to the top and pointing towards x , evaluates to true. The then current value of x is emitted on the right.

Example Figure 6.2.1d shows a simple increment of x with the classical multiplexer, whereas Figure 6.2.1e depicts the same program with the set operator. One can see less visual clutter in the latter. The else branch of the x assigned, which does not carry much information at all, can be ignored. Figure 6.2.1f shows the program with the proposed alternative set icon. Additionally, the concrete expression can be turned on interactively as wire label if desired by the modeller.

If synthesizing every assignment from the example in Listing 6.2.1, the diagram looks like Figure 6.2.2a. While giving a concrete overview over the single assignments, the only information about their relationships might be their ordering, which can be visually obfuscated or challenging in big diagrams with many relationships. Since dataflow assignments in SCCharts are inherently concurrent to each other and other regions, the IURP or explicitly set constraints impose an ordering. The interactive transient view concept in KIELER allows for more concrete visual help to show these kinds of relationships between variable accesses. Sequentiality can be illustrated explicitly, as shown in Figure 6.2.2b. The red dashed edges illustrate which instances of x are identical. The established order can be combined into a single diagram, such as illustrated in Figure 6.2.2c. The explicit write and read access nodes for x have been removed and a single flow for x is now visible. Additionally, constants, such as 1 in the example, can be combined as well but are not required to be if the modeller decides otherwise.

Although the illustration in Figure 6.2.2c is the most compact depiction, the set operator might add ambiguity, which impairs understandability when more than one expression term is involved in the set operation. Here, it is unclear if $x + 1$ is set two times when b becomes true. Therefore, as before, the modeller is free to choose the illustration, which fits their needs best. The conservative conditional view is shown in Figure 6.2.2d.

Note that it is still an open question in SCCharts if sequential dataflow accesses should be modelled explicitly or not. In the concurrent context of CFRs, the IURP is used by default, but the modeller can influence the schedule with SDs. Similarly, the modeller could influence the ordering of dataflow assignments explicitly, which are naturally concurrent.

In the latest release of KIELER SCCharts, dataflow assignments are not forced to be sequential by their order if conflicting, but sequentiality has to be marked explicitly by the modeller. Figure 6.2.3 shows a second example of the dataflow extension with an alternative syntax for sequentiality using an explicit sequential operator. Whether the sequential schedule should be enforced by the order or stated explicitly by the modeller, still remains to be determined. The fast prototyping enabled by KiCo makes experiments with both variants easy.

6. Rapid Prototyping

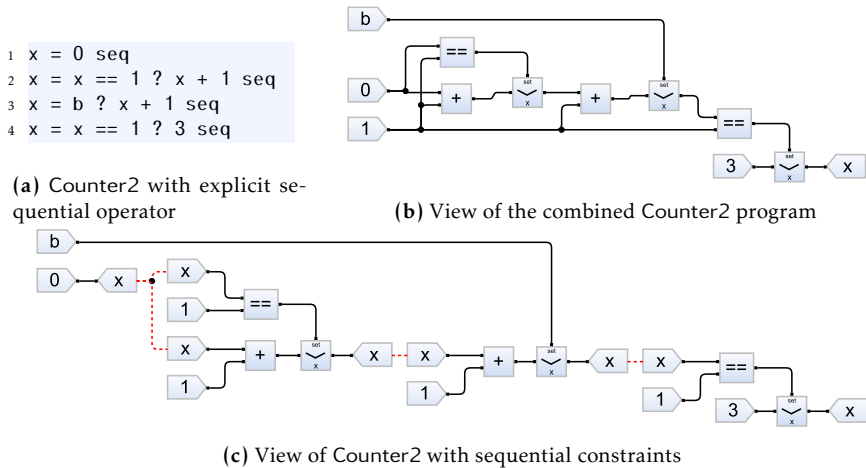


Figure 6.2.3. Explicit sequential constraints in dataflow SCCharts

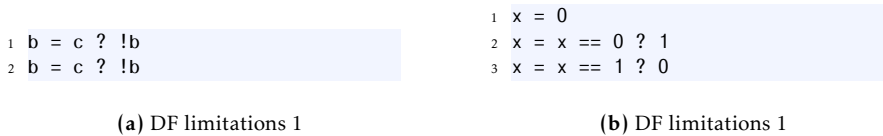


Figure 6.2.4. Dataflow SCCharts limitations imposed by circular dependencies

Example The sequential constraints of Counter2 are shown in Figure 6.2.3c. 0 is assigned to x, which is then used in the addition but also in the comparison. Afterwards, x is incremented if b is present and finally, it is set to 3 if x is 1.

— The merged diagram is shown in Figure 6.2.3b.

Example The usual dependency limitations of SCCharts also apply in the dataflow extension. While concurrent and sequential relationships can be resolved as mentioned above, circular dependencies will render the SCChart unconstructive, i. e. not schedulable. Figure 6.2.4 shows two examples of simple dataflow regions, which cannot be scheduled in SCCharts.

—

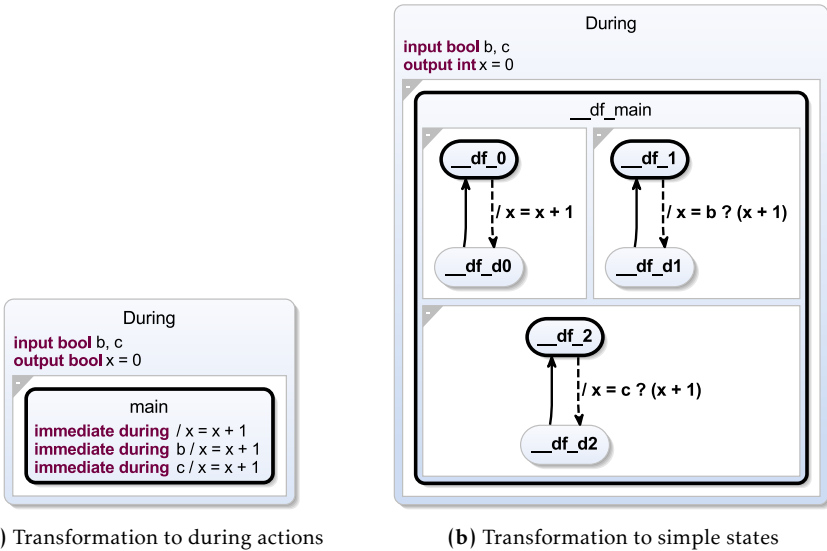


Figure 6.2.5. Transformation of the example from Figure 6.2.2 to control-flow SC-Charts

6.2.3 Transformation to Control-flow

Languages which provide dataflow/control-flow hybrid models often employ a dataflow-based MoC and embed statemachines in some way, such as Lustre and SCADE with their clock extensions [BCH+08]. In this section, the model-based compilation approach is used to demonstrate that the other way around is also possible.

As stated in the introduction of Section 6.2, SCCharts' DFRs are inherently concurrent and should be executed in every tick as long as the enclosing superstate is active, similar to immediate during actions in a pure control-flow-oriented domain. Naturally, every dataflow assignment can be translated into a corresponding immediate during action. During actions are considered syntactic sugar in SCCharts and are eventually transformed into simple states which self-loop.

6. Rapid Prototyping

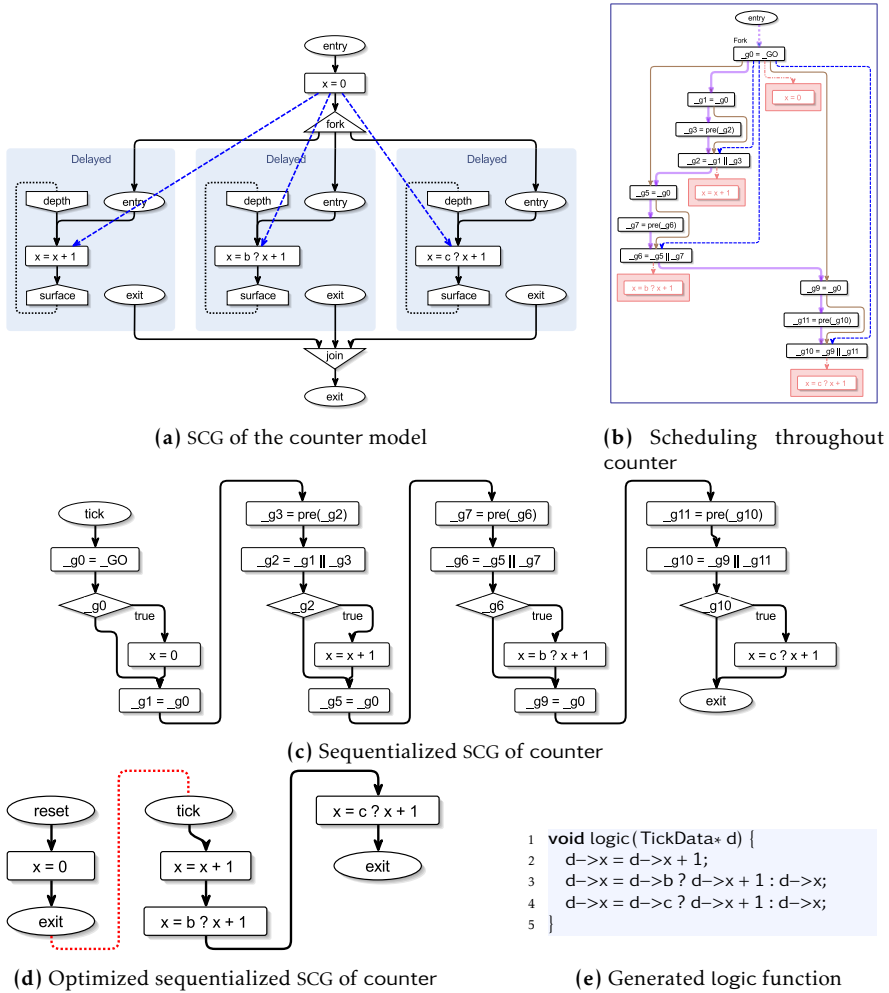


Figure 6.2.6. Transformation of the counter example from Figure 6.2.2

6.2. SCCharts Dataflow

A processor can transform the dataflow SCChart depicted in Figure 6.2.2 *Example* to a semantically identical pure control-flow-oriented SCChart, illustrated in Figure 6.2.5a. As before, x is incremented indefinitely by 3, 2 or 1, depending on the inputs b and c . The during actions are then transformed further by the default SCCharts compilation chain. Alternatively, the dataflow processor can directly create simple states with self-loops. In either way, the model depicted in Figure 6.2.5b shows the semantically equivalent Core SCChart. —

Technically, the dataflow transformation is implemented as dedicated *Implementation* processor in KIELER. Since it can be invoked on any regional scope and can reference other SCCharts as well, it is called programmatically as co-processor within the referenced SCCharts processors whenever a DFR is encountered in the model parse. —

Since the control-flow compilation of SCCharts takes care of the IURP, there is no need for special handling of the IURP in the dataflow transformation. Note that to schedule the set operator concurrently, different concurrent instances of the operator have to be commuting as well, i. e. the conditions in this case are not allowed to depend on x , as explained as limitation in Section 6.2.1. If the modeller schedules dataflow assignments explicitly, the transformation must also transfer these instructions to Core SCCharts level, which can be done via SDs.

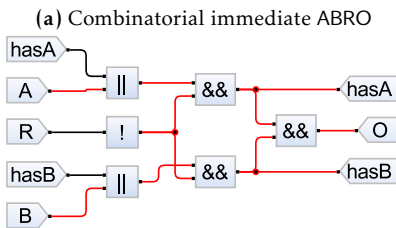
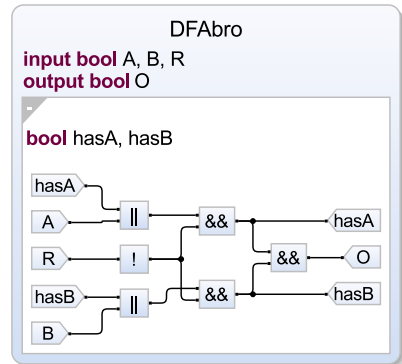
Figure 6.2.6 shows the standard compilation of the Counter Core SCChart. *Example* The SCG of the counter example is shown in Figure 6.2.6a. As usual, the modeller can inspect some or all dependencies here. The scheduling path is depicted in purple in Figure 6.2.6b. Following the netlist-based approach, the sequentialized program, shown in Figure 6.2.6c, contains the guard assignments for the states which originated from the dataflow–control-flow transformation. However, for assignments which get evaluated in every tick, such as in active dataflow regions, the netlist guards form *persistent state* patterns, which can be optimized according to Section 5.2.8. The final optimized version of the counter example is shown in Figure 6.2.6d. The serialized C equivalent is depicted in Figure 6.2.6e. It does not include any overhead and fulfils the intention of the modeller: In every tick, x is increased by 1, 2 or 3 depending on the inputs b and c . —

6. Rapid Prototyping

```

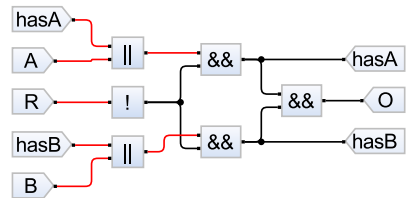
1 scchart DfAbro {
2   input bool A,B,R
3   output bool O
4
5   dataflow:
6   bool hasA, hasB
7
8   hasA = (hasA || A) && !R
9   hasB = (hasB || B) && !R
10  O = hasA && hasB
11 }

```



(c) Immediate reaction to A and B

(b) Diagram of the immediate dataflow ABRO



(d) Immediate reaction to A, B and R

Figure 6.2.7. Immediate ABRO as dataflow SCCharts

6.2.4 Dataflow Modelling

DFRs can be embedded as regional scope in a superstate (resp. root state) similar to control-flow regions.

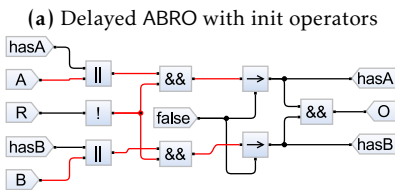
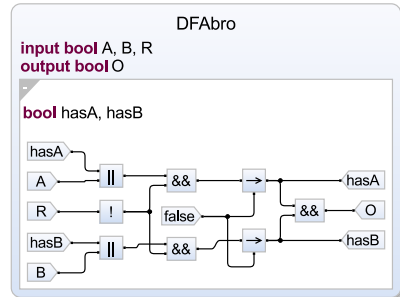
Example Figure 6.2.7 shows a variant of ABRO in as dataflow model in SCChart using only combinatorial functions. Figure 6.2.7a shows the textual syntax of the model shown graphically in Figure 6.2.7b. The interface is as usual for ABRO. In the DFR, two local variables, `hasA` and `hasB`, store whether or not A and B are true in the current tick or have been set to true in previous ticks. `O` becomes true if both, `hasA` and `hasB`, are true. Everything is reset by `R`. Contrary to the classical ABRO, which is irresponsive for the first tick, this program reacts immediately. A simulation step with A and B present is

6.2. SCCharts Dataflow

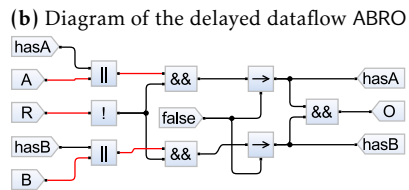
```

1 scchart DFABro {
2   input bool A,B,R
3   output bool O
4
5   dataflow:
6   bool hasA, hasB
7
8   hasA = false -> ((hasA || A) && !R)
9   hasB = false -> ((hasB || B) && !R)
10  O = hasA && hasB
11 }

```



(c) Delayed reaction to A and B



(d) Delayed reaction to A, B and R

Figure 6.2.8. Delayed ABRO as dataflow SCCharts

depicted in Figure 6.2.7c. Wires in red indicate *true*. O is true immediately. In the second step, shown in Figure 6.2.7d, R is also true and resets the circuit.

To obtain the classical behaviour of ABRO, SCCharts have a processor which transforms an init operator ($->$) similar to Lustre. The init expression returns the left-hand operand in the first tick and the right-hand operand in the following ticks.

A delayed dataflow ABRO with the two init operations is shown in Figure 6.2.8. Now, hasA and hasB are guarded by the init in the first tick, see Figure 6.2.8a. In the diagram in Figure 6.2.8b, the init is shown as own actor. The simulation step with A and B true in the first tick in Figure 6.2.8c, does not emit O, because the inits propagate false in the first tick. Afterwards, with R present, O stays false.

Observe that due to the SCMoc, the dataflow ABRO in both variants does not need a pre operator, contrary to the ABRO model presented in Sec-

6. Rapid Prototyping

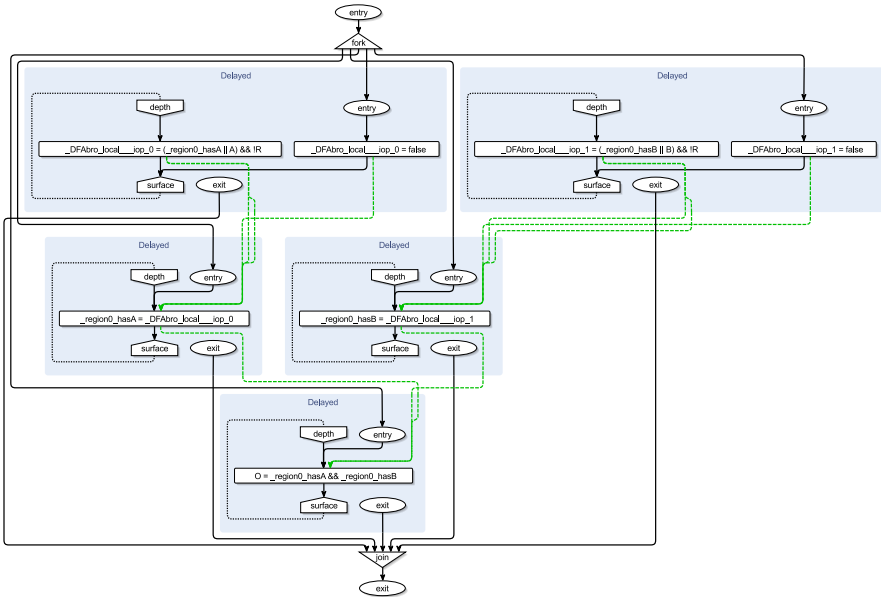
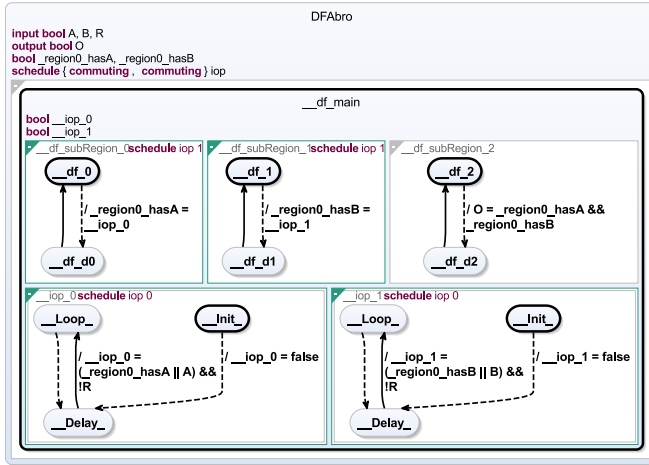


Figure 6.2.9. Inducing a schedule for the init operation

tion 2.3.6 on page 47 in Lustre. The examples exploit the variable persistence of the SCM_{oC}. `hasA` and `hasB` can directly use the current state of their variable storages and therefore their values from the previous tick. This is not possible in classical synchronous languages, which must obtain the value via pre or more complex operations, such as *followed-by*, which inherently employs a pre operation. However, the schedule for self-referencing must be resolved in the netlist. While there are several ways to implement the transformation for the `init` operator, e. g. exploiting the standard IURP, a simple way to ensure correct scheduling are SDs, which can be added automatically by the transformation.

Figure 6.2.9 shows the schedule of the operations induced by an SD *Implementation* added by the transformation. The indices in the SD are set to *commuting*, meaning that directives with the same index can be scheduled arbitrarily. Concurrently, it is decided for `hasA` (resp. `hasB`) via `__iop_0` (resp. `__iop_1`) if the left-hand or right-hand part or the `init` is active. The result is assigned to `hasA` (resp. `hasB`). To avoid a dependency cycle, the transformation adds a schedule `iop`. The regions that set `__iop_0` and `__iop_1` are scheduled strictly before they are used. Therefore, a reference to `hasA` (resp. `hasB`) refers to the incarnation of the variable from the previous tick. The concrete dependencies in the SCG are shown layouted in Figure 6.2.9b. This is another application example of SDs. SDs can improve single compilation steps even if the modeller does not actively use them to model.

Similar to the possibility to reference other SCCharts in the control-flow-oriented modelling in SCCharts, also shown in Section D.1, other models can be referenced as own actors in DFRs. Since DFRs employ the same SCM_{oC} as CFR, there is not restriction on what SCCharts can be referenced.

Figure 6.2.10 shows a variant of ABRO using a hybrid dataflow-control-*Example* flow modelling style. The textual model in Figure 6.2.10a shows two SCCharts, `DFAbro` and `Latch`. They can also be modelled in separate files but it is perfectly fine for small models to put them into one unit. `Latch` models a simple latch which stores a binary information. It has a set input `S` and an reset input `R`. The current state is written to the output `O`. The behaviour is modelled in control-flow form using two states, `False` and `True`, which are toggled depending on the inputs. The root model `DFAbro` comprises the usual interface for ABRO and a DFR. Inside the DFR, the two local variables

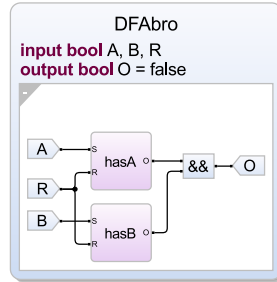
6. Rapid Prototyping

```

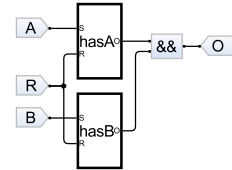
1 scchart DfAbro {
2   input bool A, B, R
3   output bool O = false
4
5   dataflow:
6   ref Latch hasA, hasB
7
8   hasA = {A, R}
9   hasB = {B, R}
10  O = hasA.O && hasB.O
11 }
12
13 scchart Latch {
14  input bool S, R
15  output bool O = false
16
17  initial state False
18  if S && !R do O = true
19  go to True
20
21  state True
22  if R do O = false
23  go to False
24 }

```

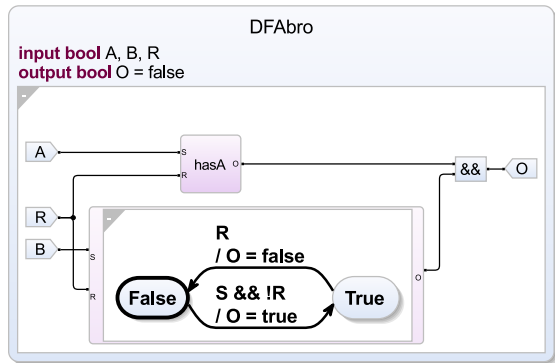
(a) Textual SCChart using multiple SCCharts and hybrid modelling



(b) Referred actors in a dataflow region

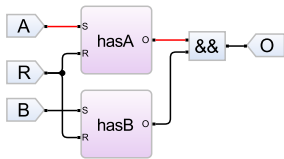


(c) Dynamically skinned actors

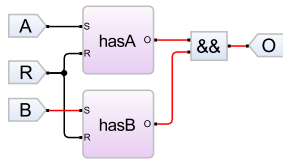


(d) Referred control-flow actor

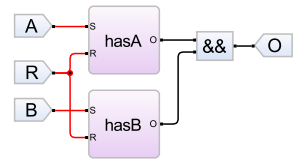
Figure 6.2.10. Mixing dataflow-based and control-flow-based modelling styles



(a) Simulation step with A true; second step



(b) Simulation step with B and A stored in the latch; third step



(c) Simulation step with all inputs true; fourth step

Figure 6.2.11. Simulation of DfAbro from Figure 6.2.10

hasA and hasB are declared as Latch. They constitute references to the previously modelled Latch and can be used in SCL expressions. In the DFR, the inputs S and R of hasA are connected to A and R of ABRO. hasB is connected analogously but with B instead of A. The output of ABRO O is then set to the conjunction of the outputs of the latches. Note that $\text{hasA} = \{A, R\}$ is a short form of $\text{hasA.S} = A$ and $\text{hasA.R} = R$ to set all inputs simultaneously. Both forms are valid. The diagram of the model is depicted in Figure 6.2.10b. The purple actors represent referenced SCCharts. They can be handled like any other scope. Figure 6.2.10d shows the expanded hasB actor and the hybrid nature of the model. In the simulation, ABRO behaves as expected. If A becomes true in the second step, as depicted in Figure 6.2.11a, the output of the latch is toggled. If B follows in the third step but with A false again, O is still true because the stored value of A is still true. Finally, if R becomes true, the behaviour is reset.

Figure 6.2.10c shows that referenced actors can be *skinned* arbitrarily in KIELER. A textual native description of the view framework KLightD can be added to an SCChart. If that chart becomes referenced, the specific description overrides the default depiction. It is also possible to skin different instances of the same reference differently by applying the synthesis description to the variable and not to the referenced chart. In the example, the actor looks more like a standard latch but with the SCCharts colour scheme. Similarly, the different depictions of the set operator in Figure 6.2.1 are just different skins of the same model element in the diagram synthesis. In fact, the complete circuit transformation and simulation, presented in Section 5.2.9, uses the SCCharts dataflow extension in the current version of KIELER. The actors are simply skinned as logic gates. Further, Section 6.2.6 gives another example on dataflow actor skinning in illustrating views for different syntactical flavours.

As common control-flow constructs can be modelled together with DFRs, aborts can be employed to preempt DFRs. However, it might be cumbersome to have to model an embedded DFR in a superstate for one calculation just be preempted immediately. Hence, as further convenience, a DFR can be marked as *once*. It will then conceptionally be handled similar to entry actions. The transformation can be done directly or according to SLIC.

6. Rapid Prototyping

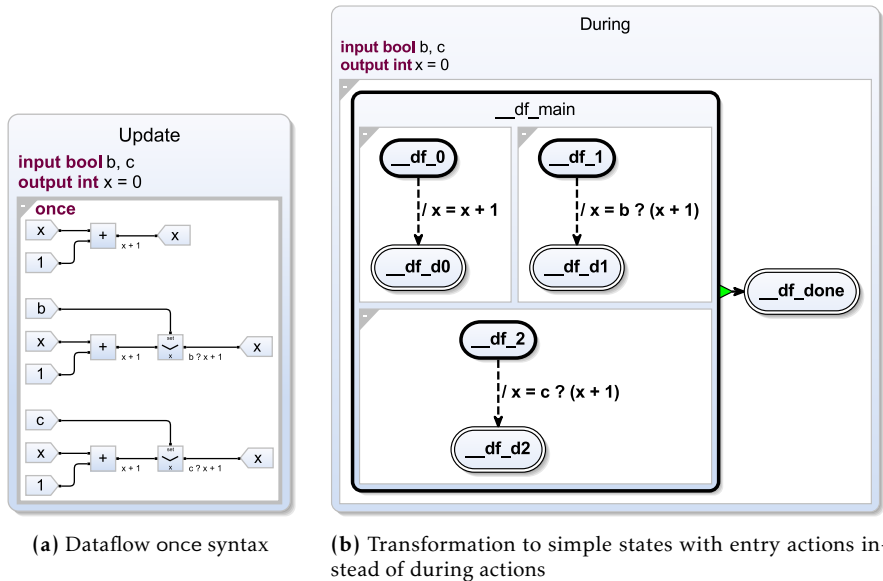


Figure 6.2.12. Dataflow once transformation of the Counter example from Listing 6.2.1

Example Figure 6.2.12 shows a DFR modelled with the once flag. Syntactically the dataflow region is depicted with a thick border similar to initial states. It is shown in Figure 6.2.12a. The straightforward transformation is illustrated in Figure 6.2.12b. Instead of single self-loops, each assignment is evaluated once. The state encapsulating the semantic dataflow is then left immediately.

6.2.5 Transient Dataflow Views

While it might be preferable in some contexts to be able to model dataflow relations in dataflow regions, such as PID controllers, both worlds can also benefit from each other through dedicated views without the need to model other domains directly. As explained at the beginning of this section, DFRs are semantically similar to immediate during actions of superstates in CFRs. These actions are usually depicted textually in the KIELER SCCharts editor, as

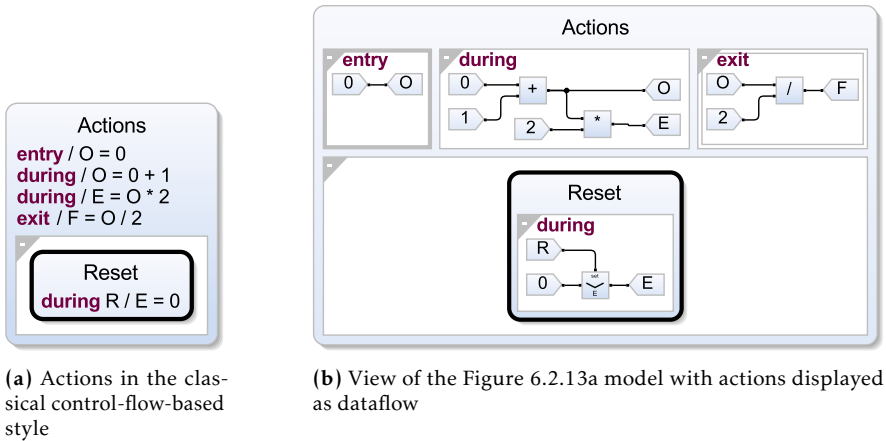


Figure 6.2.13. Different action views of the same SCCharts model

shown in Figure 6.2.13a. However, the same view diagram synthesis which synthesizes the DFRs in the dataflow extension can be used to synthesize the actions of superstate.

Figure 6.2.13b shows the same SCChart but with the dataflow synthesis *Example* for actions. Entry and exit actions are marked with a thick and double border, respectively, similar to the graphical syntax of states.

Another example of this principle is the induced dataflow view presented elsewhere [WSS+18]. Here, dataflow relationships in pure state-based models are made visible explicitly during the view synthesis. There is no need to modify the original control-flow-based model to gain this information. A variation of this view, the causal dataflow view, is presented in Section 4.1. —

6.2.6 Dataflow Evaluation

This section explains how the translation into a control-flow-based statecharts languages affects the generated code.

Figure 6.2.14a shows the Lustre code of the counting node, originally *Example* presented by Colaco et al. [CPP05]. The corresponding SCADE diagram is depicted in Figure 6.2.14b. Similar to SCADE, both equations are depicted

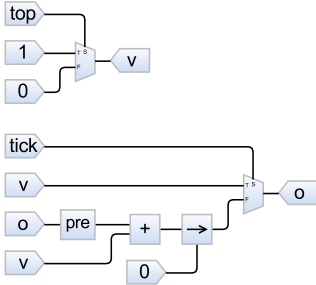
6. Rapid Prototyping

```

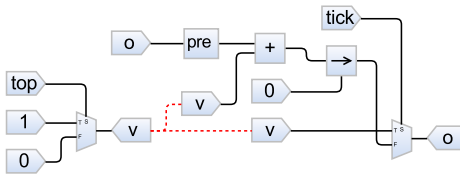
1 node counting(tick:bool; top:bool)
2   returns (o:int);
3   var v:int;
4   let o = if tick then v else
5         0 -> pre o + v;
6       v = if top then 1 else 0;
7 tel;

```

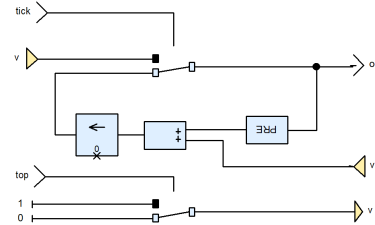
(a) Lustre, adapted from [CPP05, Fig. 1]



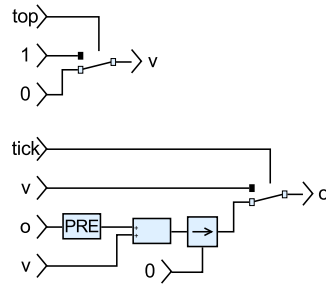
(c) View 1, SCChart style



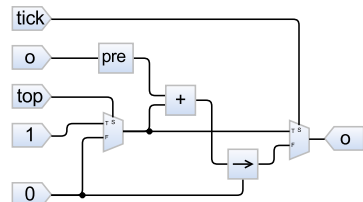
(e) View 3, displaying the sequential flow of data



(b) Manually drawn SCADE diagram, adapted from [CPP05, Fig. 1]



(d) View 2, SCADE style



(f) View 4, with connected equations

Figure 6.2.14. Counting node example in Lustre, SCADE and four different views synthesized by KIELER

in the graphical representation. As described before, the diagram synthesis can be configured interactively to display several variations to fit the modellers needs. For example, the whole diagram can be *skinned* to get a SCADE look-and-feel, see Figure 6.2.14d. The disconnected components can also be ordered sequentially, as in Figure 6.2.14e. The sequential ordering is indicated by the red dashed hyper-edge. Variable v , written in the first equation, is read in the second one, even though their order is reversed in the textual representation of the node. Since v is not visible from outside the node, its graphical input/output representation can be omitted altogether, see Figure 6.2.14f.

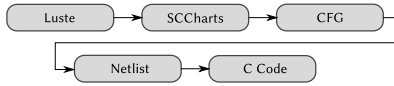
To exemplify the code generation for Lustre programs in KIELER, KiCo — Example is configured as depicted in Figure 6.2.15a. First, the Lustre program is compiled into an SCCharts model. Possible graphical representation of the generated SCChart are shown in Figure 6.2.14c to Figure 6.2.14f. The data-flow representation of SCCharts is translated into its semantically equivalent control-flow-oriented counter-part according to the transformation w.r.t. immediate during actions discussed earlier, shown in Figure 6.2.15b. From here, the default netlist-based compilation approach of SCCharts, as explained in Section 5.2, is used. In this example ANSI C code is synthesized.

The overhead, which might be introduced by the control-flow-based compilation approach, can be reduced in stateless models. Figure 6.2.15e shows the sequentialized result of the compilation before the *persistent state* optimization, see Section 5.2.8. The final optimized version is depicted in Figure 6.2.15c.

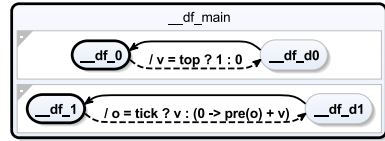
The C code of the counting node example generated by KiCo is listed in Figure 6.2.15d. Code from the immediate environment, such as the reset function and the `_GO` signal are omitted here. The generated logic function directly resembles the data-flow of the node. This example demonstrates that the generated code is still compact and readable even if SCCharts data-flow equations are translated into a statechart, then into a control-flow graph and finally into imperative code.

Note that saving the previous value of `o` is embedded in the code, because `pre` is an extended feature of SCCharts and not part of the underlying expression language. The register fetch and save can be observed in Lines 3 and 8 of Figure 6.2.15d. However, since sequential constructiveness inherently

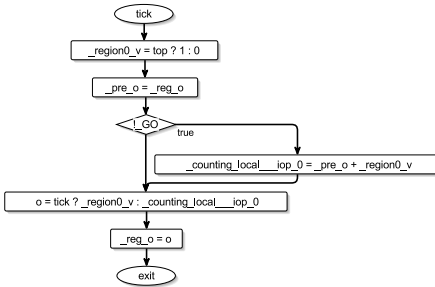
6. Rapid Prototyping



(a) Lustre-SCCharts compilation chain in KIELER



(b) Control-flow-oriented SCCharts model of counting node

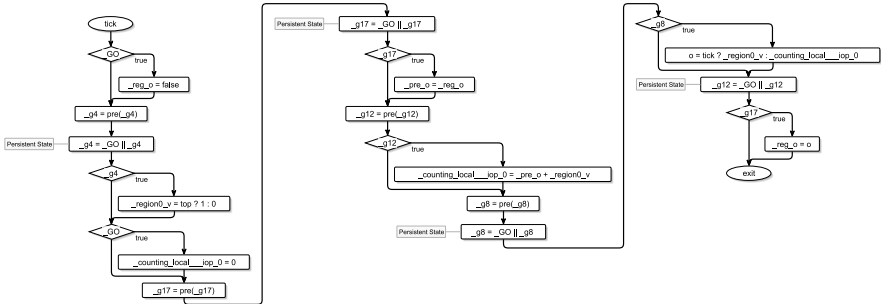


(c) Optimized sequential control-flow-graph of the counting example

```

1 void logic (TickData> d) {
2   d->_v = d->top ? 1 : 0;
3   d->_pre_o = d->_reg_o;
4   if (! d->_GO) {
5     d->_iop_0 = d->_pre_o + d->_v;
6   }
7   d->o = d->tick ? d->_v : d->_iop_0;
8   d->_reg_o = d->o;
9 }
  
```

(d) logic function for counting node generated by KiCo using the compilation chain depicted in Figure 6.2.15a; red parts are not serialized when compiling without pre



(e) Sequentialized control-flow graph of the control node from Figure 6.2.14 generated by KiCo: *persistent states* evaluate to true and can be optimized.

Figure 6.2.15. Using the netlist-based code generation approach to compile Lustre programs in KIELER

stores the immediate previous value, the modeller can omit the pre operator altogether. In this case, the marked parts in the listing will be omitted.

6.3. Symmetrical Actions

Program	Lines of code			Object code size (O0 / O2) [kiBi]		Avg. Reaction Time [ns]	
	Source	SCCharts	Lustre	SCCharts	Lustre	SCCharts	Lustre
heater_ctrl	83	170	343	4.7 / 3.1	6.9 / 4.7	505	908
stopwatch	26	105	281	2.4 / 1.8	4.2 / 3.6	152	304
counting	6	46	119	1.7 / 1.5	3.1 / 2.5	134	185
implies	4	32	32	1.6 / 1.4	1.3 / 1.2	175	95

Table 6.2.1. Selected results of the benchmarks comparing the sizes and reaction times of the SCCharts dataflow extension and Verimag Lustre V6

Apart from the counting example, the computation times of the SCCharts netlist-based compilation and the Verimag Lustre V6 compiler¹ were compared with approximately 50 other programs from a publicly available Lustre example repository². The benchmark used the same approach and technology as the final compilation approach tests explained in Section 5.5 on the same system. Table 6.2.1 shows results of the benchmarks for selected programs. The average over the deviation of the average reaction time of all programs is about -4%, and about half of the programs have a higher average reaction time using Lustre generated code. As preliminary conclusion, the SCCharts code generation did not perform worse compared to the Lustre V6 compiler on average. In depth analyses and comparisons between Lustre and SCCharts are currently under investigation by Grimm. Her main approach is outlined in Section 8.2.

6.3 Symmetrical Actions

Since being a strong conservative extension to SyncCharts, SCCharts rarely deviate from the classical SyncCharts notation. However, some changes were made for clarity reasons and are partly covered elsewhere [Mot17][HDM+13c]. For example, instead of the a hashtag (#) in the transition trigger, an immediate transition is now displayed as a dashed edge in the graphical syntax,

¹<http://www-verimag.imag.fr/Lustre-V6.html>

²<https://github.com/jahierwan/lustre-examples>

6. Rapid Prototyping

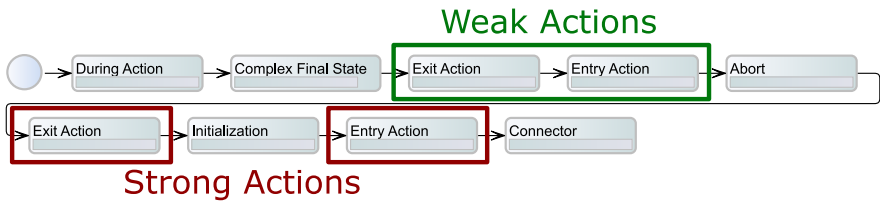


Figure 6.3.1. Excerpt from the extended SCCharts compilation system with additional actions

since the delay behaviour is a property of the transition and not of the trigger. In fact, a transition does not even require an explicit trigger or label thereof.

The *action asymmetry* refers to the asymmetrical handling of entry and exit actions w.r.t. to preemption. In SyncCharts, an entry action is a simple abbreviation. It says that the associated action is executed at every incoming transition. One can debate whether the action lives inside the state or at the incoming transitions. Therefore by definition, entry actions cannot be aborted in SyncCharts by an immediate outgoing preemption. However, exit actions, as being defined as part of the superstate and not of the outgoing transitions, are preemptable.

While it is certainly possible to build explicit processors to handle different kinds of actions (and preemptions), the compilation chain can be modelled using the same processors to handle these kinds in the model-based compilation approach. Consider Figure 6.3.1, which shows an excerpt of the extended SCCharts compilation system. The system is modified such that the same two processors responsible for the entry and the exit actions are used twice. I propose to transform *weak actions*, which can be preempted, before the abort transformation. *Strong actions* are then managed afterwards. Since any feature which is transformed before the preemption handling can be aborted by the preemption, the actions, too, are preemptable. On the other hand, the strong actions are transformed afterwards and therefore, are not affected by the abort.

Example Figure 6.3.2 shows the step-wise transformation. The OSM is shown in Figure 6.3.2a. It contains a strong and a weak entry action in the superstate Entry. The superstate is preempted immediately if A is present. First, the

6.3. Symmetrical Actions

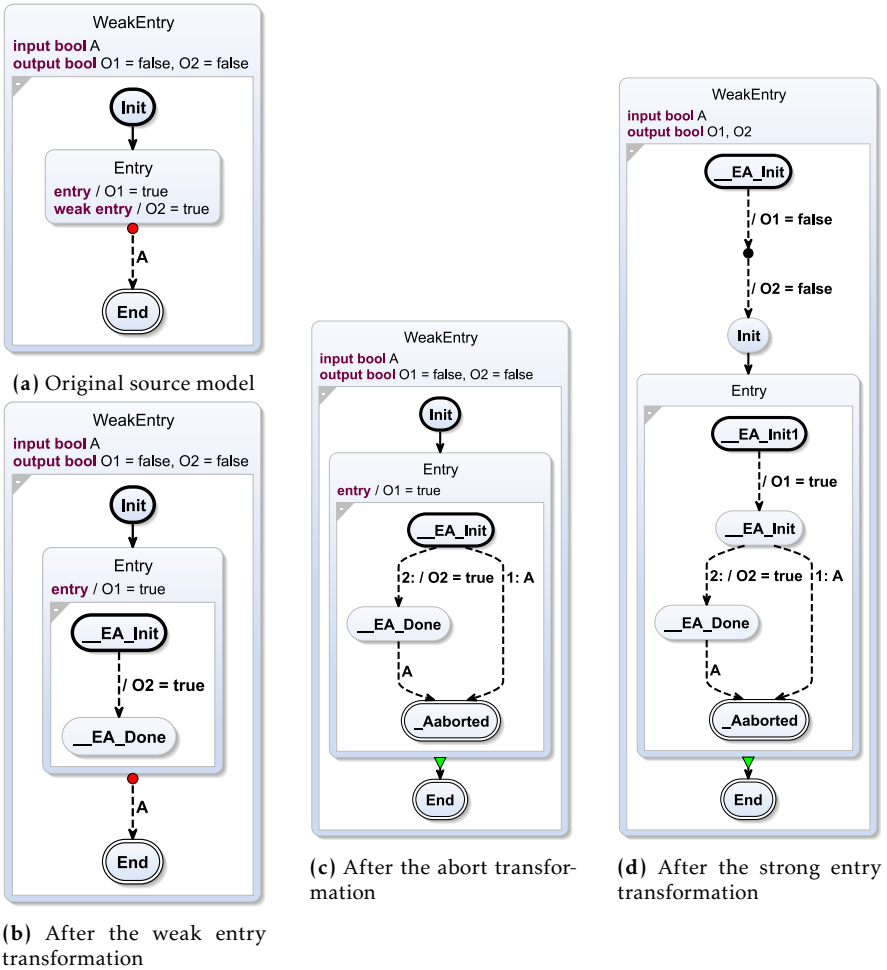


Figure 6.3.2. Step-wise transformation of the WeakEntry example

weak entry action is transformed. The transformation includes the behaviour in the superstate, which is shown in Figure 6.3.2b. Afterwards, the abort transformation handles the preemption. All behaviour inside the superstate,

6. Rapid Prototyping

```
module ReEmit:
signal A : combine boolean with or in
  emit A(true);
  if ?A then emit O end;
  emit A(true)
end
```

Listing 6.4.1. ReEmit (taken from [PEB07, p. 22]) is not valid in Esterel due to the emission of A after its value is read with ?A; it is valid in SCEst.

including the expanded weak entry action, can be aborted. Figure 6.3.2c depicts that the set of O2 introduced by the weak entry action can now be circumvented by A. However, the strong abort remains untouched. Finally, in Figure 6.3.2d, the strong abort is transformed. The new contents are not influenced by the preemption and executed regardless of A. Exit actions are transformed analogously.

Implementation

This way, weak and strong symmetrical action handling can be modelled in the compilation system without the need of new processors. The previously generated transformations for both actions combined required an addition of three lines of code to filter for the correct model elements. The rest of the processor code remained unchanged.

6.4 Sequentially Constructive Esterel

To give a non-SCChart related prototype example, this section will introduce *sequentially constructive Esterel*, or SCEst. SCEst is a dialect of Esterel, meaning that it is a conservative extension to Esterel but follows the SCM_oC. It builds on the SCL, introduced in Section 5.1. SCEst variables can be shared among threads as well as sequentially modified. This permits programs re-initialization of signal statuses and values, expressed with the new SCEst statements `unemit` and `set`. SCEst is defined as a set of transformation rules from SCEst to the SCL. The transformation rules are fairly straightforward, structural M2MTs, which can be implemented as processors. This leverages the existing formal semantics for SCL and builds on the result that the SCM_oC conservatively extends the *Berry constructiveness* demanded by Esterel [AMH+14].

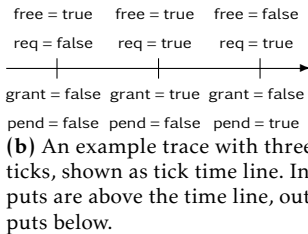
6.4. Sequentially Constructive Esterel

```

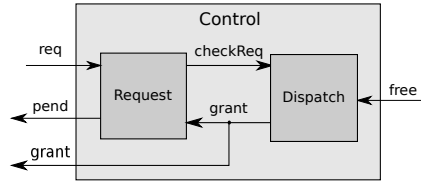
1 module Control
2 input bool free, req;
3 output bool grant, pend;
4 {
5   bool checkReq;
6
7   fork {
8     // Thread "Request"
9     Request_entry:
10    pend = false;
11    if (req)
12      pend = true;
13      checkReq = req;
14      if (pend & grant)
15        pend = false;
16      pause;
17      goto Request_entry;
18  }
19  par {
20    // Thread "Dispatch"
21    Dispatch_entry:
22    grant = false;
23    if (checkReq & free)
24      grant = true;
25    pause;
26    goto Dispatch_entry;
27  }
28  }
29 }

```

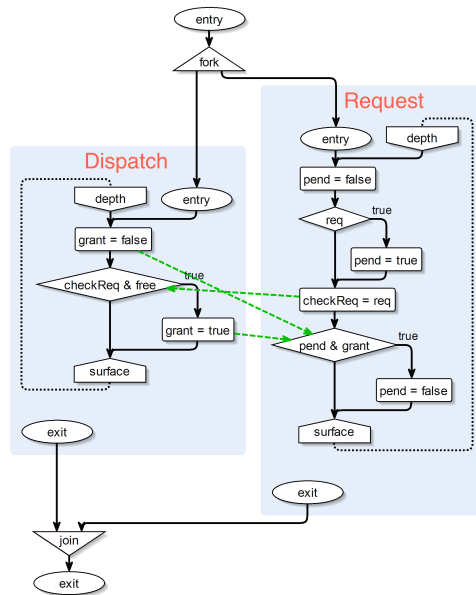
(a) The SCL program



(b) An example trace with three ticks, shown as tick time line. Inputs are above the time line, outputs below.



(c) The data-flow view



(d) The SC Graph (scg), indicating sequential flow (continuous arrows), concurrent data dependencies (dashed, green arrows), and the tick delimiter edges (dotted lines).

Figure 6.4.1. The Control example, illustrating the sequential modification of shared variables.

6. Rapid Prototyping

Example Consider the Control example presented in Figure 6.4.1. The original SCL description is shown in Figure 6.4.1a. As discussed elsewhere [HMA+14], this example is an abstracted version of a programmable logic controller software used in the railway domain.

Remark The functionality of Control as shown here could be achieved with even less code (e. g. without the usage of `checkReq` or the intermediate setting of `pend` to `true`). However, the example follows the logic of the original application and concentrates on the usage of the five shown flags.

The functionality of Control is depicted in Figure 6.4.1c and is as follows. A Request thread takes resource requests, indicated by `req`, from the environment and internally signals requests with `CheckReq` to a Dispatch thread. If a resource is available, indicated by the environment with `free`, the request is granted, signaled to the Request thread and the environment with `grant`. Otherwise, the request is still pending, indicated by the Request thread with `pend`. An example trace is shown in Figure 6.4.1b. In the initial tick, the resource is free but not requested; in the second tick, it is free, requested, and hence granted; in the third tick, the resource is requested but not free, therefore the request remains pending. The SCG for control is shown in Figure 6.4.1d.

The functionality of Control can be expressed in a rather straightforward fashion with `scl`. A stumbling point is `pend`, which (1) serves to communicate with the (concurrent) environment, and (2) may change from `false` to `true` and back to `false` within a tick. Esterel signals can handle (1), encoding `true` as signal presence and `false` as absence. However, signals cannot deliver (2) because they must evolve monotonically within a tick and must obey the *emit-before-test* discipline. Conversely, Esterel variables allow (2), but don't allow (1).

In contrast to Esterel, SCEst has no difficulties reconciling (1) and (2). SCEst provides variables with the same capabilities as `scl`. The SCEst-equivalent of Control based on variables is shown in Listing 6.4.2a. In addition, SCEst provides signals that can be used as in Esterel but with fewer restrictions:

1. SCEst signals may be emitted *after* they have been tested and possibly have been determined to be absent.

6.4. Sequentially Constructive Esterel

```

1 module ControlSCEstVar:
2   input bool free, req;
3   output bool grant,
4     pend;
5   var checkReq: boolean
6     in
7     % Thread "Request"
8     loop
9       pend := false;
10      if req then
11        pend := true
12      end;
13      checkReq := req;
14      if pend and grant
15        then
16        pend := false
17      end if;
18      pause
19    end loop
20  ||
21  % Thread "Dispatch"
22  loop
23    grant := false;
24    if checkReq and
25      free then
26      grant := true
27    end;
28    pause
29  end loop
30 end var
31 end module

```

(a) SCEst with variables

```

1 module ControlSCEstSig:
2   input free, req;
3   output grant, pend;
4   signal checkReq in
5   [
6   % Thread "Request"
7   loop
8     present req then
9       emit pend;
10      emit checkReq
11    end present;
12    present pend and
13      grant then
14      unemit pend
15    end present;
16    pause
17  end loop
18  ||
19  % Thread "Dispatch"
20  loop
21    present checkReq
22      and free then
23      emit grant
24    end present;
25    pause
26  end loop
27  ]
28 end signal
29 end module

```

(b) SCEst with signals

```

1 module ControlSCEstSig
2   input bool free;
3   input bool req;
4   output bool grant;
5   output bool pend;
6
7   fork
8     _17: grant = false;
9     pend = false;
10    pause;
11    goto _17;
12  par
13    bool checkReq;
14    fork
15      fork
16        _13: if (req) {
17          pend |= true;
18          checkReq |= true
19        };
20        if (pend & grant)
21          {
22          pend = false };
23        pause;
24        goto _13
25      par
26        _15: if (checkReq
27          & free) {
28          grant |= true };
29        pause;
30        goto _15
31      join
32    par
33      _16: checkReq =
34        false;
35      pause;
36      goto _16;
37    join
38  join

```

(c) SCEst with signals transformed to sCL

Listing 6.4.2. The Control example in SCEst with variables, and with signals, including transformation to sCL.

6. Rapid Prototyping

2. SCEst signals can be *re-initialized* to absent, with the newly added unemit statement.

This allows modelling the behaviour of `pend` in Control directly with a signal, without extra delays, signal splitting or variable copying. The resulting ControlSCEstSig code is shown in Listing 6.4.2b. This is also more concise than the original SCL version since `pend` and `grant` need not be explicitly initialized to `false/absent` at the beginning of each tick.

The SCEst compilation follows the model-based compilation approach and compiles SCEst programs to SCL, as discussed in Section 5.1. Listing 6.4.2c shows the result for `control`. The complete transformation rules for all SCEst instructions can be found elsewhere [SMR+17; RSM+15][Rat15]. From here, the usual compilation systems for SCL can be used to generate final products.

6.5 Model Extraction of C Code

Seacord et al. observed that new software is outpacing the ability to maintain it [SPL03, Chapter 1.3]. This motivates to create systems which support the documentation, maintenance and re-usability of software systems and in particular of its legacy code, which is an increasing trend in MDE [IM14]. Chapter 4 already gave pointers on how to use modern pragmatics to increase the efficiency of modellers while they are working on a project.

I propose to use the model-based compilation approach to do this retrospectively by extracting modern models from legacy C code using the model-based compilation approach. These models can then be used for documentation or also to generate new state-of-the-art code for various platforms. This section sketches out how to extract a legacy C program and then use existing compilation chains to create modern code or a circuit.

Finding “the best” visual representation for these models does not seem to be trivial even when considering only relatively simple elements of state-chart dialects. A good balance between compactness, overall overview and simplicity has to be found. Nevertheless, extracted models of complex legacy code can help to understand and maintain these systems. Detailed information about specific compilation rules and investigations towards modelling

6.5. Model Extraction of C Code

```

1 int main(int argc, char**
  argv) {
2   int a, b;
3   if (argc>0) {
4     a = atoi(argv[0]);
5   } else {
6     a = 0;
7   }
8   b = fib(a);
9   return b;
10 }

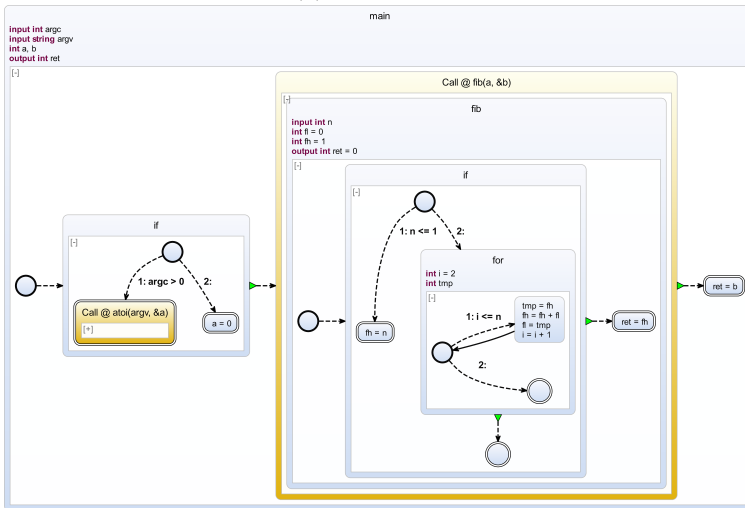
```

```

11 int fib(int n) {
12   int f1 = 0, fh = 1;
13   if (n<=1) { fh = n; }
14   else {
15     for (int i=2; i<=n; i
      ++ ) {
16       int tmp = fh;
17       fh += f1;
18       f1 = tmp;
19     }
20   }
21   return fh;

```

(a) Fibonacci in C



(b) Extracted SCChart of the Fibonacci program in Listing 6.5.1a

Figure 6.5.1. Full Fibonacci example

pragmatics for legacy C programs can be found elsewhere [SLH16] [And19; Len16; Ols16].

6. Rapid Prototyping

Complete Extraction Example A C program serves as input for the model-based compilation. It is represented by its Abstract Syntax Tree (AST), which can be used by subsequent processors to generate appropriate abstract models.

Implementation Since KIELER is an Eclipse framework, the Eclipse C Development Tooling (CDT)³ is available. The CDT is an Eclipse project which serves as a functional IDE for developing applications in C/C++. It parses files of C projects and creates the corresponding ASTs.

Technically, the AST creation via the CDT is encapsulated inside a KiCo processor. The processor takes a C source code file as input and returns an AST. Alternatively, there is also a processor for an AST generation via the GCC. The processors are interchangeable and, therefore, the approach via KiCo is agnostic towards concrete technical implementations.

Example The C program in Listing 6.5.1a calculates the n^{th} Fibonacci number. The extracted SCChart is shown in Figure 6.5.1b. At first, the argument check is performed. If an argument is provided, the function calls the `atoi` system function and converts the string into an integer which is then stored in `a`. Otherwise, `a` is set to 0. Subsequent to the argument check, the Fibonacci function `fib` is invoked. As described previously, this reference is expandable and the structure of this function can be explored immediately. The `fib` function consists of an *if statement* and a *for loop*. The transition in the *for loop* may be delayed or immediate, depending on the selected compilation approach. Eventually, the program returns the requested Fibonacci number.

Following the netlist-based code generation approach, as discussed in Section 5.2, a sequentialized netlist is generated for the Fibonacci program. Two SCGs for both functions, `main` and `fib`, are created. The netlist representation of both SCGs can be seen in Figure 6.5.2. The SCG for the `main` function calls the SCG of the `fib` function. As the netlist-based approach is designed for both, software and hardware, the computation is logically clocked, meaning that the actual computation of the Fibonacci integer n requires n clock cycles. Thus, it is possible to

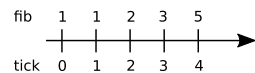


Figure 6.5.4. Clocked computation of the Fibonacci number

³<https://eclipse.org/cdt>

6.6. SCCharts Language Evaluation

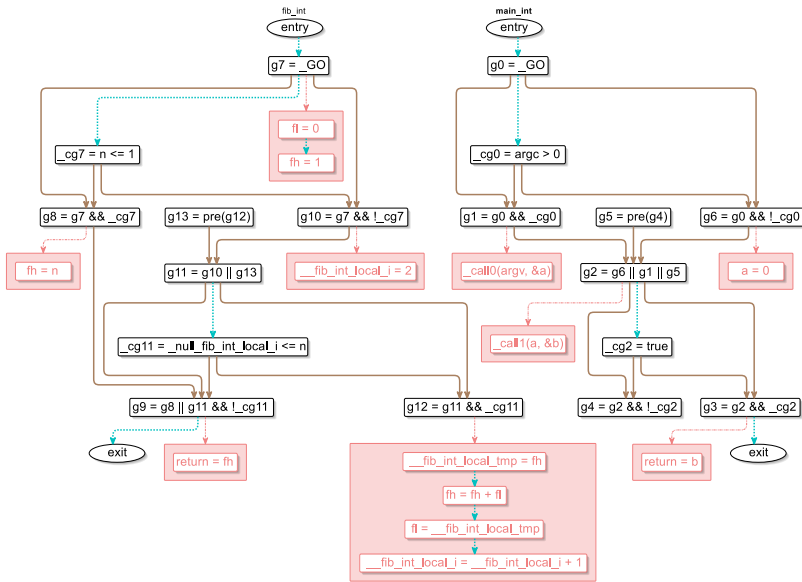


Figure 6.5.2. Generated netlist of the extracted SCChart depicted in Figure 6.5.1b

create both new code (see Figure 6.5.3a) and a circuit (see Figure 6.5.3b) with the same approach. The clocking of the Fibonacci example can be seen in Figure 6.5.4. If not interested in hardware synthesis, e. g. if readability is more important, the previously mentioned software syntheses can also be used for code generation.

6.6 SCCharts Language Evaluation

The SCCharts survey (see [SMS+19a, Appendix A]) was completed for the first time by the participants of the railway project in the summer term 2014 [SMS+15]. It was distributed subsequently on different occasions: All students of the the real-time and embedded systems lectures in the terms winter 14/15, winter 15/16, winter 17/18, and summer 19 as well as students from the synchronous lecture in the winter term 16/17 and the railway project in the summer term 2017 participated. Students from the University

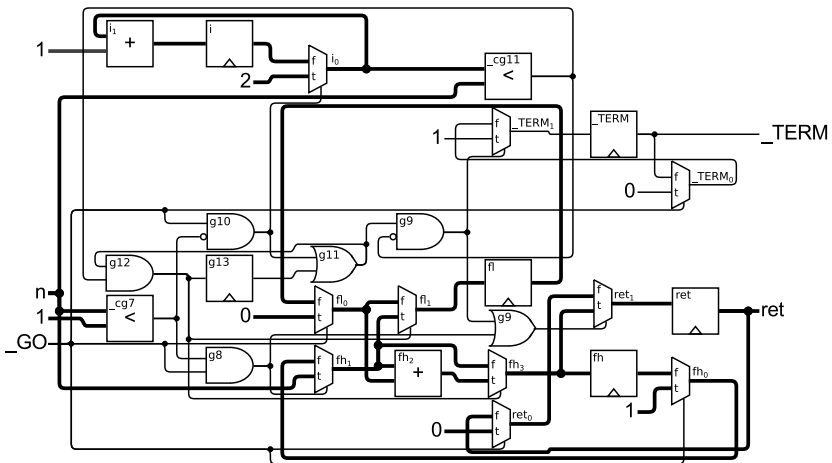
6. Rapid Prototyping

```

1 typedef struct {
2   char _GO;
3   char g7;
4   ...
5 } TickData1;
6
7 void reset1(TickData1 *d)
8 {
9   d->pg12 = 0;
10  d->_GO = 1;
11  d->_TERM = 0;
12
13 void tick1(TickData1 *d) {
14   tickLogic1(d);
15   d->_GO = 0;
16   d->pg12 = d->g12;
17
18 void tickLogic1(TickData1 *d) {
19   d->g7 = d->_GO;
20   if (d->g7) {
21     d->fl = 0;
22     d->fh = 1;
23   }
24   d->_cg7 = d->n <= 1;
25   d->g8 = d->g7 && d->_cg7;
26   if (d->g8) {
27     d->fh = d->n;
28   }
29   d->g13 = d->pg12;
30   d->g10 = d->g7 && !d->
31     _cg7;
32   if (d->g10) {
33     d->_fib_int_local_i = 2;
34   }
35   d->g11 =
36     d->_fib_int_local_i <= d->n;
37   d->g12 = d->g11 && d->
38     _cg11;
39   if (d->g12) {
40     d->_fib_int_local_tmp = d->
41     fh;
42     d->fh = d->fh + d->fl;
43     d->fl = d->
44     _fib_int_local_tmp;
45     d->_fib_int_local_i =
46     d->_fib_int_local_i + 1;
47   }
48   d->g9 = d->g11 &&
49     !d->_cg11 || d->g8;
50   if (d->g9) {
51     d->ret = d->fh;
52     d->_TERM = 1;
53   }
54 }
55 }

```

(a) Generated C code of the netlist in Figure 6.5.2 (excerpt of the fib function)



(b) Generated hardware circuit of the netlist in Figure 6.5.2

Figure 6.5.3. Possible out-of-the-box compilation targets for the Fibonacci example in Figure 6.5.1 in the KIELER SCCharts tools

6.6. SCCharts Language Evaluation

of Auckland completed the survey at the end of their embedded systems class in 2017, and the participants of the Synchron Workshop in 2016 were asked to fill out the survey during the workshop in Bamberg in December 2016. This section evaluates the results of these questionnaires and compares them to the previous results presented elsewhere [SMS+15]. As before, there are three parts that were considered:

- 1. Language Aspects:** In this part the surveys ask general questions about SCCharts and comparisons to other languages. The results are presented in Section 6.6.2.
- 2. Feature Aspects:** This part asks the participants about SCCharts features and their relevance towards their project. The results are discussed in Section 6.6.3.
- 3. Tooling Aspects:** In the third part, the participants were asked to give feedback about the KIELER SCCharts implementation. The results are covered in Section 6.6.4.

There exist some popular programming language rankings, such as the rankings from RedMonk⁴ or the TIOBE Index⁵. A brief summary of criteria, which can be considered when ranking programming languages, can be found in the notes of Prof. Toal⁶. Some of the criteria, such as simplicity, understandability and maintainability, can also be found in the SCCharts surveys. However, the goal of the SCCharts surveys was to make sure that the language and the tooling can compete with other mainstream languages, particularly in the context of embedded systems and teaching, and not to replace any established paradigms.

6.6.1 Survey Setup

The evaluation summarizes the results of the different project groups which completed the same survey. All participants were asked to fill out their survey at the end of their particular project. In the following, the groups are distinguished by the marker shape and color in the diagrams: A diamond

⁴<https://redmonk.com>

⁵<https://www.tiobe.com/tiobe-index>

⁶<https://cs.lmu.edu/~ray/notes/evaluatingprogramminglanguages>, accessed: Dec. 2019

6. Rapid Prototyping

◆ marks a railway project, a circle ● a Mindstorms project, a square ■ a synchronous lecture, a triangle ▲ a external project, and a cross ☒ the survey conducted during the Synchronous Workshop. They are always displayed in chronological order from left to right, additionally indicated by the red-to-blue color gradient.

First Survey — Railway Project, summer 2014 (◆) The first survey of this form distributed out at the end of the railway project in the summer term 14. 7 participants completed the survey and all were pursuing a master's degree in Computer Science.

Second Survey — NXT, winter 2014/15 (●) The second survey was distributed at the end of the real-time and embedded systems lecture in the winter term 14/15, where the participants solved various tasks with the NXT Lego Mindstorm, which are also discussed w.r.t. teaching in Section 7.3.1. There were 21 participants. All of them were Computer Science bachelor students.

Third Survey — NXT2, winter 2015/16 (●) The third survey was distributed at the end of the real-time and embedded systems lecture one year later. Compared to the preceding year, the tasks in this year were more challenging w.r.t. the SCCharts models. In particular, the participants reached the limit of the model sizes which could be uploaded onto the Mindstorm because of resource limitations and unoptimized code. There were 34 students. Most of the participants pursued a bachelor's degree in Computer Science with a few exceptions who had already finished their bachelor's studies.

Fourth Survey — Synchron 2016, winter 2016/17 (☒) The fourth survey was handed out to the participants of the Synchron Workshop 2016 in Bamberg⁷. As an introduction to SCCharts, a interactive tutorial (see [SMS+19a, Appendix C]) accompanied by the SCCharts Cheat Sheet (see [SMS+19a, Appendix D]) was conducted in 1.5h. As last part of the tutorial, the participants were asked to solve the pathfinder task similar to the one that is required of the students participating in the Embedded Systems lecture (also see Section 7.3.1). After the tutorial, an optional shortened survey ([SMS+19a, Appendix B]) could be re-

⁷<https://www.uni-bamberg.de/gdi/synchron-2016>

6.6. SCCharts Language Evaluation

turned. Overall, 8 responses from the Synchron Workshop audience were received.

Fifth Survey — Synchronous Lecture, winter 2016/17 (■) The fifth survey was conducted at the end of the synchronous languages lecture in the winter term 2016/17. While the tasks during the lectures differed from the ones which had to be completed during the Embedded Systems classes, the results of this survey are included for the sake of completeness. 17 participants answered the questionnaires. They were mostly students from the Master's degree program.

Sixth Survey - Railway Project 2017, summer 2017 (◆) While there were only 5 survey participants, they were all graduate students similar to the fifth survey. These results are included for reasons of completeness. Together with the 17 students from the fifth survey and the 7 students from the first railway project, 29 graduate students participated overall.

Seventh Survey — External, summer 2017 (▲) The seventh survey was distributed to a group of students from the University of Auckland. 8 undergraduate students who attended Prof. Roop's Embedded Systems class⁸ participated in summer 2017. As a member of the synchronous community, Prof. Roop is familiar with synchronous languages and their principles. The course did not receive any active support from the KIELER team during the term.

Eighth Survey — NXT3, winter 2017/18 (●) The eighth survey was distributed at the end of the real-time and embedded systems lecture in the winter term 2016/17 with similar tasks as in previous lectures. Most of the 12 students, who participated in the survey, pursued a bachelor's degree in Computer Science.

Ninth Survey — NXT4, summer 2019 (●) The last survey covered in this report was handed out at the end of the real-time and embedded systems lecture of the summer term 2019. The tasks were similar to the previous embedded systems courses. 23 participants, again mostly bachelor students in Computer Science, returned their survey responses.

⁸<https://unidirectory.auckland.ac.nz/profile/p-roop>

6. Rapid Prototyping

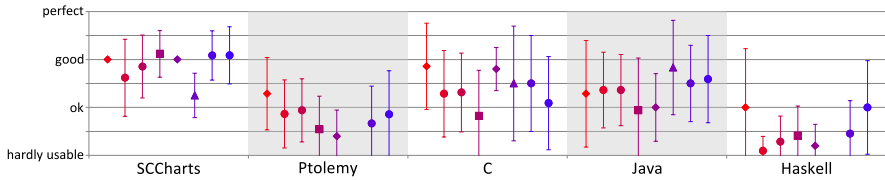


Figure 6.6.1. Language preferences

Overall 135 participants took part in this SCCharts survey over the course of five years. While the team experienced a growth in the stability of the developed SCCharts tools, the different survey groups did not have the exact same starting points and hence, there is no definite reference frame. However, the studies are arguably comparable, especially if relative ratings within a question between different groups match. Surveys of larger groups also support the relative ratings of groups with less participants

In general, in all questions which compare different languages, the synchronous and model languages taught at the department were chosen, namely SCADE, Esterel, SyncCharts, SCCharts and Ptolemy. Additionally, two mainstream imperative languages, C and Java, and one functional language, Haskell, were selected. However, because the first three language choices were relatively unknown to most of the participants, the following results will only include SCCharts, Ptolemy, C, Java, and Haskell. As the students of the Auckland group were not familiar with Ptolemy and Haskell, they are excluded from these comparisons. For Ptolemy, the students of the Real-Time and Embedded Systems classes used the latest stable version of Ptolemy II at the particular times.

6.6.2 Language Aspects

Description As a general question, the participants were asked which languages they find suited for the given tasks. The results depicted in Figure 6.6.1 show that SCCharts as well as C and Java were found suitable. Ptolemy still scored OK whereas Haskell was situated in the lower segment by the students for the tasks focused on designing hardware controller.

6.6. SCCharts Language Evaluation

Results These results position SCCharts at one level with C and Java when it comes to cyber-physical systems. SCCharts are deemed capable to model software for hardware controllers.

Deterministic Behaviour

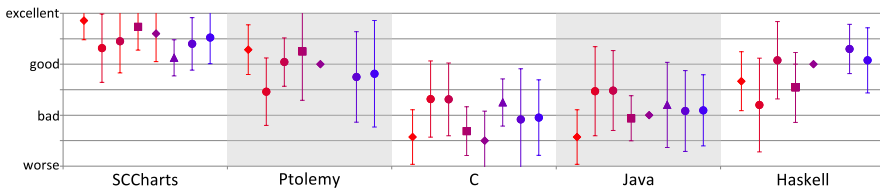


Figure 6.6.2. Deterministic concurrency

Description Figure 6.6.2 presents results for deterministic concurrency. The main question here is how much effort is necessary to avoid race conditions.

Results Naturally, archiving deterministic behaviour is easier with synchronous languages, including SCCharts, because they are deterministic by design. Especially when it comes to race conditions, synchronous languages excel. The rules which synchronous MoCs enforce make it relatively simple to write deterministic and concurrent programs. However, as a consequence the set of programs which are accepted as *constructive* is more restricted than in classical languages. Also, the participants rated concurrent programming in purely functional languages such as Haskell higher, which might be rooted in the fact that these languages employ a side-effect free paradigm.

Note that in contrast to the results of the 2014 survey, the voted grades of the classical programming languages are not as poor as before.

The modeller's experience also plays a role. The ratings of the graduate students are more extreme than the ratings from the undergraduate students. Usually, it is necessary to teach race condition problems to undergraduate

6. Rapid Prototyping

students, whereas concurrency in synchronous languages can be taught relatively easy.

Programming Paradigms

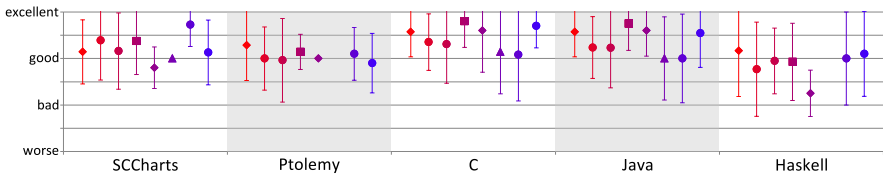


Figure 6.6.3. Sequentiality

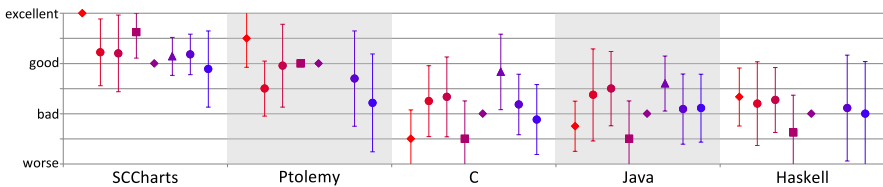


Figure 6.6.4. Separate timing & functionality

Description The next two questions cover sequentiality in programs, with the results displayed in Figure 6.6.3, and the separation of timing and functionality, with the results depicted in Figure 6.6.4. Naturally, writing sequential programs is not particularly difficult in classical imperative programming languages and is a common drawback of synchronous languages. Here, due to the classical synchronous MoC, expressing sequential control flow becomes difficult for a modeller. Closing this gap is a key challenge of SCCharts.

Especially safety-critical systems often require a separation of functionality and concrete timing. Worst-Case Execution Time (WCET) analyses can prove that each reaction of a program can be asserted. This is another strong field for synchronous languages because they are designed in this way.

6.6. SCCharts Language Evaluation

Results The two synchronous languages only have slight disadvantages compared to the imperative languages. However, despite the fact that SCCharts use the sequentially constructive MoC, in comparison to Ptolemy, the ratings are only slightly better with an upwards trend towards the end of the study. Maybe expressing sequentiality is more of a convenience feature in synchronous modelling and not that much of a general issue. Nonetheless, the ability to express sequential pattern arguably helps traditional programmers to transition to the synchronous paradigm, which will be discussed further in Section 6.6.2.

The results in Figure 6.6.4 are less extreme than in 2014, but the trend is similar: Synchronous languages are leading, mostly because they are designed this way. However, in classical programming languages it seems to be notoriously difficult to separate the functionality from the actual timing of the program. Particularly the studies from the larger railway projects confirm this. Also, Ptolemy was rated a mark worse than in 2014, which might have its cause in fewer Ptolemy exercises as the focus of the lecture shifted more towards KIELER as the IDE matured. Nonetheless, with many different *directors* and MoCs in Ptolemy, it might be a somewhat more confusing to separate timing and functionality, even if Ptolemy is considered more powerful than SCCharts w.r.t. MoCs and expressiveness.

Problem Solving

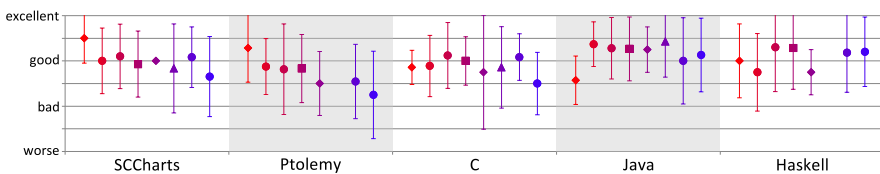


Figure 6.6.5. Solving abstract problems

Description This section presents the results of the questions which language performs better in solving *abstract*, Figure 6.6.5, and/or *low-level*, Figure 6.6.6, tasks. Abstract problem solving focuses on the ability to find

6. Rapid Prototyping

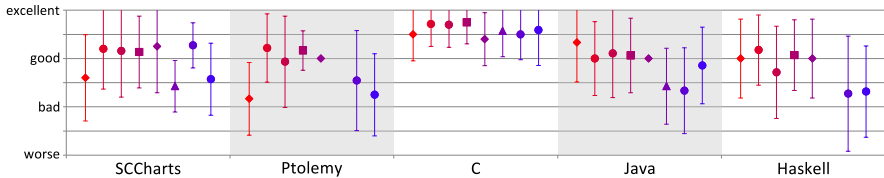


Figure 6.6.6. Solving low-level problems

solutions to problems, such as keeping a pathfinder robot on track (see Section 7.3.1). Contrary, low-level problem solving gives solutions to particular low-level problems, such as how the next three track segments of the railway installation (see Section 7.3.2) be powered up.

Results Towards solving abstract problems, the difference is marginal when compared to C and worse for the synchronous languages when compared to Java. The first railway project is the exception here. In contrast, the results for the low-level ratings are reversed. Here, synchronous languages rate considerably better for solving low-level task for the Mindstorms tasks and worse during the railway project. However, C is the undisputed leader in this comparison when it comes to low-level tasks.

One reason for these ratings may be the task sizes within the different projects. In 2014, the participants were asked to create one large system with a concrete C interface. During the Mindstorms classes, several smaller models were developed. Therefore, picking SCCharts to model the complex railway system in an abstract way might be a good choice, while dealing with the C interface in this context can become cumbersome. However, for smaller models, such as the ones for the Mindstorms, these strengths may become disadvantages because of the MoCs restrictions. The hardware interaction is not so complicated even on model level.

Furthermore, new additions to the KIELER SCCharts tools have improved the capability to interact with hardware directly. The need to use *hostcode* calls was reduced significantly. Therefore, the modeler is free to concentrate on the actual problem even though when dealing with low-level issues. An example was shown in Figure 3.2.2 on page 67.

6.6. SCCharts Language Evaluation

It should be investigated further if the project size directly influences the language preference and efficiency of the developer and how modern model-based developing environments can improve the developer's experience if this is the case.

Language Difficulty

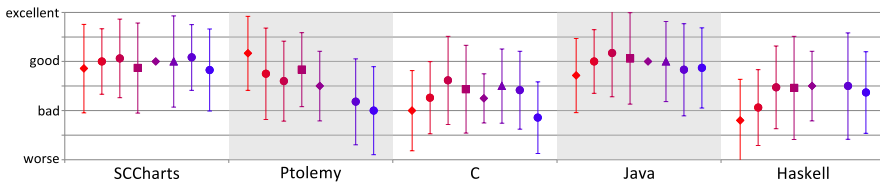


Figure 6.6.7. Understandability

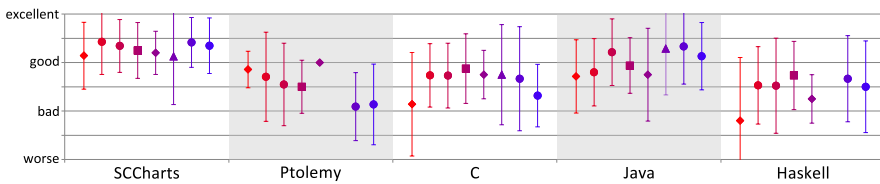


Figure 6.6.8. Simplicity

Description Figure 6.6.7 illustrates the ratings on how difficult it is to read and understand models or programs. Especially when working with large models and/or in teams, it is crucial to keep models understandable.

Figure 6.6.8 shows the votes on how simple it is to learn a certain language in order to fully utilize its features to create comprehensible models or programs. A language which is simple to learn and whose more advanced features are easy to comprehend helps to improve the efficiency of the developers and to maintain an understandable state of the project.

6. Rapid Prototyping

Results Even though the project sizes are different, the understandability and simplicity of the languages are almost equal. In both categories SCCharts was rated to be as good or better than Java, which has the highest rating compared to the other mainstream languages. In comparison to Ptolemy, SCCharts also scored better, but again, here, arguably the reasons can mostly be found in the unfamiliarity of the participants and the higher complexity of the Ptolemy language. The project size does not seem to have a great impact on understandability or simplicity of the SCCharts models. Also, despite the fact that the external student group did not receive direct support if problems occurred with KIELER, their results confirm the results of the department's local students, and therefore support the fact that SCCharts is indeed simple to learn and to understand.

Modularity

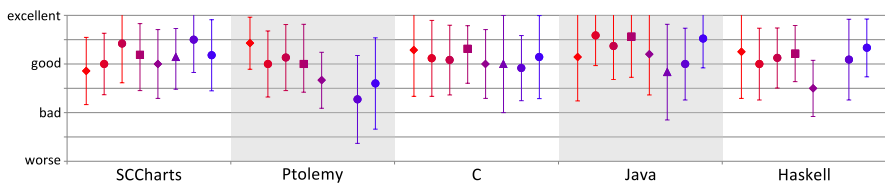


Figure 6.6.9. Composability

Description Figure 6.6.9 depicts the answers to the question how easy it is to compose multiple models/programs to form larger projects. The ability to divide a project into smaller parts enables enhanced structuring and facilitates team development. It is also a key enabler for defining interfaces within the project and for the reuse of components.

Results With the exception of the last surveys w.r.t. Ptolemy, all languages were rated good w.r.t. composability. In modelling languages structuring and navigating projects is not a trivial task. The features of the editor, such as navigating hierarchies through new windows like in Ptolemy or SCADE,

6.6. SCCharts Language Evaluation

shape the developers' experience. It is important for these languages to be on par with classical languages, which basically use the structure mechanisms of the file system or an extension thereof.

In the first railway study, SCCharts were rated worse in comparison to the other languages. However, the relative SCCharts results improved in the later studies, where SCCharts was deemed more on par with the other languages. Here, project size but also KIELER improvements, such as the implementation of an own import mechanism and the abolishment of the Eclipse project natures⁹, might be important factors for the ratings' improvements in later studies; especially compared to the separate window approach used in languages such as Ptolemy.

Project Revisions

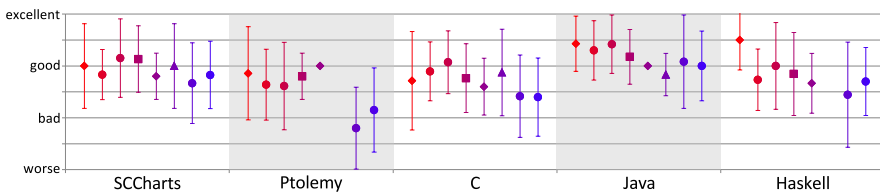


Figure 6.6.10. Maintainability

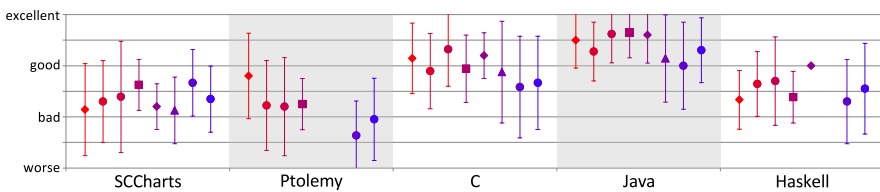


Figure 6.6.11. Debugging

⁹<https://www.eclipse.org/articles/Article-Builders/builders.html>, accessed: 29.05.2020

6. Rapid Prototyping

Description Figure 6.6.10 shows the *maintainability* and Figure 6.6.11 the *debugging* ratings. Both questions ask how difficult it is to change existing models/programs. The subtle difference here is that maintainability aims towards the enhancement and adaptation of an existing model, and debugging tries to find and eliminate errors. While important for all programs, especially embedded systems tend to have long life cycles, and maintenance becomes a crucial aspect over time. Furthermore, the synchronous MoCs are usually more restrictive than classical programming languages, and causality errors may be difficult to find and repair. Therefore, the developer requires more assistance from the IDE to solve these problems.

Results The maintainability ratings for SCCharts, also for the external group, are good and on a par with C. While it still seems to be easier to maintain projects in classical programming languages, the ratings indicate that using a Statechart dialect is a viable solution w.r.t. maintainability. Furthermore, when creating code for different targets or new code from legacy models, e. g. for upgraded hardware of legacy products, the model-based approach should be able to show its true maintenance potential. However, this feature was not covered by the tasks of the conducted studies and remains future work.

In comparison to classical programming, debugging is still a weak spot of synchronous languages and SCCharts. The results are distinctly worse compared to C and Java. Causality problems, e. g. caused by cyclic dependencies, are often difficult to spot during model creation time. Also, not fully supported or broken features, such as arrays for new transformations, may not be as visible as they should be during modelling in such a large academic project. Depending on the project size, maintaining an overview is only possible with good module composition. During runtime, classical breakpoint debugging or assertions are not available in the actual SCCharts tooling.

To address these issues, recent SCCharts research focused on tools for finding issues, such as dedicated views, as presented in Section 4.1 and debugging [Gri16][Eum20], which is sketched out in Section 8.2.2.

6.6.3 Feature Aspects

This survey category discusses the importance of SCCharts language features. For each feature, the survey participants rated how important they deemed that feature w.r.t. relevance for their project. Since the project sizes differed, distinctive features might be of different importance. The railway project required features which facilitate one large project connected to an existing C interface, whereas the NXT projects consisted of several small models which interacted with the Mindstorms interface via the template engine. To structure the features, the aspects are grouped in five blocks, whose results are each displayed in their own figure.

Basic Transition Features

Figure 6.6.12 shows essential transition features, such as priorities, triggers, and effects.

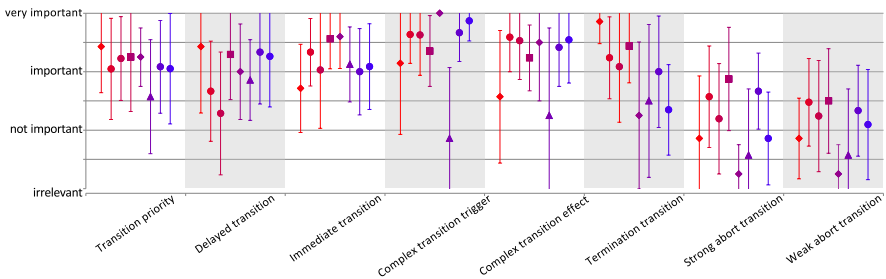


Figure 6.6.12. Essential transition features

Delay behavior In the Mindstorms projects, delayed transitions played a less important role whereby immediate transitions were used more often. Especially the more complex Mindstorm tasks often required instantaneous calculations, such as PID controller results, or decision making, e. g. via decisions trees. There is no state required and therefore consumption of time can be avoided. These tasks often did not require complex concurrency, and therefore avoided tick consumptions which might had been necessary due

6. Rapid Prototyping

to causal relationships. Today, the SCCharts dataflow extension, discussed in Section 6.2, gives the modeller the opportunity to model in a more hybrid-like style. Instantaneous calculations can be modelled naturally in dataflow, whereas states take advantage of the Statechart's way.

Trigger and effects Especially when working with large models and dealing with many effects, sequences of transition effects make the model less readable. As can be seen in Figure 6.6.12, the railway team used fewer transition effects than the Mindstorm teams. They compensated the lack of effects on the transitions with entry actions in states (Section 6.6.3). This results in semantically equal behaviour but displays the models in a more compact way. Modern pragmatic features, such as label management [Sch19], can also help to keep the model size reasonable.

Preemption To avoid scheduling issues in the railway projects, superstates were left via normal terminations in most cases. Preemption was used rarely. Since then, the abort transformations were improved. Preemption was used more often in the Mindstorm projects. Better overview due to the smaller project size might also have contributed to this decision. This strengthens the argument that dedicated preemption mechanisms in the SCCharts kernel language are not necessary. However, it is still an open question if such kernel additions are beneficial to ease the downstream compilation and if they improve the readability of the generated code.

History, Suspension and Actions

Figure 6.6.13 shows the participants' results regarding history and action features.

History All history-related transition features were not deemed important for the given tasks. However, they are popular in other model-based languages, such as SyncCharts and Ptolemy. They have a moderate internal complexity in SCCharts (see Section 4.3) and could be excluded from the compilation chain to simplify the compilation when it comes to similar tasks.

6.6. SCCharts Language Evaluation

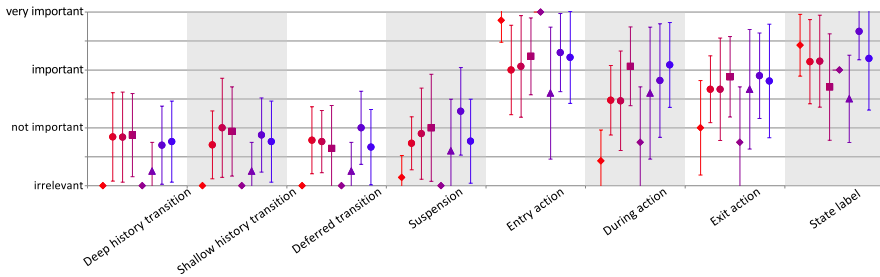


Figure 6.6.13. History and local action features

Suspension Equal to the history ratings, suspension features were not required. It is noteworthy that suspension is part of Esterel’s kernel language [PEB07] but not deemed necessary in SCCharts for the given tasks.

Actions As previously mentioned in Section 6.6.3, since the beginning of the use of SCCharts in teaching, users tend to write actions (especially entry actions) to model effects, mainly to keep the diagram sizes reasonable. This trend continued, although the need might not be as urgent for the Mindstorms projects due to the smaller model sizes.

More recent projects made more use of during and exit actions. Again, smaller project sizes, the tasks themselves, and compiler improvements, which nowadays support concurrent actions better, make the use of these features more viable. It is an interesting question if this way of modelling should be adapted as the main way of effect emission. In this case, the transitions themselves would only comprise triggers and the semantic duality of these approaches could be abolished.

Concurrency, Declarations and Data Types

Figure 6.6.14 shows the participants’ results regarding concurrency, declarations, and different data types.

Concurrency As one decisive feature of synchronous languages, concurrency was rated very important. While it may not be of paramount impor-

6. Rapid Prototyping

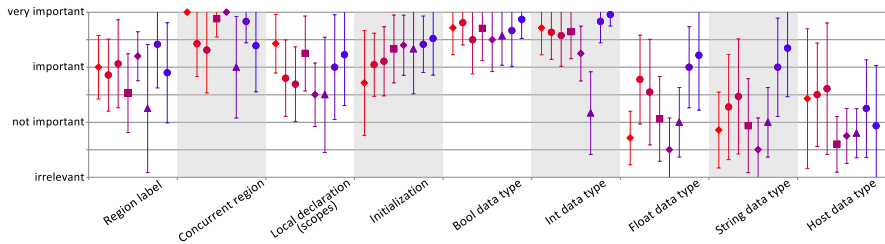


Figure 6.6.14. Concurrency, declarations, and data types

tance compared to the railway projects where eleven trains must perform the same tasks concurrently, there are also a multitude of task which can be solved concurrently in the Mindstorms' settings. After all, deterministic concurrency is one of the defining pillars which justifies the existence of synchronous languages.

Declaration Scoping, together with local variables, and the possibility to initialize these variables were also rated important. These convenient structuring mechanisms come at low cost for the compiler and help to keep the project maintainable.

Data Types While boolean and integer data types were necessary from day one, floating point and string data types gained popularity over the course of time. The particular data type usage is use-case specific but especially calculation-heavy program parts, such as PID controllers, often need fractions. In the Mindstorms setting, the string type is handy to give immediate feedback because the Mindstorm unit has a display. This is not the case in the railway setup, where string support is only needed for logging operations. New wrapper and deployment capabilities in the KIELER compiler also made the host data type less mandatory over time.

Additional Extended Features

Figure 6.6.15 shows the participants' results for additional extended features, such as *Count Delay*, *Signals*, and *Referenced SCCharts*.

6.6. SCCharts Language Evaluation

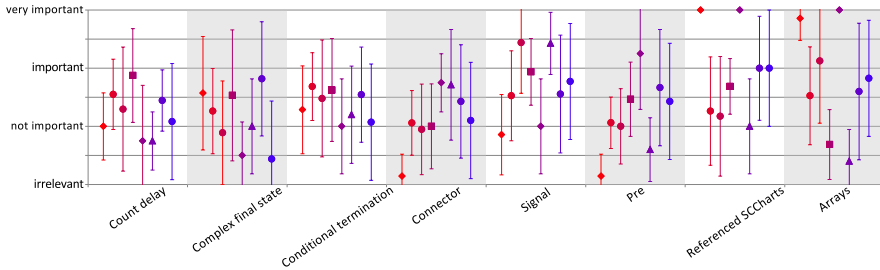


Figure 6.6.15. Additional extended features

Count Delay An arguably important feature in synchronous languages in general is *count delay*. A count delay with an integer n delays a transition firing until the trigger condition evaluated to true n times. Though it was always a part of the SCCharts language set, the handling, especially when working with several count delay incarnations with the same signal, e. g. in a concurrent context, sometimes becomes complicated. Depending on the actual model, this can lead to circular dependencies, which are hidden due to the extended nature of the feature. New scheduling possibilities, such as the SDs introduced in Section 6.1, and transformations that solve these problems are now part of KIELER SCCharts (cf. the SCCharts timeline in Figure 2.2.2 on page 25). While it is sometimes handy to just state a count delay, e. g. to count seconds, for the given tasks, the count delay rating was below important.

Complex Final State In contrast to *simple final states*, which terminate a region as soon as they are reached, *complex final states* are final states which have inner behaviour or can be left again. The region is only terminated if all concurrent regions of the parent superstate reach a final state. This is not a core feature of SCCharts and the extended transformation has a high internal complexity (cf. Section 4.3 on page 110 ff.), which makes it not viable, e. g. for the Mindstorms, due to the resource limitations as discussed in Section 7.3.1.

6. Rapid Prototyping

Conditional Termination A *conditional termination* can only fire, so that the originating superstate is left, if its trigger evaluates to true. While this feature can be useful, it was not deemed to be mandatory.

Connector Although thought to be useful for describing certain flows in documentation, the *connector* feature did not seem to have any practical relevance in the projects. However, it is usually used to increase the readability and hence, the maintainability of the diagram. It also comes at low cost for the compiler.

Signal A *signal* is another prominent and mandatory feature of many synchronous languages. Usually, signals are necessary for concurrent communication. However, they are not mandatory in SCCharts as they can be fully emulated with boolean variables and the scheduling regime of SCCharts. This explains the high rating of the external student group, whose lecture focused on classical synchronous languages and not SCCharts-specific characteristics. Nonetheless, also the other groups found signals with their intrinsic absence semantics useful in SCCharts.

The emulation of signals in SCCharts introduces extra concurrent regions with implicit during actions, which complicate the compilation especially in combination with the abort transformation. They were used more often in the smaller projects and avoided in the railway projects. Implemented optimizations might have contributed to the fact that they are more widely used in later projects.

Pre Indispensable in classical synchronous languages, *pre* is not as important in SCCharts. While this might also depend on the particular tasks (the external group also rated *pre* unimportant), SCCharts mainly uses variables, which store their value even across tick boundaries. Furthermore, as *pre* is an extended feature in SCCharts with a moderate internal complexity (cf. Section 4.3 on page 110 ff.), this can lead to resource issues in resource limited platforms.

Referenced SCCharts *Referenced SCCharts* are SCCharts' main mechanism to support modularity. SCCharts can reference other SCCharts models and

6.6. SCCharts Language Evaluation

include their behaviour. While still important for structuring (see also the results for modularity in Section 6.6.2), the Mindstorms tasks' project sizes would also allow for models without referenced SCCharts, whereas the feature made the railway project practical in the first place. This is a confirmation that modularity is always good and mandatory for larger projects.

Arrays The specific use-case also dictates whether arrays are mandatory or not. The C API and especially the need to address many peripheral devices in the railway projects required arrays to work efficiently. This was not necessary when working with the Mindstorms. Furthermore, template engines on the deployment layer (cf. Section 3.2.1 on page 65 ff.) make direct host interaction often obsolete.

Future Features

In the later surveys, we asked the participants, which possible future features, which were not or not fully implemented at the time of the participation, they deemed important. Figure 6.6.16 shows the participants' results. Overall the participants' results in this category seem to be slightly indecisive. However, the answers are still evaluated relative to each other within one survey group to see which features were rated more important than others.

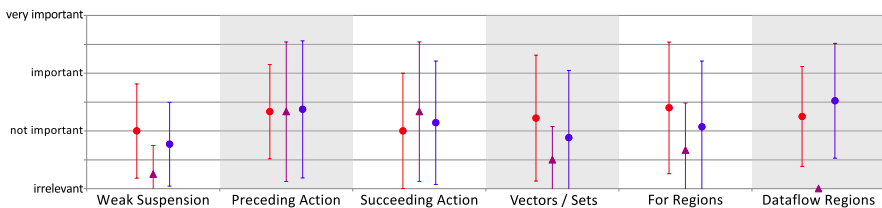


Figure 6.6.16. Future features

Weak Suspension A *weak suspend* is implemented in languages such as Esterel V7 and in a limited version in Quartz. It allows the actual tick to finish but restarts the current tick anew in the next tick. This feature was rated unimportant in the projects. Furthermore, as the simple suspend is

6. Rapid Prototyping

rarely used in the given tasks, this feature is most likely not required in SCCharts.

Additional Actions Like SyncCharts, SCCharts handle entry actions and exit actions asymmetrically w.r.t. preemption. To give the modeller the possibility to decide whether or not an action should be preemptable, two new actions could be added. The participants' ratings are indecisive but with a high standard deviation. These additional actions are now part of the KIELER SCCharts distribution, as discussed in Section 6.3.

Vectors / Sets Vector assignments should ease the usability of array and large data structures. However, the low ratings and recent imperative coding style developments, e. g. imperative loops (see Section 8.2 on page 331), might deem this addition unnecessary. It is still used in the dataflow extension, as explained in Section 6.2, as a convenient syntax feature to set all inputs at once.

For Regions For regions can duplicate region behaviour, either for a specified range or an array. This convenient feature assigns an iterator variable to each duplicated region. While not critical, this feature saves modelling time and can increase readability and maintainability. It is now part of Extended SCCharts as discussed in Section D.3.

Dataflow Regions DFRs allow direct computations within a region. This enables the modeller to create hybrid SCCharts and reduces the need to model immediate transition chains for computations without real state change. Equations can be modelled in DFRs, whereas state-based control-flow use traditional node-link graphs. They are now part of KIELER SCCharts, as discussed in Section 6.2.

6.6.4 Tooling Aspects

In the next set of questions of the survey, the participants should rate the overall quality of the SCCharts tools in the actual KIELER implementation.

The tooling category also includes the results of the synchronous survey group ■ from the Synchron Workshop 2016 in Bamberg.

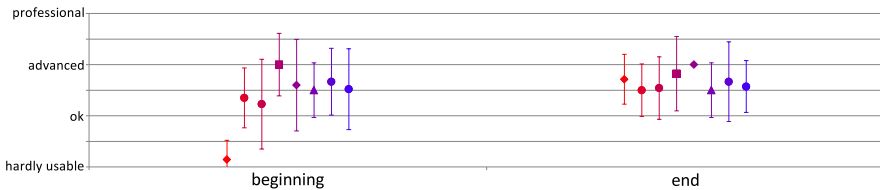


Figure 6.6.17. Tool quality

Overall Quality of SCCharts Tools Figure 6.6.17 compares the overall quality of the tools at the beginning of the projects with the end of the projects. The SCCharts tools were completely new when the first railway project began. They performed badly in larger projects especially because there was no matured concept for modularity. As the tools improved and with the addition of the *referenced SCCharts* feature, as explained in Section D.1, the rating increased significantly.

The Mindstorms projects did not suffer from these issues at the start of the projects. Moreover, they benefited greatly from the lessons learned during the railway project. Of course, the participants of the later projects did not have this perspective. However, even comparatively small changes in the usability are visible in the figure as the final rating also improved marginally during the department internal projects. Overall the SCCharts tooling is rated between ok and advanced at the end of all projects. While the team is satisfied with the way this large academic product matured, there is still room for improvements.

Model Creation and Debugging

When it comes to model creation and debugging, Figure 6.6.18 shows that the discrepancy between small and large models was rated more extremely in the railway projects. Naturally, what was seen as a large model differs within the project groups. However, all participants agreed that debug-

6. Rapid Prototyping

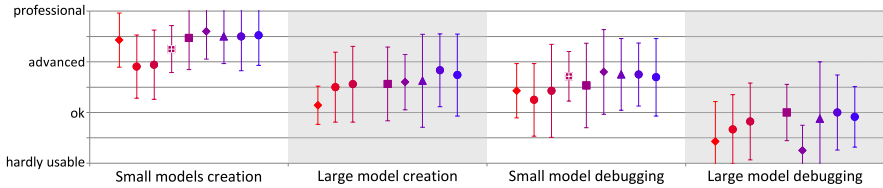


Figure 6.6.18. Model creation & debugging

ging, particularly of large models, is difficult. As already established in Section 6.6.2, debugging remains a point of possible improvements for SC-Charts. However, the relative trends indicate that the tooling improved over time. This suggests to shift the focus even more on usability, maintainability and debugging features.

Further Tooling Aspects

Figure 6.6.19 shows several aspects of the tooling. It is noteworthy that the ratings from the professional group are generally higher than the ratings of the student groups. This can be contributed to the fact that the professional participants have more experience in working with similar problem tasks and comparable tools. They also may have a deeper understanding about the issues which need to be solved and about the available alternatives.

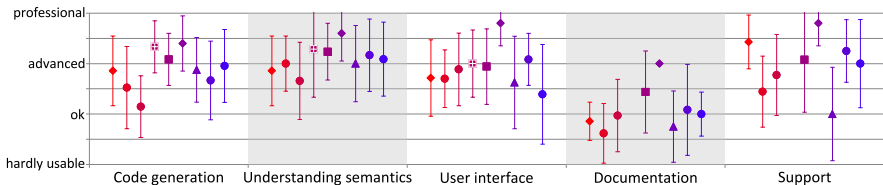


Figure 6.6.19. Tooling aspects

Code Generation The code generation was rated worse in the first two Mindstorms projects, which can be attributed to the resource limitations and the lack of proper code optimizations in the early stages of the KIELER project.

6.6. SCCharts Language Evaluation

In the beginning, SCCharts were quite generous when it came to the usage of guard variables. Especially larger models, such as the Barcode Reader (see Section 7.3.1), resulted in larger programs which were problematic for the firmware used. In later versions, several optimizations, as introduced in Section 5.2.8, were added to make larger models executable on resource-scarce systems. Overall, the code generation was deemed around the advanced mark. The optimization of the generated code remains an open topic, especially when working with limited resource systems, since more powerful optimizations, such as the SCCP, can be used in the compilation chain by sacrificing some of the understandability between the transformation steps.

Understanding Semantics A clear understanding of the semantics of language features is important. Since all ratings are above average, the tooling provides clear representations of the features in use and the processes involved during model creation and compilation are comprehensible.

User Interface The rating of the user interface was almost the same in all projects. There is a distinct drop in the ratings in the last study group. A reason for this might be that some students in this lecture run did not complete the initial tutorial. This resulted in problems later on since they faced errors as a result of their missing firmware or Java installation. Another reason for it might be the recent popularity of other IDEs, such as VSCode¹⁰ and IntelliJ¹¹, which use different, arguably novel, UI concepts than the Eclipse-based framework. Active research [Dom18; Ren18] in this area examines these pragmatic questions.

Documentation Although the documentation improved over time, this is still another weak spot of the SCCharts project. Documentation and examples are present and have been expanded, especially before the last iteration of the embedded systems class. However, the latest improvements do not seem to have a big impact on the ratings and are not enough to give a better

¹⁰<https://code.visualstudio.com>

¹¹<https://www.jetbrains.com/idea>

6. Rapid Prototyping

than ok impression. More extensive documentation and/or better ways of presenting the actual state of the project should be explored in the future.

Support Naturally, the in-house projects scored better in the support ratings. The latest improvements within the KIELER project, such as the template engine and more pragmatic features, increased these ratings again over time.

As tooling conclusion, further KIELER development should focus on three areas.

Debugging While improved over time, debugging is a weak spot of SCCharts and possibly statecharts dialects in general and therefore should be improved further.

Common Coding Paradigms To increase the reach and acceptance of statecharts-based dialects, synchronous languages should partly adopt accepted coding paradigms, such as OOP. Sequentiality is a cornerstone in the mindset of most programmers and should be supported inherently. Concurrency should be exploited where conducive but should not be mandatory to solve simple tasks. Modelling notations should fit the modellers needs, e. g. dataflow for equations and states for control-flow.

Modelling Pragmatics Designing a language according to the previously mentioned principles is only half of the story. The language must seamlessly integrate with a modelling framework which enables an efficient workflow. Automatic diagram syntheses and filtering techniques for pragmatic navigation are paramount.

Section 8.2 outlines currently active research areas in the context of KIELER SCCharts. Exotic features, such as weak suspend, may be of less importance and occupy development time. If necessary, they can be abandoned. Nonetheless, the KiCo makes it also possible to quickly prototype and experiment with new or exotic features.

Practical Applications

The most important property of a program is whether it accomplishes the intention of its user.
— Tony Hoare

This chapter gives further insights towards the usefulness of statechart dialects in practical industry, academic and teaching projects. Section 7.1 shows how SCCharts statemachines can be generated automatically from high-level specifications, such as Excel spreadsheets, by adding a single processor to the standard compilation chain. Furthermore, model checking on SCCharts models within the KIELER framework will be exemplified. Section 7.2 shows how the established language methodologies can be used to implement *run-time enforcement*, which can be used to create and deploy models for safety-critical devices, such as pacemaker devices. These devices are required to be fault secure. Therefore, deterministic synchronous languages make a good fit. However, they do not always come with the intuitive setting general purpose programmers would expect or need to fulfil the task at hand easily. Run-time enforcement is a way to automatically generate bidirectional enforcers according to a specified *safety automaton*. Finally, SCCharts have been used in class to teach students principles about real-time, embedded, and safety-critical systems over the last ten years. Section 7.3 gives an overview over four students projects which used the KIELER tools and adapted them to their needs. The projects, namely Lego Mindstorms, Railway installation, Quadrocopter and Raceyard, range from small scale tasks during semester, to tasks which require a total term.

7. Practical Applications

Spec #	State	Trigger 1	Trigger 2	Action	Target state	Comment
DiB_01	Dispatch	MTLd	object == TRAIN	clear()	Select	Example: XY A7 / A8
DiB_02	Dispatch	MTLd	object == CABOOSE	openWindow(object)	Select	111.1001-210
DiB_04	Dispatch	MTRd	object == LAYER	openWindow(object)	ElementPopup	
DiB_04a	ElementPopup	MTRd	object == NULL		Dispatch	TLM considered as null
DiB_04b	ElementPopup	Move	object != NULL	clearSelection()	ElementPopup	
DiB_07	ElementPopup	MTLd	object == selected	closeWindow(object)	CW	
DiB_12a	ElementPopup	MTLd	receipt == NULL	closeWindow(object)	Dispatch	
DiB_12b	ElementPopup	MTRd	receipt == NULL	closeWindow(object)	Dispatch	
DiB_19	ElementPopup	MTLd	type = MISC		Dispatch	
DiB_20	Select	MTLd	object != NULL	F = false	Select	
DiB_21	Select	MTRd	object != NULL	F = false	Target	
DiB_23	Select	MTRd	obj-ect == selected		Target	
DiB_24	Select	MTLd	object == NULL		Dispatch	111.101010.910
DiB_25	Select	MTRd	object == NULL		Dispatch	000.1111-001
DiB_34	Target	MTLd	object == NULL		Dispatch	111.101010.910
DiB_35	Target	MTRd	object == NULL		Dispatch	111.101010.910
DiB_40	Target	MTRd	object == selected	openWindow(object)	Target	
DiB_42d	Target	MTLd	object == selected	closeWindow(object)	Target	000.1111-001
DiB_47	CW	MTLd	object == NULL		Dispatch	111.101010.910
DiB_52	Select	TIMEOUT		closeWindow()	Dispatch	000.1111-001
DiB_53	Target	TIMEOUT		closeWindow()	Dispatch	000.1111-001
DiB_56	ElementPopup	TIMEOUT		closeWindow()	Dispatch	000.1111-001

Table 7.1.1. Excerpt of a high-level specification for a user interface in the railway domain written in a common spreadsheet software; the concrete functions are anonymized for confidentially reasons.

7.1 Specification and Verification

Specifications often come first in large software projects. Section 7.1.1 demonstrates how high-level software specification can directly be turned into a statemachine for documentation reasons and are eventually used to generate the necessary code automatically using the techniques discussed in the previous chapters. To verify the functionality of the specification, model checking can be used to prove correct behaviour, which is demonstrated in Section 7.1.2. These *transient specifications* and model checking fit to the iMURD methodology and help the user to retrieve a bigger picture. They also help to spot errors in the specification early.

7.1.1 Transient Specifications

It is not uncommon for project specifications to be assembled in third-party spreadsheet software by project team leaders. These specifications are then turned into processes or—often directly programmatically—into statemachines, which in turn also need to be synthesized graphically as documentation. This process can be automatized by adding a single processor to

7.1. Specification and Verification

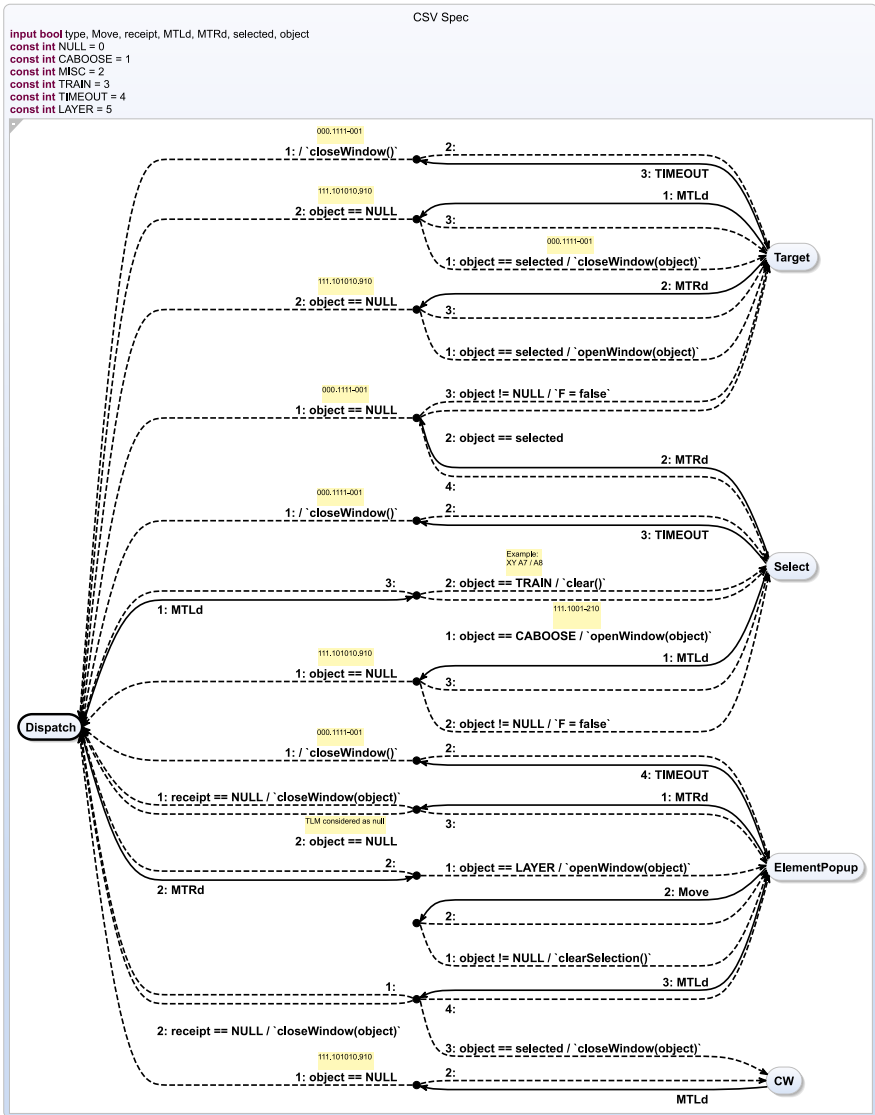


Figure 7.1.1. Automatically generated SCChart for the high-level specification shown in Table 7.1.1

7. Practical Applications

the compilation chain that transforms a high-level spreadsheet specification into statechart meta-model, e. g. SCCharts.

Example Table 7.1.1 shows an excerpt of industrial high-level specification for a user interface in the railway domain. While the concrete functions are anonymized for confidentiality reasons, basic functions, such as MTL for the left mouse button, can be identified. Usually this specification has to be turned into code manually and simultaneously the documentation has to be created. Fundamentally, the spreadsheet encodes a flat statemachine with different states and transition conditions. The columns hold information (from left to right) about the original specification number, the actual state, the first and second transition trigger, an optional action, a target state, and a comment about the specification. The sheet was already structured this way when sent to the KIELER team by an industrial user, so statemachine like definitions were already used. A spreadsheet structured this way can easily be transformed into an SCChart with a KiCo processor.

Implementation The processor used to generate the graphical version of the specification seen in Figure 7.1.1 is 256 lines long. States are created and connected according to the state and transition information. Additionally, the comments can be added to transitions and add to the documentation. This diagram instantly serves as visualization for the team lead. From here, the SCChart model serves as original source model for subsequent compilations or further verification, e. g. by model checking, which is discussed in the following section.

7.1.2 Model Checking

Model checking is a powerful tool to prove properties of a model. Usually, modern model checkers allow properties to be expressed in some form of logic. Temporal logics are an extension to propositional logic. They add an abstract ordering to events. A well-established temporal logic is the Linear Temporal Logic (LTL). LTL formulas are described by

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid G\varphi \mid F\varphi \mid \varphi_1 U \varphi_2 \mid X\varphi.$$

U (until) states that φ_1 holds until φ_2 is satisfied. X (next) refers to the next event. G (globally) means that the proposition holds always, and F (finally)

7.1. Specification and Verification

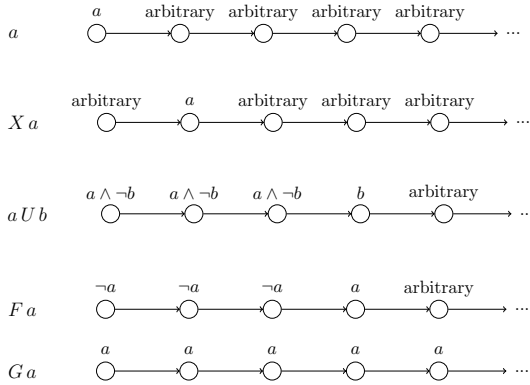


Figure 7.1.2. Illustration of the LTL semantics [BK08]

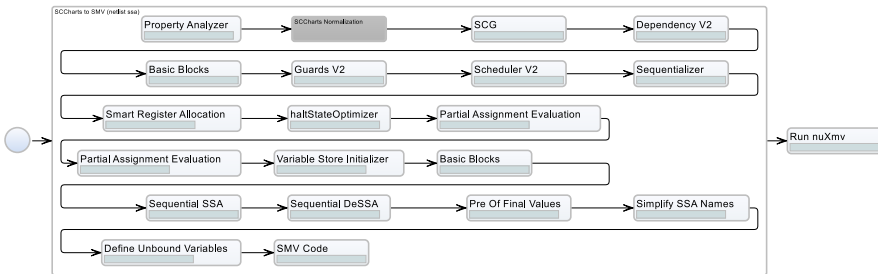


Figure 7.1.3. NuSMV compilation system in KIELER

at some time in the future. Formally, they can be defined as $G\varphi = \neg F\neg\varphi$ and $F\varphi = trueU\varphi$. Baier and Katoen give a formal definition of the LTL [BK08]. A graphical illustration of the temporal operators is shown in Figure 7.1.2.

To experiment with different kinds of model checkers within the KIELER project, Stange implemented KiCo compilation chains for the model checkers SPIN¹, NuSMV² and nuXMV³ [Sta19]. SPIN is one of the oldest model

¹<http://spinroot.com>

²<http://nusmv.fbk.eu>

³<https://nuxmv.fbk.eu>

7. Practical Applications

checkers. It constructs a *verifier* in C code, which then checks for the property in a constructed state space via BFS or DFS. nuXMV is a symbolic model checker which extends the open-source NuSMV, which uses Binary Decision Diagrams (BDDs). Additionally, nuXMV supports a range of other model checking algorithms, such as satisfiability solvers [CCD+14].

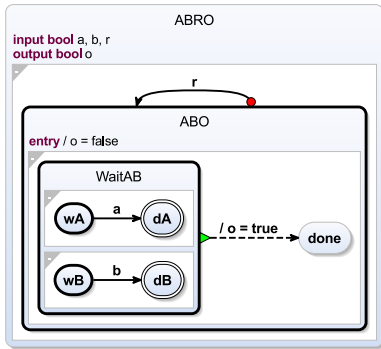
According to the KiCo workflow presented in Section 3.1, an SCCharts model can be transformed into languages the model checkers can understand. A KiCo processor then runs the corresponding model checker and retrieves the result, which can then be visualized in KIELER. In this instance, the result either confirms the stated properties or gives a counter-example, which is converted into a runnable trace file in KIELER. This trace can be loaded into the simulation engine, to visualize the counter-example. As an example, the compilation system for NuSMV is shown in Figure 7.1.3.

Example Figure 7.1.4a shows an ABRO model with variables. Model checking properties can be defined for ABRO's behaviour and checked automatically. Figure 7.1.4b shows three examples: The invariant states that o cannot be present if r is present. The first LTL property says that in any tick (G) if the next tick (X) has a, b, and not r, then o will be set. Finally, the second LTL property states that o will never be true.

These kinds of properties can be annotated to any SCChart and then run with KIELER. Figure 7.1.4c shows the model checking results of the aforementioned properties. The first two properties pass the check, whereas the last one fails. The result for the last property, shown in Figure 7.1.4d, is given back to KIELER where it is turned into a simulation trace. The running simulation, see Figure 7.1.4e, shows that o can indeed be emitted.

Model checking should be promoted further in KIELER as it strengthens the argument for using modelling languages in general. It can be also used to teach model checking principals and temporal logic. The interested reader can explore more in-depth technical details about model checking for SCCharts in KIELER elsewhere [Sta19].

7.1. Specification and Verification



(a) ABRO with variables

```

1  -> State: 1.1 <-
2  a = FALSE
3  b = FALSE
4  r = FALSE
5  o = FALSE
6  -> State: 1.2 <-
7  a = TRUE
8  b = TRUE
9  o = TRUE

```

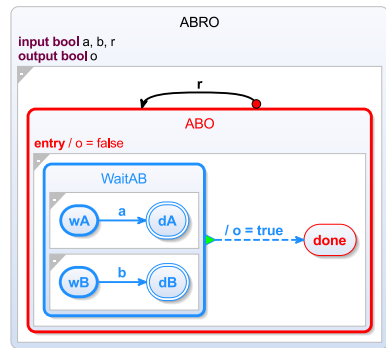
(d) ABRO $G!o$ counter-example returned from the model checker

- ▷ Invariant property: $r \rightarrow !o$
- ▷ LTL property: $G X ((a \ \& \ b \ \& \ !r) \rightarrow o)$
- ▷ LTL property: $G !o$

(b) Model checking properties

Name	Formula	Result
$r \rightarrow o$	$r \rightarrow !o$	PASSED
$G X a \ \& \ b \ \& \ !r \rightarrow o$	$G X ((a \ \& \ b \ \& \ !r) \rightarrow o)$	PASSED
$G !o$	$G !o$	FAILED

(c) Model checking results for ABROs properties within KIELER



(e) KIELER simulating the ABRO $G!o$ counter-example

Figure 7.1.4. Model Checking for SCCharts

7. Practical Applications

Spec #	State	Trigger 1	Trigger 2	Action	Target state	Comment
YNi_01	Reference	btime	nullair		NullAir	
YNi_02	Reference	btime			Sample	
YNi_03	Sample	btime	yes & laststep == step + 1	samplestep = laststep; step -	Reference	
YNi_04	Sample	btime	yes	laststep == step; step -	Reference	
YNi_05	Sample	btime		step -	Reference	
YNi_06	NullAir	btime	yes	nullerrors+ +	Reference	
YNi_07	NullAir	btime			Reference	
YNo_01	Cycle	nullerrors > 1		nullerror = true; done = true	Done	
YNo_02	Cycle	samplestep > 0		validanswer = true; done = true	Done	
YNo_03	Cycle	step == 0		done = true	Done	

Table 7.1.2. Behaviour of an EN 13725 panelist cycle in a y/n odour determination measurement: The specification above the delimiter describes the inner behaviour of the cycle; the specification below the outer.

7.1.3 Example—DIN EN 13725

This section gives a full example on the transient specification and model checking verification topic. The DIN EN 13725⁴ is an industry norm for the determination of odour concentration by dynamic olfactometry and odour emission rate from stationary sources. Here, it serves as complete specification and model checking example without the need of any manual code written by a programming expert.

In the *yes/no method* of the norm, selected panel members are given alternating air to breath. In a 2.2s rhythm, a panel member cycles through 4 phases: *reference air, breathing out, sample air, breathing out*. The concentration of the sample is increased with every cycle. If the panel members can recognize a difference between the reference and sample air, they give a *yes* answer. After two consecutive yes answers, the actual concentration step of that panel member is taken to calculate an individual odour value. However, to check for false positives, there is a possibility that the sample air in one cycle is replaced by so called *null air*. So, a simple sample run for a panel member could end if (1) they gave two correct consecutive yes answers, (2) made two null air errors or (3) received all sample cycles without any yes responses or errors. The DIN EN 13725 also states that there cannot be two consecutive null samples.

The textual specification given can be written in a spreadsheet style presented in Section 7.1. Table 7.1.2 gives a basic specification for a panel mem-

⁴Retrievable via <https://www.vdi.de>

7.1. Specification and Verification

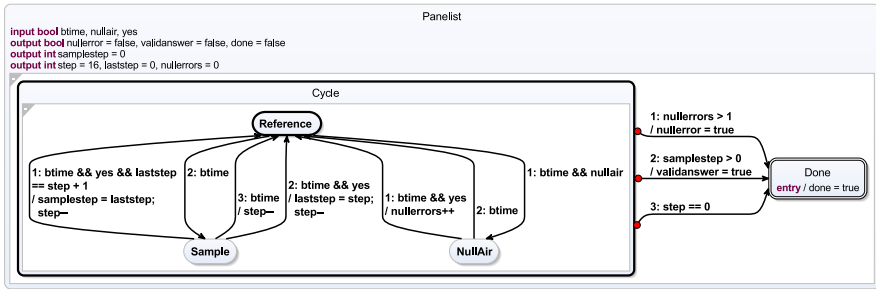


Figure 7.1.5. Diagram of a panel member cycle for DIN EN 13725 limit y/n

ber cycle during a y/n odour determination measurement. The specifications above the delimiter describe the inner behaviour during the measurement, whereas the lower part specifies the abort criteria of the higher hierarchy telling the system when the cycle is complete. The flag *btime* is a boolean flag which signals the breathing rhythm, *nullair* is an input that signals that the upcoming cycle will be a null air cycle. Furthermore, *step* is the actual step; descending in every iteration. If the panel member gives a yes answer, *yes* will be true. However, the rule that two null air cycles cannot follow each other is missing from this example. The diagram for this description is depicted in Figure 7.1.5

The specification can be complemented by model checking properties. For the specified measurement, two simple checks are the LTL formula

$$G \text{ btime} \rightarrow F \text{ done}$$

and the invariant

$$\text{done} \rightarrow (\text{nullerror} \mid \text{validanswer} \mid \text{step} == 0).$$

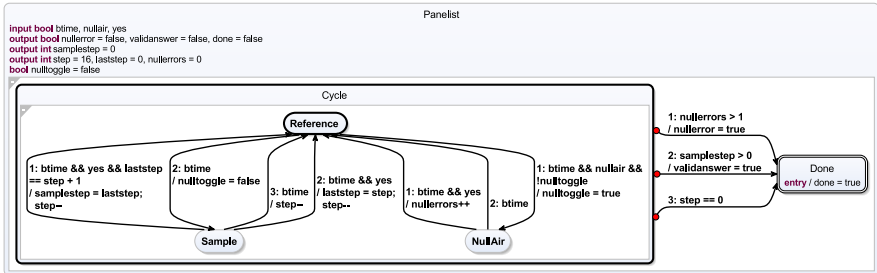
The checks ensure that (1) as long as a breathing input is given done will be reached eventually and (2) the only way to end the measurement is due to one of the specified ways: too many incorrect answers, a valid answer, or the last step is reached.

However, the model checker gives a FAILED response for the first check, which hints at an error in the textual specification. The checker also returns

7. Practical Applications

Spec #	State	Trigger 1	Trigger 2	Action	Target state	Comment
YNI_01	Reference	btime	nullair & !nulltoggle	nulltoggle = true	NullAir	
YNI_02	Reference	btime		nulltoggle = false	Sample	

(a) Corrected specification according to the done model check



(b) Corrected diagram of the panel member cycle from Figure 7.1.5

Figure 7.1.6. Corrected behaviour of the DIN EN 13725 panel member cycle in a y/n odour determination measurement according to the done model check

a counter-example. In the case, the trace shows that if `btime` and `nullair` are always true, the cycle will never end. The error is quickly fixed by a toggle which prohibits two immediately following null air requests, which is indeed the way it is specified in the DIN EN 13725. The corrected specification in `YNI_01` and `YNI_02` is shown in Figure 7.1.6.

7.2 Runtime Enforcement

A runtime enforcer is an additional layer of safety for safety-critical devices. A bi-directional enforcement makes sure that pre-defined safety properties of a Cyber-Physical System (CPS) are met. Such policies can be expressed as Discrete Timed Automata (DTA). A real-life motivation for such a runtime enforcer is a pacemaker device. A pacemaker must give a pace in precise intervals if necessary but must also react to the actual physical conditions of the heart. An overview is given in Figure 7.2.1. The overall schema in Figure 7.2.1a shows that the enforcer is situated between the heart and the pacemaker controller. It receives Atrial Sense (*AS*) and Ventricular Sense (*VS*) inputs from the heart. After being checked, these inputs are forwarded

7.2. Runtime Enforcement

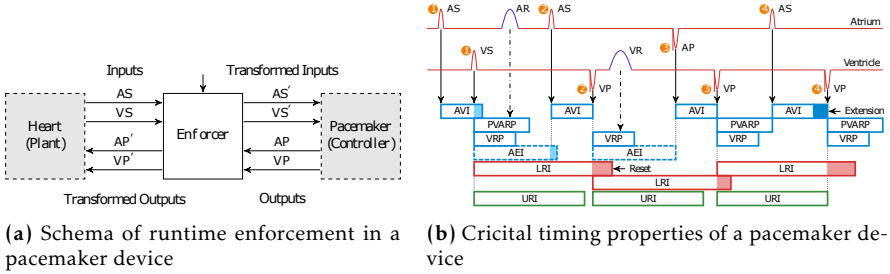


Figure 7.2.1. Runtime enforcement in a pacemaker device (from [PRS+17b])

to the pacemaker controller. The controller sends Atrial Pace (AP) and Ventricular Pace (VP) signals in the other direction. Figure 7.2.1b shows the complex timing interactions between atrium and ventricle events of the heart. These can be summarized in five safety properties:

- P1** AP and VP cannot happen simultaneously.
- P2** VS or VP must be true within AVI_{TICKS} after an atrial event AS or AP .
- P3** AS or AP must be true within AET_{TICKS} after a ventricle event VS or VP .
- P4** After a ventricle event, another ventricle event can happen only after URI_{TICKS} .
- P5** After a ventricle event, another ventricle event should happen within LRI_{TICKS} .

A safety property can be expressed as DTA. A DTA is a finite automaton extended by clocks and a set of non-accepting trap locations. Safety properties expressed as DTAs can automatically be synthesized into runtime enforcers that make sure that the non-accepting location is not reached. The complete formal definition of a runtime enforcer can be found elsewhere [PRS+17b].

Figure 7.2.2 shows the safety property $P2$ expressed as DTA in Figure 7.2.2a and in SCCharts in Figure 7.2.2b. In KIELER, a DTA in SCCharts form can be transformed into an enforcer using KiCo. The runtime enforcer generated from the DTA is depicted in Figure 7.2.2d and Figure 7.2.2c in its unoptimized and optimized versions. Basically, inputs are checked in the input region before they are redirected to the actual tick function. Afterwards,

7. Practical Applications

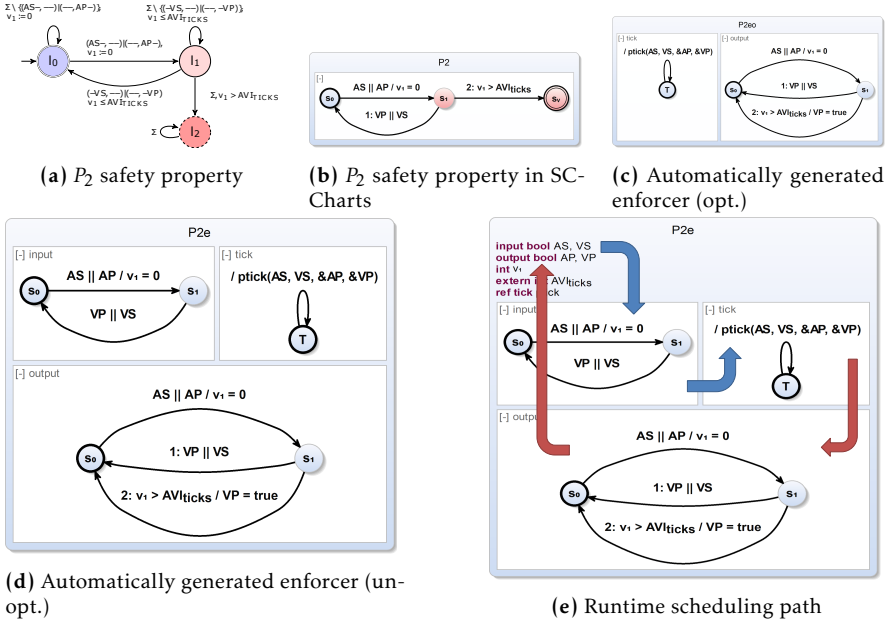


Figure 7.2.2. Example safety automaton of the P_2 property in SCCharts (from [PRS+17b])

in the output region, the outputs of the tick functions are checked. In the example, VP is set to true if the clock exceeds the $AVTICKS$ interval limit. The scheduling path is shown in Figure 7.2.2e. The region invocation in the model in the original publication [PRS+17b] in 2017 was explicitly stated to achieve a concise model. While possible to be scheduling according to the IURP, the concurrent variable accesses would have required more variables to achieve the same schedule. Today, exactly the same behaviour can be modelled dynamically using SDs, as described in Section 6.1.

7.3 Teaching

KIELER SCCharts have been thoroughly used in class to teach CPS principles. This section gives an overview over conducted student projects of different difficulty. Section 7.3.1 discusses two small-scale Lego Mindstorms projects. Afterwards, Section 7.3.2 gives an overview over a small team-scale railway project which has to be conducted over the course of a semester. Section 7.3.3 presents a challenging Quadrocopter CPS and the results of modelling with SCCharts in the context of a large Formula Student Raceyard project follow in Section 7.3.4. The tasks and results may also serve as inspiration for new or refined projects w.r.t. teaching.

7.3.1 Mindstorms

Four of the nine surveys, which have been discussed in Section 6.6, were conducted after the Real-Time and Embedded Systems class held at the Department of Computer Science at Kiel University. During the lecture several tasks were used to give a step by step introduction to SCCharts and how to program the LEGO® Mindstorms®⁵. The tasks from the last lecture in the summer term '19, which similarly appeared in the preceding iterations with only slight variations, are presented here. The goal throughout the semester is to iteratively develop more complex SCCharts models to fulfil real-time tasks with a Mindstorm robot. The last two tasks, which are described in the following, ask the students to create (1) a system which can read barcodes from a sheet of paper and (2) a pathfinder which follows a set course as fast as possible. The latter is realized as a contest between the different student teams consisting of two students.

LEGO® Mindstorms® were designed to combine creativity and problem solving into one easy-to-program unit. The core bricks allow use-case-specific configuration and design. Participants of the Embedded Real-Time lectures used the Mindstorms NXT version for all tasks. The NXT is available since 2006 and is based on an ARM processor⁶. It has three actor ports and four sensor ports as well as an USB port. Moreover, it can connect via

⁵<https://lego.com>

⁶<https://www.arm.com>

7. Practical Applications



Figure 7.3.1. An example of an assembled NXT, available at: www.lego.com

Bluetooth. The LEGO® Mindstorms® Education NXT base set includes three motors, which have rotation sensors, two touch sensors, a sound sensor, an ultrasonic sensor and a light sensor, which also includes a lamp. Three lamps from the previous Mindstorms generation and enough LEGO® bricks to design different robots as well as instructions on how a robot can be assembled are also included. An assembled NXT robot can be seen in Figure 7.3.1.

The leJOS⁷ framework is a replacement firmware for the NXT which allows to program the Mindstorm in Java. LeJOS builds upon the TinyVM⁸, a small virtual machine for Java primarily used in embedded systems. The memory footprint of the OS is only approximately 10Kb with objects only having an overhead of 4 bytes each. However, the low memory usage goal of the TinyVM results in a field limitation of 255 each, meaning that there are at the most 255 classes, with each one including up to 255 variables.

⁷<http://www.lejos.org>

⁸<http://tinyvm.sourceforge.net>

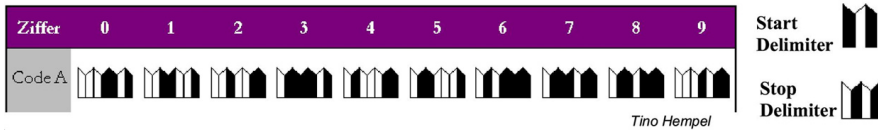


Figure 7.3.2. Each barcode begins and ends with a delimiter. The number is encoded by seven black or white bars which correspond to the European Article Number.

The maximum array length per dimension is also 255. Variables are not aligned in memory for space efficiency reasons. These limitations are great for teaching programming of real-time and embedded systems, since such systems often only have scarce resources available.

Since the KIELER compiler is able to generate Java code, SCCharts can be used to program an NXT via leJOS. However, since the netlist-based code generation approach of the KIELER SCCharts tools tends to use more variables for program guards than a manually-written Java program, the variable usage is a concern. Some of the optimizations discussed earlier made this approach for larger task viable in the first place. Also, internal complexity, as discussed in Section 4.3, was a concern. While the overall workflow of feature expansion helps to keep complex models manageable, constructs with high internal complexity, which is hidden from the modeller, may render some features practically unusable in systems with scarce resources (cf. Section 4.3). However, as the small tasks did not require complex features, they were not used widely.

Barcode Reader

The task is to develop a controller which enables a Mindstorm to travel over one or several barcodes and to read the corresponding numbers according to the encoding described in Figure 7.3.2. It is deemed the most difficult of that class by the students as it combines real-time CPS aspects while pushing the variable limit of the TinyVM. This task also provides a simulation visualization within KIELER via an SVG as seen in Figure 7.3.3. The barcodes consist of 5mm wide bars and include four numbers. The first three correspond to the encoded number, the last is a checksum which ensures

7. Practical Applications



Figure 7.3.3. Barcode reader simulation visualization

correctness of the previous digits. A barcode with digits $d_1 - d_4$ is valid if $d_4 = 9 - ((d_1 + d_2 + d_3) \% 10)$.

Depending on the modelling approach, it might not be possible to develop a state machine which determines the number because of the variable limitation, as explained previously. A delayed state for each bar combination usually results in too many variables. The variable usage can be optimized by using an array or a bit vector instead of a state machine. Alternatively, not all bars of a digit have to be checked since their encoding is redundant. An exemplary controller, developed in the winter term 15/16, which does not check all bars of a digit, can be seen in Figure 7.3.4. The `drawLight` region draws to the display of the Mindstorm, the `lightSensor` region reads the light value and categorizes it in light or dark, and the `readNumber` region evaluates the light values. This controller has more functions than the task requires, since it also rotates according to the read number on the barcode and continues to search for a new barcode in that direction. Models developed in the summer term 19 tend to be bigger, since the code generation was optimized with respect to used variables.

Student models often include a setup phase to calibrate the light sensor. The start delimiter is often used for to measuring the length of each line. 5mm rotations can also be achieved with the rotary sensor of the motors, but this solution is not robust. Inaccuracies in the brick assembly and deviations in the drive paths make this solution prone to errors.

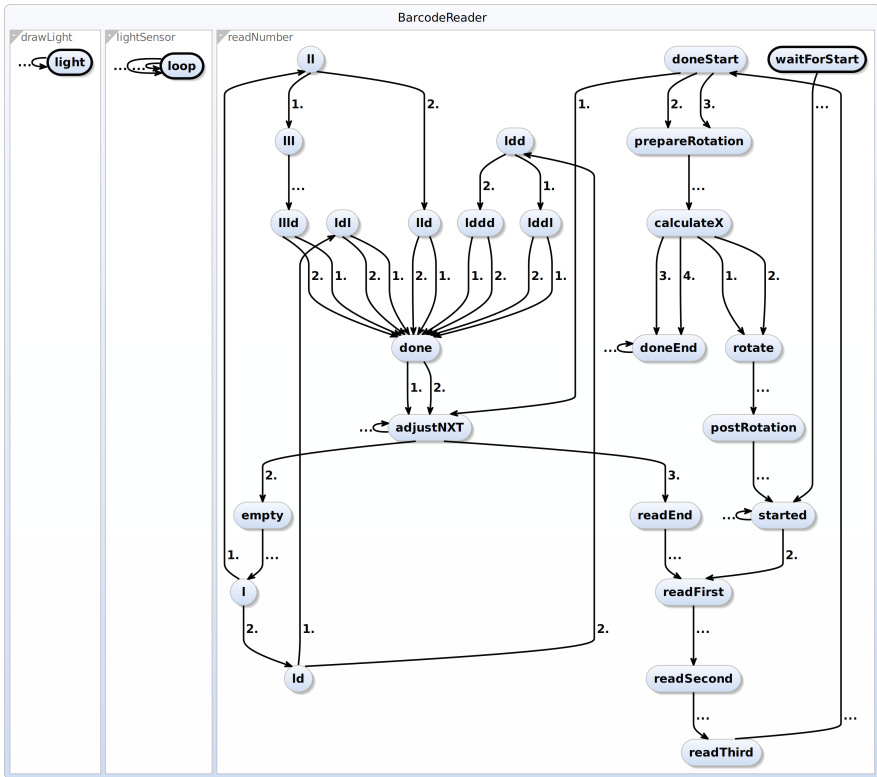


Figure 7.3.4. An example for a barcode reader model (from [SMS+19a])

7.3.1.1 Pathfinder

The pathfinder task is usually the final task of the class. The robots must follow a path on a given map. The participants present their solutions during a contest. The winner of the contest is the team which developed the fastest robot which can finish the course without error. The path has a thickness of approximately 1.5cm from start to finish and the mat has the dimensions of approximately 1.4m width and 1.8m height. As before, a corresponding visualization is provided within KIELER, which can be seen in Figure 7.3.5. It

7. Practical Applications

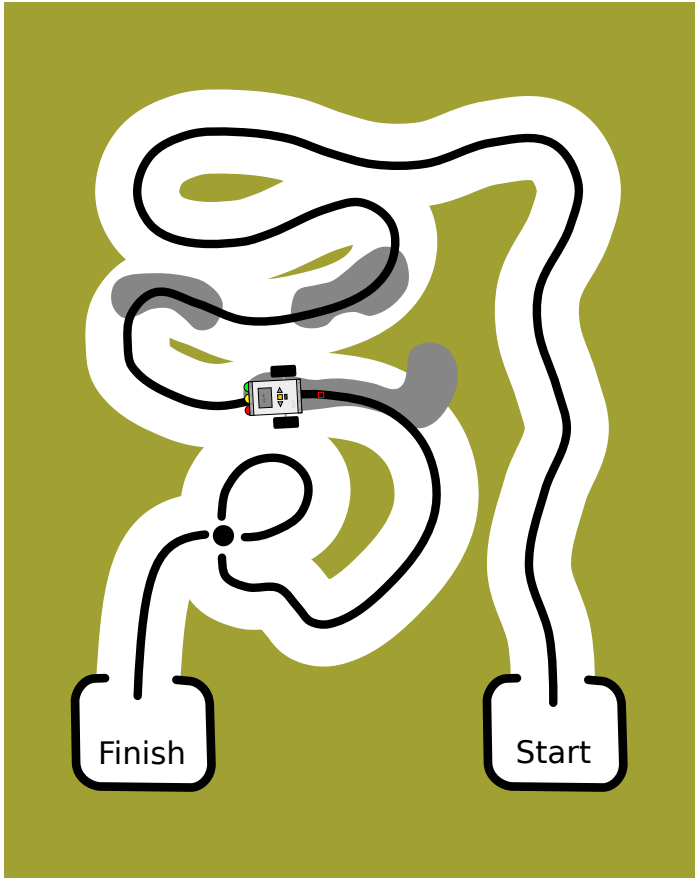


Figure 7.3.5. The pathfinder mat as used in the simulation visualization.

can be used to simulate the controller. The simulation also teaches students that simulations are useful to spot coarse inaccuracies, although it might not resemble the real world exactly and the system has to be tested on the physical mat. The mat has the following additional obstacles and challenges, which might also not be present in the simulation:

- ▷ The grey areas might confuse the light sensor and the robot might go off track. This is not simulated in the simulation. The light value is very precise while simulating and the light on the physical mat might be noisy. However, more realistic noisy environments can be modelled, e. g. by using Probabilistic Priority Transitions (PrPTs), which were introduced in Section D.2.
- ▷ There is white text printed on the path after the grey areas. This makes it more difficult to recognize whether the robot is actually on the line or not.
- ▷ The crossing has white areas on entry and exit. The loop's crossing must be passed straight. This can result in skipping or being trapped in the loop if the wrong black line is recognized.
- ▷ The robot must be able to turn fast enough to successfully pass the narrow curves.

There are several different approaches to solve this task. Most groups developed a controller with a state for driving left and one for driving right. Additional states for finding the line again after losing it and calibrating the light sensor are also present in most models. These solutions usually have between two and eight states. The most advanced controllers model a PID controller and drive on the edge of the line. Groups which model their pathfinder via a PID controller have often one of the fastest NXTs. Calibration of the light sensor or a good threshold value for the brightness are essential to many groups and need to be fine tuned. It is also not permitted to hard-code the path into the code. The robot should work just as fine as before if the path is travelled from Finish to Start (and it is also not permitted to hard-code both directions).

7.3.2 Railway System

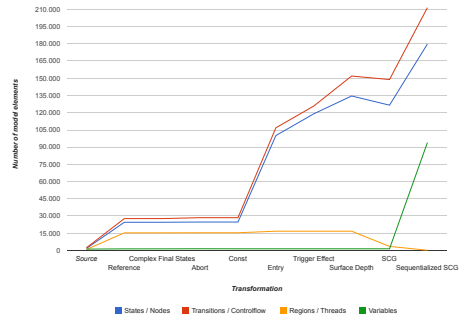
Two of the nine surveys discussed in Section 6.6 were conducted after railway projects. The model railway demonstrator⁹ shown in Figure 7.3.6a is a practical lab at Kiel University since 1995. Since 2006 the lab is managed by the Real-Time and Embedded Systems Group. SCCharts was first used in 2014

⁹<http://www.informatik.uni-kiel.de/~railway>

7. Practical Applications



(a) Railway installation



(b) Number of model elements during expansion

Figure 7.3.6. The first SCCharts railway project

in this context. The results are documented in a technical report [SMS+15] but also summarized as first case-study in Section 6.6. The track layout of the demonstrator was inspired by a mountain pass in Canada, the *Kicking Horse Pass*¹⁰. Since its initial version in 1995, the model railway installation has the following characteristics: Scale H0, approximately 130 meter of tracks, 48 blocks of track segments, 28 switch points, 56 signal lights, and 80 reed contacts. The current version is the fifth iteration of the railway demonstrator, now using COTS components, such as Arduinos and Raspberry Pis, to control the peripherals. The railway hardware is connected to the Arduino nodes, which are linked via USB to a small number of Raspberry Pis. These are connected via Ethernet to a central controller, which runs a C program. The controller interprets the reed contact information to infer the train positions. It controls the voltage, i. e. speed, of the track segments, the switch points and signal lights.

In various practical labs the task was given to model a controller in various synchronous languages, such as SCADE, Esterel, Ptolemy and SCCharts and to generate code for the railway installation. This usually requires a coordinated team effort which takes place over the course of a semester. In the summer term 2014 the group supervised a railway project in order to

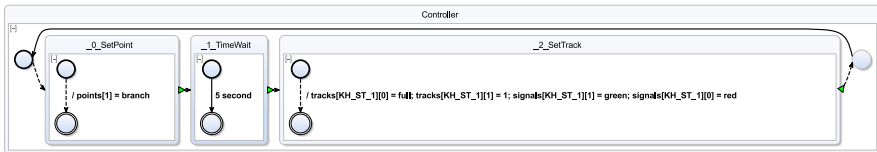
¹⁰http://en.wikipedia.org/wiki/Kicking_Horse_Pass

```

1 Start:
2   Set point 1 to branch.
3   Wait for 5 seconds.
4   Set track KH_ST_1 to full.
5 Loop.

```

(a) RailSL code for a loop



(b) Generated SCChart for the RailSL loop

Figure 7.3.7. SCChart generated from a looped RailSL block (from [Eum17]).

evaluate SCCharts as a language and the KIELER SCCharts tooling for the first time. During the railway project seven participants worked approximately six months with SCCharts building a controller which runs up to eleven trains concurrently with dynamic schedules. The controller fully expands to 135,000 states, 152,000 transitions, and 17,000 concurrent regions after eliminating all reference states. 1,628 states together with 2,219 transition in 183 concurrent regions were modelled manually. The eleven train model railway controller had a final size of roughly 400,000 lines of C code. Detailed information about the project and its results can be discovered separately in the technical report [SMS+15].

Figure 7.3.6b shows the number of model elements for the SCCharts model railway controller example at every intermediate stage of the compile chain. It shows how much complexity of the model is hidden by using Extended SCCharts features for modelling the complex behaviour of this controller. Furthermore, the students also struggled with teething troubles of our early prototype tool chain. The insights gained during such a large project helped immensely in improving the quality and stability of the compiler, which eventually resulted in KiCo.

Eumann used the interactive model-based approach to create a simple railway DSL, named RailSL, for pupils to demonstrate modelling concepts

7. Practical Applications

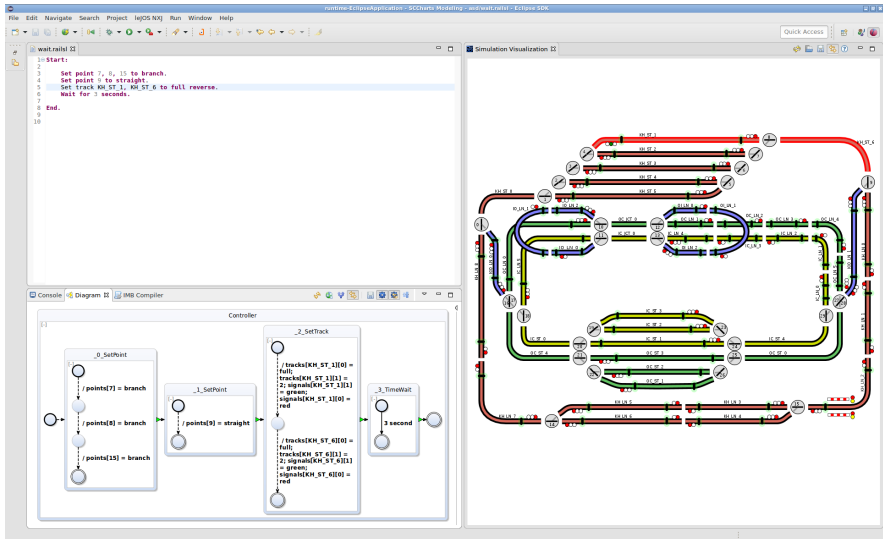


Figure 7.3.8. RailSL IDE with reasonable modeller feedback (from [Eum17]).

and tools [Eum17]. The minimalistic DSL is internally translated into an SCChart, which drives the railway installation. Figure 7.3.7 shows (a) a simple loop in RailSL and (b) the corresponding generated SCChart.

Following iMURD, the DSL can be accompanied with reasonable views to guide the modeller. The RailSL IDE, depicted in Figure 7.3.8, shows a schematic of the railway track layout next to the RailSL editor. Referenced tracks and switch points are highlighted instantaneously to provide meaningful feedback. The generated model can be displayed below the editor, e. g. for debugging purposes.

7.3.3 Quadrocopter

The group hosted a quadrocopter project in 2015. While only allowing for a smaller team size and generally requiring a smaller model as flight controller, the project of 2015 was as challenging as the railway projects because of the physical environment. Instead of a stationary installation, the quadrocopter

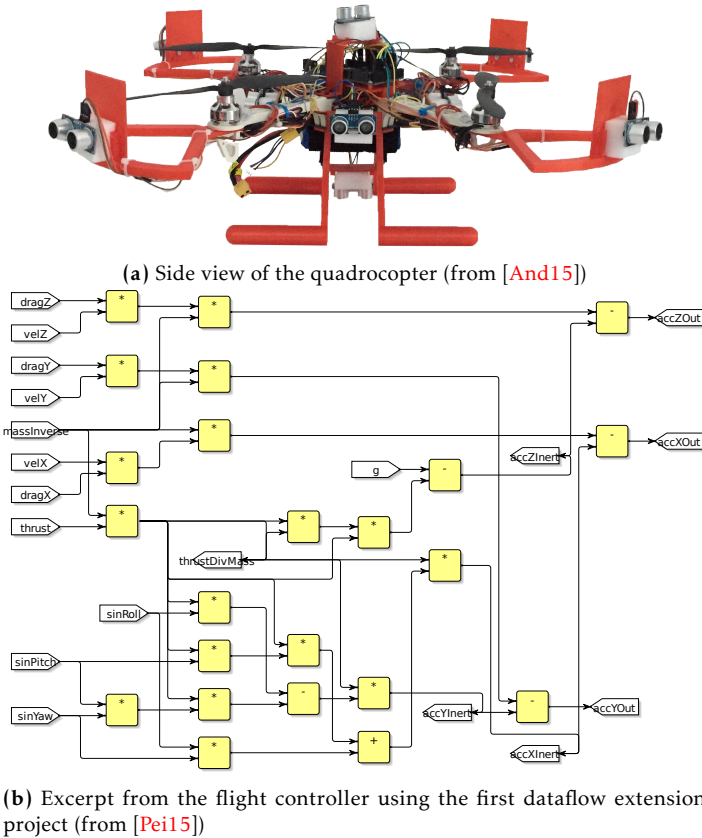


Figure 7.3.9. Quadcopter student project 2015

is a moving entity. It was equipped with ten ultra-sonic sensors in order to avoid collisions. However, testing the system and providing measures for incorrect behaviour that would protect the actual hardware, such as the rotor blades, was quite challenging. The whole quadcopter, which is shown in Figure 7.3.9a, was 3D-printed with exception of the core frame. An Arduino was used to host the flight controller model. The project was

7. Practical Applications

also the first practical application for the SCCharts dataflow extension (cf. Section 6.2), which is a natural fit to design an PID controller, which is commonly used in quadcopter flight systems. Figure 7.3.9b shows the model for the acceleration calculation in the early versions of the dataflow extension synthesis.

7.3.4 Raceyard

SCCharts have been successfully used to model a controller for the Kieler Formula Student Team Raceyard [SSS+19]. The Formula Student is an international design competition for students, where every year the goal for each team is to design, construct and test a race car. These race cars are then compared and judged during official events. Since 2005 Team Raceyard takes part in the Formula Student as the official team of the University of Applied Sciences Kiel.

For the task of designing and testing the control systems of the Electric Control Unit (ECU), Raceyard uses the MathWorks' modelling software Simulink¹¹ (also see Section B.2 on page 360 ff.). Of special importance for Raceyard are the integrated blocks for PID controllers, scopes and displays to view all signals within the system during and after run-time. Verifying as well as fine-tuning the ECU is made via the simulation software IPG Carmaker¹². IPG Carmaker provides a virtual 3D environment and simulated drivers of varying abilities so that the effects of the ECU can intuitively be seen on the 3D car model. Due to integration of IPG Carmaker into Simulink, values from the simulation, such as the velocity of the car or the forces acting upon it, can be measured and saved using the scope- and display-blocks in Simulink. Following the test-phase, the Simulink model is then converted into C code and put on an STM32F40 micro controller inside the car.

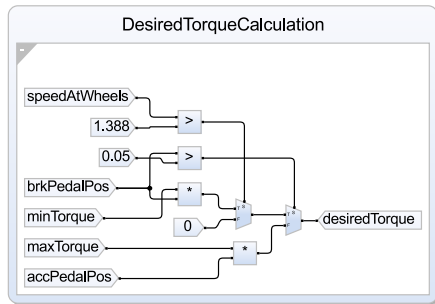
Work done by Santarossa et al. [SSS+19] showed how a functionally equivalent system can be designed via KIELER SCCharts. A complete controller was modelled in KIELER and validated to behave the same as the original controller both in Simulink directly as well as in the 3D simulation environment IPG Carmaker, as can be seen in Figure 7.3.10a. Tests on the

¹¹<https://mathworks.com/products/simulink.html>

¹²<https://ipg-automotive.com>



(a) One of the validation tracks made in IPG Carmaker



(b) SCCharts dataflow model of the desired torque calculation

Figure 7.3.10. Using SCCharts Models in Simulink to Model an Electronic Control Unit [SSS+19]

performance of both controllers show that while a slowdown can be observed when comparing the generated C Code, simulation time in IPG Carmaker only increases by a negligible factor. The newly developed dataflow extension (cf. Section 6.2 on page 247) was also tested in this project, shown in Figure 7.3.10b, as dataflow modelling becomes the natural choice when designing computation-heavy controllers, such as PIDs.

Such large-scale projects usually require highly dedicated participants, *Remark* who are willing to put above-average effort over several months into the project. While sometimes seen as a source of delay w.r.t. a fast graduation, they are in fact invaluable practical experiences. They should be promoted and rewarded more often—also by related teaching facilities. It may be noted here as further incentive that Mr. Santarossa, who worked for over a year in the Formula Student Team, graduated with honours and currently pursues a PhD.

Conclusion

Never delay kissing a pretty girl or opening a bottle of whiskey.
— Ernest Hemingway

The last chapter of this thesis summarizes the results towards interactive model-based compilation implemented as the KIELER Compiler (KiCo) in KIELER, which comprises the reference compiler for SCCharts, in Section 8.1. Ongoing and future works are discussed in Section 8.2.

8.1 Summary

Chapter 3 presented the methodical interactive model-based compilation approach. Comparing the structure of the proposed approach with the definition from Lattner [Lat06] (cf. Section 2.3.2 on page 30) shows that interactive model-based compilation is indeed a concept for a compiler infrastructure. Processors are modular and reusable components for building compilers. They can be shared across different compilers and are usually small self-contained units, which reduces development time and expenses. Besides the processors that already exist within the KIELER project, new ones can be created easily and embedded within compilation systems interactively, since these systems themselves are dynamic models.

Contrary to LLVM, the approach sacrifices some of the cross-compiler modularity to stay in familiar domains for as long as possible to provide the modeller with reasonable views. Nonetheless, different compilers can profit from many common processors, as the KiCo universe on Section 4.2

8. Conclusion

shows and the different SCCharts compilations in Chapter 5 demonstrate. Chapter 6 showed that language additions and variations can be prototyped easily. Besides implementing model transformations, the concept can be used to perform various tasks, such as quality measurements, simulation and code deployment, as has been demonstrated in Section 4.3 and Section 5.5.

In the initial contribution towards statecharts [Har87], Harel wrote *“The future lies in visual languages and methodologies that, with appropriate structuring elements, can exploit all the obvious advantages of graphical man-machine interaction. Validating these theses is something a scientific paper cannot really achieve, but an effort has been made here to convince the reader that they are worthy of serious consideration.”* This thesis explained ways of how domain experts and tool developers can profit from the interactive model-based compilation approach. Section 4.1 gave an overview of modelling views used within the KIELER project and the general idea behind interactive model-understanding-refinement-documentation feedback loop (iMURD). However, an unlimited number of other views, tailored to the specific needs of their users, are imaginable. Efforts have been made throughout Chapters 3–7 to show the documentation capabilities of interactive intermediate results.

KiCo was used to implement the reference compiler for the synchronous language SCCharts. Chapter 5 implements three different compilation approaches and shows the advantages of the model-based compilation approach, which provides the modeller with meaningful feedback and allows for rapid prototyping. Similarly, any other language can be implemented. Particularly VDSLs may profit from such a meta-model engine, which apart from providing a fully modular compiler also includes means for synthesizing transient views automatically. New DSLs can build upon already included semantics and technology for simulation and benchmarking. Extensions, language variations and the transfer of established concepts to other languages, such as Esterel, have been shown in Chapter 6. Finally, Chapter 7 discussed successful industrial, teaching and academic projects. At least within the KIELER project, the team grew to appreciate the simple and efficient way to prototype new processors and assemble new systems, or, to put into the somewhat blunt words of a former student assistant: *“Es ist so einfach, geilen Sch... einzubauen.”*¹(It’s so simple to implement hot sh**.)

I sincerely hope that parts of this methodology will help you, too.

— Steven Smyth

8.2 Future Work

As future work, the KIELER team wishes to evaluate further by which means the tooling can be improved to help the modeller and whole developer teams even more. New user front-end technologies, such as combined web/desktop application possible via frameworks, e. g. `electron`², are currently under investigation. A first front-end based on the Theia framework has already been developed [Dom18; Ren18]. It supports model-based compilation via `KiCo` using the Language Server Protocol (LSP) and can be used as web and desktop application. Especially debugging model-based programs is a greater concern, as the survey evaluation has shown. The prototype is depicted in Figure 8.2.1a showing a running KIELER in a standard browser. To the right, `Pyro` shows a similar approach for `CINCO`, cf. Section 2.3.3.

Further potential future work towards meta-tool engines, runtime debugging, object-oriented statecharts and sequentially constructive dataflow languages is discussed in the following sections.

8.2.1 Enabling Domain-Specific Groups

Meta tool generators, such as `CINCO`, and *meta tool engines*, such as `KiCo`, make it increasingly easy to generate highly specialized tools surrounding specific DSLs. These tools assist domain experts in actually solving their domain problems without the need to involve many general purpose programming experts. Arguably, by employing these tools a team of domain experts can concentrate on the issues at hand and solve them faster.

However, both approaches contain a semantic gap between a new DSL and the automatically generated product, which is depicted in Figure 8.2.2. Presently, this gap needs to be resolved by programming experts. In the

¹Andreas Stange, Aug 2017

²<https://electronjs.org>

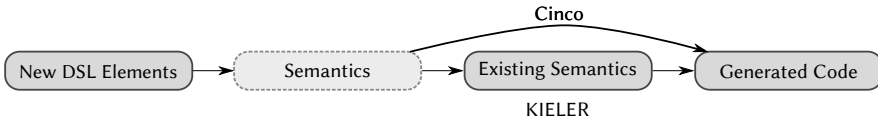


Figure 8.2.2. The semantic gap

CINCO approach, the semantics of the elements of the new DSL have to be programmed directly. In KIELER, a second method is to transform new semantics to already existing semantics and use the expression language of the engine. Therefore, the steps towards the final generated code might be smaller, but the programming experts also have to have knowledge about the specific, e. g. SCCharts, semantics. Contrary, in CINCO, the developers can directly use general purpose languages (which is also possible in KiCo), but the steps might be bigger. It is reasonable to introduce a third *tooling expert* group, which is located in-between domain and programming experts and which must be enabled to fulfil this task reasonably. The open question is how.

Graph rewriting systems are one possibility to close the semantic gap. They can describe the semantics of new DSL elements on a higher abstraction level and re-use semantics already present. Especially if a graphical notation exists, as it is the case with SCCharts, new DSL elements can be transformed to already existing graph constructs, such as the SCCharts Kernel Pattern (SKP). Furthermore, as SCCharts uses one model to synthesize graphical and textual views on programs, the principle can also be extended to text rewriting by applying graph rewriting to the semantic models. It may also be beneficial to add a high abstraction graph rewriting system to KIELER.

However, even within the high-level compilation of SCCharts, it is not trivial to give a high-level description for single more complex transformation steps. Currently, SCCharts transformations are written in Xtend, which is a modern and arguably simpler Java variant w.r.t. model manipulation. It is fair to say that many of the common programming practices and structuring mechanics are used and that they may not be substituted easily. Therefore, an even simpler abstraction, such as the transient models in Section 7.1.1, might be promising or necessary. This would, however,

8. Conclusion

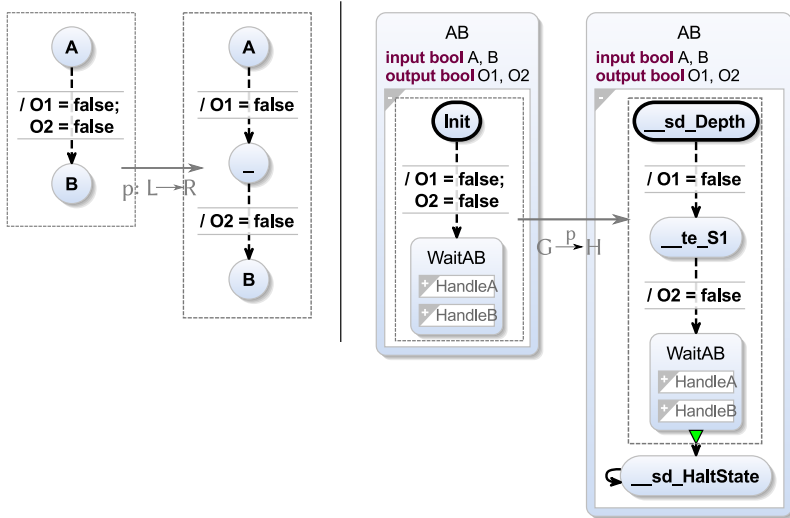


Figure 8.2.3. SCCharts graph rewriting example

restrict the expressiveness of the newly generated DSLs. Finding an easy to use way to describe the semantics for new automatically generated DSLs to close this semantic gap will be the challenge for any upcoming meta tool generator or meta tool engine.

8.2.2 Runtime Debugging of Statecharts

As shown in Section 6.6, debugging is a weak spot of SCCharts and perhaps even Statechart dialects in general. While simulation works well in a closed environment and can be enriched by standard debugging techniques, such as breakpoints, on modelling level [Gri16], debugging interaction with an external environment often becomes difficult. Eumann explores possibilities to debug statecharts directly in their live environment [Eum20]. This has been motivated by an industrial user from the railway domain, who uses SCCharts together with Java code.

A model-based debugger runs on host language level and uses knowledge

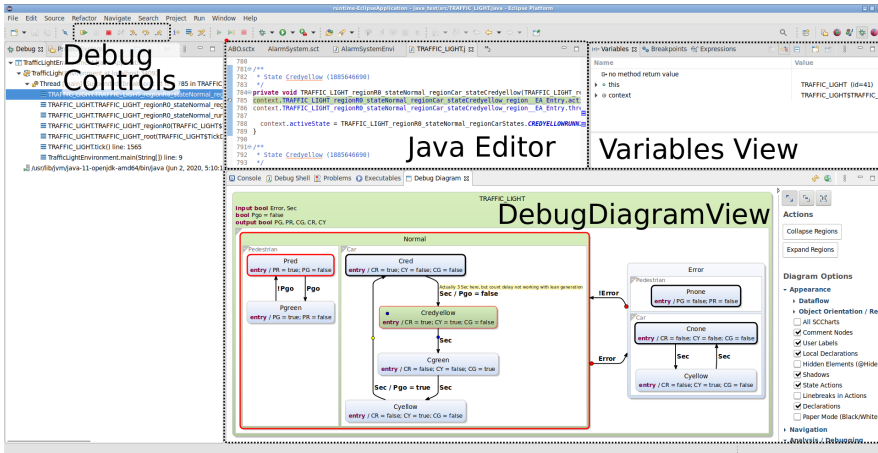


Figure 8.2.4. KIELER user interface with components relevant to debugging highlighted (from [Eum20])

of the code generation process. With this information, it can visualize the generated code's memory state on the model level, making it easier for the user to grasp the current state of the model. The debugger allows for the placement of host-language breakpoints through the source model. The execution is interrupted in the right place without knowledge about the generated code. The concept fully integrates into KiCo and uses its tracing capabilities to annotate the generated code with information about its source model elements. Figure 8.2.4 shows an active debug session. Although the generated code is debugged, the information about the source model can be extracted during debug-time and visualized. Concepts like this are important to increase the general reach of modelling pragmatics.

8.2.3 Object-orientated SCCharts

Object orientation is a powerful and widely used paradigm for abstraction and structuring in programming. However, in synchronous languages, originally developed to design embedded reactive systems, there are only

8. Conclusion

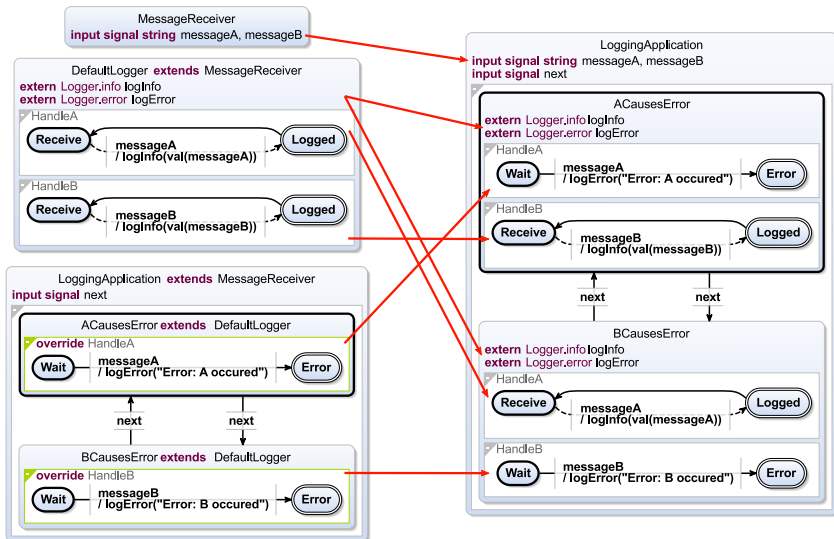


Figure 8.2.5. Example for usage of inheritance in SCCharts (left) and the result after inheritance is statically expanded by the compiler (right). Red arrows indicate where the parts of the model are expanded into (from [SSM19]).

few object-oriented influences. Schulz-Rosengarten investigates how the object-oriented paradigm can help to improve the pragmatic modelling methodology in combination with statecharts. His work includes direct modelling aspects, such as adding inheritance to synchronous visual languages, such as SCCharts, but also comprises MoC aspects by retaining determinism when using object-oriented classes for encapsulating external, i. e. host language, objects. One way to ensure the latter is by using SDs presented in Section 6.1.

Figure 8.2.5 illustrates on the left-hand side how inheritance can be used in SCCharts. In the underlying scenario, incoming messages, here `messageA` and `messageB`, must be processed differently depending on the state of the application. By default, a receive message must be logged. This common behaviour is modelled in the `DefaultLogger`, which has separate regions for each message. In the actual application represented by `LoggingApplication`,

the behaviour differs from the default logging behaviour depending on the state. Inheritance is considered an extended feature in SCCharts and removed by an M2MT. It fits into the interactive model-based compilation methodology and extends the reference SCCharts expansion, introduced in Section D.1. Figure 8.2.5 presents on the right-hand side the result of this transformation.

Figure 8.2.6 shows a host language object encapsulated in an SCCharts class. While the concrete external behaviour is hidden from the modeller and implemented in the host language, SDs can still be used to establish scheduling rules. Here, the SD CounterSD is used to define a scheduling order between increment, decrement and getValue. The schedule uses two indices, which are both commuting. Index 0 is assigned to increment and decrement and index 1 to getValue. Therefore, increment and decrement are scheduled before getValue.

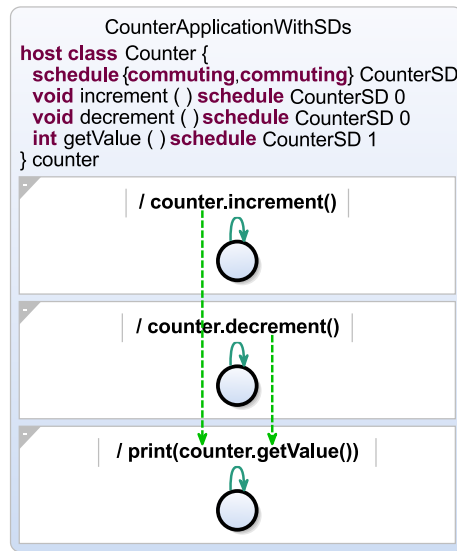


Figure 8.2.6. Deterministic usage of host language object Counter using an SD ([SSM19])

8.2.4 Sequentially Constructive Dataflow

Grimm investigates possibilities towards modelling pragmatics w.r.t. sequentially constructive dataflow languages. Therefore, a systematic approach for automatically creating visual diagrams, akin to a SCADe model, from a Lustre code is developed. This not only saves tedious manual drawing effort but also allows the creation of different views for the same program. Grimm builds upon the SCCharts dataflow extension presented in this thesis and enriches it with Lustre semantics, which permits a translation from Lustre to graphical SCCharts. Therefore, the established SCCharts simulation and

8. Conclusion

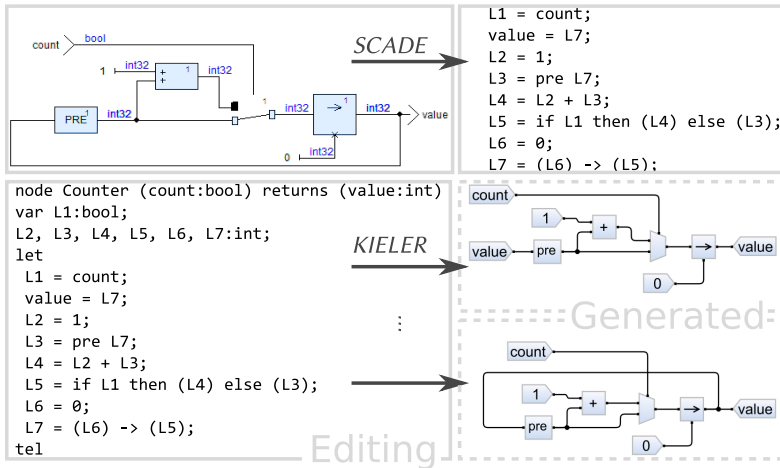


Figure 8.2.7. Traditional modelling flow (top) vs. modelling pragmatics (bottom) (from [GSS+20])

code synthesis powered by KiCo can be used as an alternative to existing Lustre compilation techniques. The SCM_oC underlying SCCharts can be used to conservatively extend Lustre, thereby providing a deterministic semantics to Lustre programs that would be rejected under its original semantics. Figure 8.2.7 illustrates the classic approach, which uses the graphical syntax as primary driver, compared to the pragmatic modelling approach, which synthesizes views with different strategies.

Acknowledgements

*Along every step of our journey through life, our mind is being programmed.
If we are not programming it ourselves, someone else is doing it to us.
— Joseph Rain – The Unfinished Book About Who We Are*

I would like to express my appreciation to the following persons who have accompanied me on my academic journey with respect to my thesis.

Special thanks go to my *Doktorvater Prof. Dr. Reinhard von Hanxleden* to whom I convey my sincere thanks for not only making it possible to write my thesis at the Real-Time and Embedded Systems Group but also for the trust he has placed in me. The previous five years of research and teaching were a brilliant and invaluable experience to me. I have learned so much and met so many amazing people during this time. Thank you very much.

Special thanks go to *Prof. Dr. Bernhard Steffen*. Thank you for encouraging us to participate in your international symposiums and for your invitation to the modelling colloquium in Dortmund. These research exchanges were always fruitful and helped to shape our vision. I sincerely hope that these gatherings will continue in the future.

Thanks go to *Prof. Dr. Partha Roop* for the invitation to the pacemaker device study and for using KIELER to teach CPS foundations. It was a great pleasure to be part of such a dedicated team. I hope we will meet again at one of the annual synchronous workshop meetings.

Thanks go to *Prof. Dr. Michael Mendler* for all the advice given during our mutual paper sessions. Meeting you at conferences and here in Kiel was always a pleasure for me.

8. Conclusion

I thank *Prof. Dr. Sergei Knizhin* for his professionalism and hospitality during our academic exchanges. I hope that they will continue after the current pandemic.

I thank my former student, colleague and successor as the team leader of the KIELER semantics team, *Alexander Schulz-Rosengarten* for countless hours of consideration, for proof reading my thesis and for keeping my workload well-balanced, especially in the final weeks of writing this thesis. Your professionalism is invaluable for the KIELER project. I wish you all the best for your research and thesis.

I thank my colleague, *Lena Grimm*, for the mutual hours of research with respect to sequential constructiveness in dataflow languages and for her personal support.

I thank *Sören Domrös* for his help during the steam-boiler experiments.

I thank *Maria Kosche*, *Gianna Persichini*, *Kristina Rolsing* and *Felix Niefind* for their personal support during the last year of writing this thesis.

Last but not least, I thank my family, my brother *Robert S. Smyth jun.* and my parents, *Robert S. Smyth sen.* and *Waltraud F. M. Smyth*, who made my studies, and therefore this thesis, possible in the first place.

Publications

This section presents my publications related to this thesis. While this thesis wraps up the research towards interactive model-based compilation in an own contribution, selected parts of the following publications reappear throughout this thesis to form a coherent story.

Section A.1 lists the major peer-reviewed publications, which are also cited in the presentation of the contributions in Section 1.1. Section A.2 shows published works or workshop presentations which were not peer-reviewed but also contribute to my thesis. Section A.3 lists other published works, which are related to my thesis topic, but which I do not consider a major contribution. Theses which I supervised follow in Section A.4.

A.1 Major Publications

I have been the leading or supervising author of the publications discussed in this section. They are peer-reviewed and published in the proceedings of the corresponding conference or the specified journal. The entries are ordered according to the list of contributions in Section 1.1.

[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. LNCS. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260

“Towards Interactive Compilation Models” lies the foundation for the compilation systems concepts I proposed in Chapter 3.

A. Publications

- [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

This paper contributes towards guidance in model-based compilations. The content, which resulted in the iMURD loop I presented, is summarized in Chapter 4.

- [SLH16] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. “Model extraction for legacy C programs with SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA ’16), Doctoral Symposium*. Vol. 74. Electronic Communications of the EASST. With accompanying poster. Corfu, Greece, Oct. 2016. doi: 10.14279/tuj.eceasst.74.1044

It was shown that model-based compilation tools can be used to extract, display and re-compile legacy C programs. The approach is sketched out in Section 6.5. The work was preceded by Mr. Lenga’s bachelor’s thesis, which I supervised.

- [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

This paper explores human-readable code generation for statecharts, which is particularly interesting for safety-critical applications where the final code has to be verified manually. This resulted in the state-based compilation approaches, which I proposed in Section 5.4.

- [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

This paper explains SDs, as proposed in this thesis. SDs are a first class citizen language construct, which enables a modeller to influence the scheduling of the underlying MoC. Section 6.1 explains the concept.

- [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

As discussed in Section 2.1, SCCharts is a conservative extension to SyncCharts, which can be seen as graphical representation of Esterel. Therefore, it follows that sequential constructiveness can be applied to other synchronous languages. SCEst adds the sequential constructive paradigm to Esterel. The details are explained in Section 6.4. The work was preceded by Mr. Rathlev’s master’s thesis, which I supervised.

- [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 (TECS)—Special Issue on MEMOCODE 2015* 17.2 (Dec. 2017), 33:1–33:26. ISSN: 1539-9087

The long version of the SCEst conference paper [RSM+15] gives more details about the sequential constructive transformations. As mentioned, details can be found in Section 6.4.

A.2 Minor Publications

The following contributions are not peer-reviewed or I have not been the leading author. However, all of them, with the exception of the technical reports, were presented before and discussed afterwards with fellow experts. The gained insights influenced the results presented in this thesis. The entries are ordered according to the list of contributions in Section 1.1.

A. Publications

- [SSH18d] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Watch your compiler work — Compiler models and environments*. Technical Report 1806. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018

The technical report gives further information on interactive compilation models. In particular, the report illustrates in two case studies how compilation models can be used to create optimizations and how compilations can be altered to accept a broader class of models. Chapter 3 discusses some of the results.

- [SMH15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “A data-flow approach for compiling the sequentially constructive language (SCL)”. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörtshach, Austria, Oct. 2015

This paper describes the mechanics of the netlist-based compilation from a technical point of view. Several steps have been refined since 2015. Section 5.2 discusses the complete compilation approach.

- [RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. Vol. 9953. LNCS. Corfu, Greece, Oct. 2016, pp. 150–170. doi: 10.1007/978-3-662-45234-9

The KIELER SCCharts editor supports different code generation approach by default. One approach is the netlist-based compilation which can be used to generate code for software and hardware. Besides giving an early overview of the general code generation tool chain, which is discussed in Chapter 3, this work also explains the workflow and results when generating circuits directly, which is sketched out in Section 5.2.9.

- [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

This paper investigates pragmatic modelling techniques for Lustre. The comparison to the KIELER dataflow extension is explained in Section 6.2.

[SSH18b] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Practical causality handling for synchronous languages*. Technical Report 1808. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2018

The technical report gives further information on SDs, which are presented in Section 6.1.

[MS15] Christian Motika and Steven Smyth. *Updates on SCCharts*. Presentation at the 22th International Open Workshop on Synchronous Programming (SYNCHRON '15), Kiel, Germany. Dec. 2015

The presentation about SCCharts at the Synchron workshop gave an overview over the progress of the SCCharts development. Besides general improvement of the SCCharts tools, which were influence by the railway project [SMS+15], I presented the dataflow extension of SCCharts for the first time publicly. Section 6.2 explains the hybrid concept.

[SSS+19] Monty Santarossa, Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Using SCCharts models in Simulink to model an electric control unit*. Technical Report 1903. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2019

This technical report presents the results of SCCharts in the context of the Student Formula Raceyard project. An overview is given in Section 7.3.4.

[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

A. Publications

2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015

The technical report 1510 presents the results of the railway student project of 2014/15. The railway system and the associated projects are described in Section 7.3.2. The practical nature of the project, where students were tasked to create a railway controller which manages up to eleven trains simultaneously, helped to stabilize the KIELER SC-Charts tools and influenced the development. Particularly, referenced SCCharts, see Section D.1, did profit from the project. The report also gave the first results on our SCCharts language evaluation, which are discussed in more detail in Section 6.6.

[MSS+16] Christian Motika, Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *KIELER SCCharts tutorial*. Interactive Tutorial performed at the 23th International Open Workshop on Synchronous Programming (SYNCHRON '16), Bamberg, Germany. Dec. 2016

The SCCharts tutorial was first conducted in 2016 at the Synchron workshop in Bamberg. The tutorial included introductions of the SC-Charts languages as well as the KIELER SCCharts tooling. Subsequently, the professional participants were asked to solve real-time tasks with Mindstorms robots and to eventually fill out a survey, which results I present in Section 6.6.

[SDH19] Steven Smyth, Sören Domrös, and Reinhard von Hanxleden. *A case-study on manual verification of state-based source code generated by KIELER SCCharts*. Technical Report 1905. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019

The report presents the results of the second case-study towards the state-based compilation approach. The approach is explained in Section 5.4.

[SMS+19b] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxle-

A.3. Other Publications

den. *SCCharts: the mindstorms report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019

This report describes and evaluates the SCCharts questionnaires given out during a five year span of SCCharts tools development. Section 6.6 presents the results.

[PRS+17b] 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

Runtime enforcement observes a synchronous program and bidirectionally corrects inputs/outputs if they leave limits specified by safety automata. Synchronous languages, such as SCCharts, can be used to describe such automata. The automata can be transformed into an enforcer program which behaves as specified. I contribute towards the practical generation of these enforcers according to safety automata specifications. Runtime enforcement is sketched out in Section 7.2.

[PRS+17a] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. “Runtime enforcement of reactive systems using synchronous enforcers”. In: *Proc. International SPIN Symposium on Model Checking of Software (SPIN ’17)*. Santa Barbara, CA, USA, 713–14 2017

This publication also describes runtime enforcement extending on the fundamentals described before. The SCCharts part, as discussed in Section 7.2, is similar to the works published earlier.

A.3 Other Publications

This section lists other published work, which is related to my thesis topic, but which I do not consider a major or minor contribution. However, as parts of these publications influenced the subsequent work, they are listed for the

A. Publications

sake of completeness. They are ordered according to their topics w.r.t. the chapters of this thesis.

- [MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. “Compiling SCCharts—A case-study on interactive model-based compilation”. In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 461–480. doi: 10.1007/978-3-662-45234-9

This work gives details about the spiritual predecessor of the KIELER Compiler described in Part I. I contributed to an overview over the first two low-level compilation approaches of SCCharts, which are discussed in Chapter 5.

- [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. LNCS 11200. Springer International Publishing, 2018, pp. 504–523

Despite the fact that languages tend to be either control-flow or dataflow oriented, both representations can help to get an overview over a program and to identify potential causality problems. Different views, including a new refinement of the work presented in “Augmenting state models with data flow”, is discussed in Section 4.1.

- [SSM+19] Steven Smyth, Alexander Schulz-Rosengarten, Christian Motika, and Reinhard von Hanxleden. “The KIELER SCCharts Editor—A modular open-source modeling suite with automatic diagram synthesis”. In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '19)*. Florence, Italy, Mar. 2019

This University Booth contribution summarizes the KIELER SCCharts tools.

- [HDM+13b] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer,

A.3. Other Publications

and Owen O'Brien. *SCCharts: Sequentially Constructive Statecharts*. Presentation at Synchronous Programming (SYNCHRON '13), Schloss Dagstuhl, Germany. Nov. 2013

The first concepts about sequential constructivity and SCCharts were demonstrated at the annually Synchron workshop in 2013. Eventually, the work resulted in the initial contribution on SCCharts [HDM+14].

[HDM+13a] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. *Compiling SCCharts to hardware and software*. Presentation at Synchronous Programming (SYNCHRON '13), Schloss Dagstuhl, Germany. Nov. 2013

The compilation approaches which were initially used by the KIELER SCCharts tools were also discussed at the Synchron workshop in 2013.

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

The long version of the first concept draft towards SCCharts are available online as Technical Report 1311.

[MSH+13] Christian Motika, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. *Sequentially Constructive Charts (SCCharts)*. Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF '13), Berkeley, CA, USA. Nov. 2013

The concepts which ultimately resulted in the initial contribution on SCCharts [HDM+14] were first publicly presented at the Ptolemy Miniconference in 2013.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and

A. Publications

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

This is the initial contribution of the SCCharts language. I contributed to the first two the compilation approaches for SCCharts, which are also used in the KIELER SCCharts tools. They are explained in Chapter 5. Note that the paper also gives information about the high-level transformations of SCCharts, which were part of the thesis from Motika [Mot17] in 2017 but not of this work.

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

While this work mainly illustrates how state-of-the-art tools can interactively help reactive system modellers to satisfy given deadlines, it also demonstrates the tool chain used to accomplish that goal. Fuhrmann continued the reactive system modelling research, which is not part of this work, and published the results elsewhere [Fuh17].

[FBS+14b] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. *Towards interactive timing analysis for designing reactive systems*. Tech. rep. UCB/EECS-2014-26. EECS Department, University of California, Berkeley, Apr. 2014. URL: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-26.html>

This technical report contains additional information on the reaction system modelling published elsewhere [FBS+14a].

[SSH+18b] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden, and Michael Mandler. “On reconciling concurrency, sequentiality and determinacy for reactive systems — a sequentially constructive circuit semantics for Esterel”. In: *2018 18th International Conference on Application of Concurrency to System Design (ACSD)*. June 2018, pp. 95–104. DOI: 10.1109/ACSD.2018.00018

A.3. Other Publications

This work reconciles the original partly speculative approach of sequential constructiveness with strict circuit semantic rules. While the semantic peculiarities of this work are of no concerns in this thesis, parts of the circuit generation tool chain, which is sketched out in Section 5.2.9, were reconciled during this work.

- [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

This publication explores new ways of object-oriented modelling. Objects use the aforementioned SDs to order method calls. Further pointers to this work are sketched out in Section 8.2.

- [PRS+16b] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. “Runtime enforcement of reactive systems using synchronous enforcers”. In: *ArXiv e-prints* (Dec. 2016). arXiv: 1612.05030 [cs.FL]. URL: <http://adsabs.harvard.edu/abs/2016arXiv161205030P>

The report on runtime enforcement, which eventually led to the presentation mentioned before.

- [PRS+16a] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. *Runtime enforcement of reactive systems using synchronous enforcers*. Presentation performed at the 23th International Open Workshop on Synchronous Programming (SYNCHRON '16), Bamberg, Germany. Dec. 2016

The Synchron workshop presentation of the runtime enforcement.

- [HMM+17] Reinhard von Hanxleden, Michael Mendler, Christian Motika, Christoph Daniel Schulze, and Steven Smyth. “SCCharts, KIELER and the Eclipse Layout Kernel—Statecharts for safety-critical applications and a pragmatics-aware modeling environment”. In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '17)*. Lausanne, Switzerland, Mar. 2017

A. Publications

This poster presentation from the University Booth at the DATE conference in 2017 gives an overview over the KIELER project (see Section 2.2).

[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. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2018

The technical report to the previously-mentioned publication on the reconciling of sequential constructivity [SSH+18b] gives further technical details and a proof sketch w.r.t. the semantics.

A.4 Supervised Theses

This section lists all theses which were supervised by me and explains their relation to the work presented here.

[Uml15] Axel Umland. “Konzept zur Erweiterung von SCCharts um Datenfluss”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf>. Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015

Umland implemented the first sketch of the SCCharts dataflow notation in the KIELER SCCharts tools. These first steps ultimately led to the SCCharts hybrid models that exist today. They are explained in detail in Section 6.2.

[Rat15] Karsten Rathlev. “From Esterel to SCL”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf>. Master thesis. Kiel University, Department of Computer Science, Mar. 2015

In his thesis, Rathlev combined Esterel with sequential constructiveness. The final product was named Sequentially Constructive Esterel (SCEst). The results led to the publications listed above [RSM+15; SMR+17].

[Sta15] Andreas Stange. “Comfortable SCCharts modeling for embedded systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-bt.pdf>. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2015

Stange added a project management (ProM), which improved the KIELER tooling beyond plain compilation tasks. ProM manages wrapper code generation, code deployment and refines the existing simulation [Mot09] of projects. Many of these tasks can now be declared and executed with the KIELER compiler as is discussed in Part I. Therefore, ProM is no longer in use today, but the lessons learned helped to realize KiCo.

[And15] Lewe Andersen. “Quadrocopter flight control design using SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

In their bachelor’s project, Andersen, Machaczek and Peiler, built a quadrocopter drone and used the SCCharts tooling to program the necessary microprocessors. The project gave helpful insights, which eventually improved the language as well as the overall tooling. Andersen, in particular, developed the flight controller of the copter.

[Mac15] Felix Machaczek. “Collision avoidance of safety-critical real-time systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fma-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

Machazek evaluated different methods of collision detection for the above-mentioned quadrocopter system. Eventually, an ultra-sonic system was implemented using diverse COTS hardware.

[Pei15] Lars Peiler. “Modeling simulations of autonomous, safety-critical systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

A. Publications

Peiler enhanced simulation of the tooling for the above-mentioned quadcopter system, which could then be used to evaluate the new SCCharts hybrid model approach [Uml15].

- [**Nas15**] Stanislaw Nasin. “Transformaion from SCCharts to Esterel”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sna-mt.pdf>. Master Thesis. Kiel University, Department of Computer Science, Oct. 2015

Nasin explored M2MT from SCCharts to Esterel.

- [**We15**] Tibor Weiß. “Von Nebenläufigkeit zur Parallelität in SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/twe-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Oct. 2015

Weiss implemented the SCPDG to explore parallelism in SCCharts and synchronous languages in general. The SCPDG is a program representation that can be used to archive maximum possible static parallelism. Refinements of this representation are used in Section 5.2.4.

- [**Ols16**] Lars Olsson. “Modellextraktion aus C code”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/leo-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2016

Olsson explored the capabilities of SCCharts to serve as an extraction language for legacy C programs. The topic was later deepened by Lenga [Len16].

- [**Fl16**] Niclas Flieger. “Comparison of compilation approaches in KIELER”. Master thesis. Kiel University, Department of Computer Science, Apr. 2016

Flieger implemented tools to compare different compilation approach. Although the original implementation is no longer in use in KIELER, some insights have been used to create the comparison discussed in Section 5.5.

- [**Bus16**] Jonas Busse. “SCCharts modeling for embedded systems with

limited resources”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbus-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016

Busse optimized the sequential form of the SCG in the netlist-based compilation, see Section 5.2, with compiler optimizations, such as *copy propagation* [ASU86] and *register relocation* [CAC+81]. Section 5.2.8 explains more on the optimization of the netlist-based approach.

- [Sch16]** Alexander Schulz-Rosengarten. “Strict sequential constructiveness”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf>. Master thesis. Kiel University, Department of Computer Science, Sept. 2016

Schulz-Rosengarten developed a non-speculative variant of SCCharts. This foundation led to the publications about the reconciled semantics of SCCharts [SSH+18b; SSH+18a]. They are related to the sequentially constructive Esterel approach discussed in Section 6.4, but the work itself is not part of this thesis.

- [Len16]** Stephan Lenga. “Model-based compilation of legacy C programs”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sle-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016

Lenga explored further possibilities of SCCharts as extraction languages for legacy C programs. Section 6.5 discusses the results.

- [Rah17]** Milad Rahimi-Barfeh. “Incremental compilation of SCEst”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mrb-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2017

Rahimi-Barfeh implemented an incremental compiler for SCEst using the newest KIELER compiler technologies, see Part I. The approach also makes use of dynamic compilation models, which are discussed in more detail in Section 3.3.5.

- [Pei17]** Lars Peiler. “Priority-based compilation of SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/pei-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2017

A. Publications

informatik.uni-kiel.de/~biblio/downloads/theses/lpe-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Oct. 2017

Peiler implemented the priority-based compilation approach [Han09b] within the KIELER SCCharts tools using the new compiler technology, see Part I. The priority-based approach is explained in Section 5.3.

- [Eum17] Philip Eumann. “A domain-specific language for railway control”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-bt.pdf>. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2017

This work presents an abstract language for describing train schedules. The language targets pupils and non-students that solve tasks at a railway demonstrator during public events at the University. The railway installation is described in ??.

- [Sta19] Andreas Stange. “Model checking for SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, May 2019

This thesis explored model checking SCCharts within the KIELER SCCharts tools. Section 7.1 gives a practical example on model checking in KIELER, which is realized with the newest version of KiCo.

- [And19] Lewe Andersen. “Dataflow and statemachine extraction from C code”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, Dec. 2019

This thesis explores model extraction from standard code patterns for statemachines. Section 6.5 outlines the C extraction topic.

- [Eum20] Philip Eumann. “Runtime debugging of SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-mt.pdf>. Masters’s thesis. Kiel University, Department of Computer Science, Mar. 2020

A.4. Supervised Theses

Runtime debugging of code generated from SCCharts as implemented in this work is shown in Section 8.2.2.

Related Modelling Tools

B.1 UML Statecharts

The UML [Gro15] is an object modelling language that defines a comprehensive set of notations and semantics. State machines are the primary means within the UML to model complex dynamic behaviour [Dou99]. An example of a UML statechart modelled in Rhapsody is depicted in Figure B.1.1. Since Eran Gery and David Harel took part in defining the meaning of UML statecharts [Har09], one can spot many similarities to Harel's statecharts presented in Section 2.3.4 and what is implemented in Rhapsody. Rhapsody can be seen as an executable kernel of the UML [HK04]. There exist other semantics for UML statecharts, based on Structured Operational Semantics (SOS) [Bee02; Plo81], Petri nets [ABC16] or message-passing [Jür02]. Regarding the runtime MoC, the important difference in the UML Statechart execution towards the classical synchronous MoC according to Berry is the *run-to-completion* semantics. Once an event is triggered, its corresponding actions are executed until completed. Similar to statecharts, defining concise semantics seems to be hard. Fechner et al. listed 29 new unclarities in the semantics of the UML 2.0 [FSK+05], particularly with history, priority and entry and exit actions.

UML supports a second formalism for state machines, namely *activity diagrams*. They differ from statecharts mainly in the situations they are applied: States in activity diagrams are changed when the executions of actions finish.

B. Related Modelling Tools

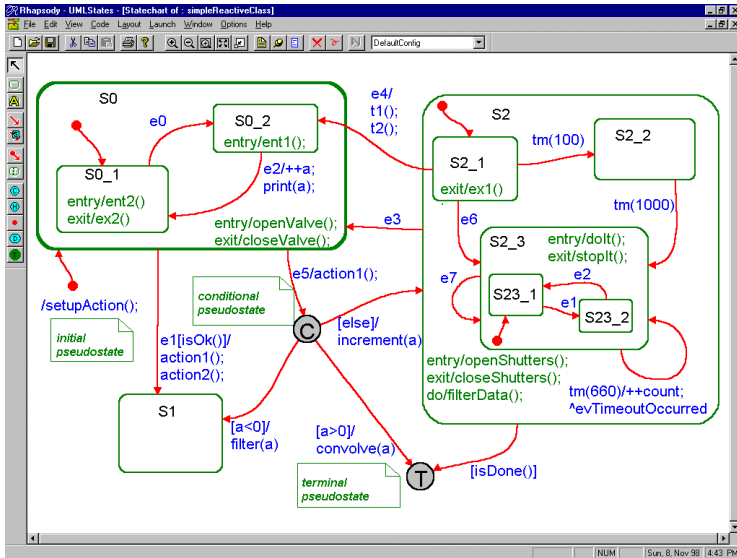


Figure B.1.1. Example of an UML statechart modelled in Rhapsody (from [Dou99])

B.2 Simulink & Stateflow

MathWorks Simulink is another wide-spread industrial tool for modelling and testing. It provides multi-domain models for various tasks with works together with the prominent MATLAB tool family¹. Figure B.2.1 shows two examples. In Figure B.2.1a a model for a helicopter control system is depicted. Typical elements of control systems are dataflow actors and continuous semantics, e.g. for PID controllers. The second example in Figure B.2.1b shows a block diagram for a driver assistance system. Car panel elements can be embedded into the model.

Simulink can be combined with other modelling tools which support different engineering disciplines. Hooman et al. [HMP04] coupled Rose RT

¹ <https://mathworks.com/products/matlab.html>

² Available at the Simulink homepage: <https://mathworks.com/products/simulink.html> (accessed: 17.11.2019)

B. Related Modelling Tools

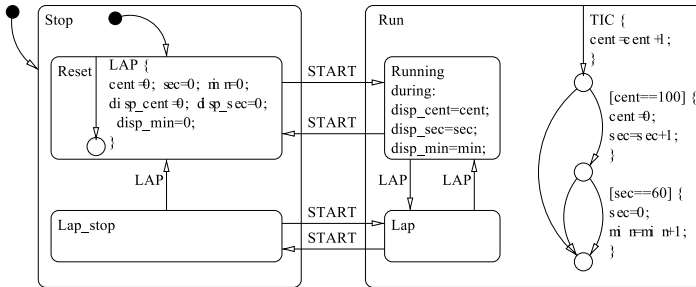


Figure B.2.2. Stopwatch example in Stateflow (from [HR04])

There are similarities to Harel's statecharts, such as hierarchy-crossing transitions between the superstates. Additionally, the charts contains a notion for flowcharts. Unlike states, junction points in flowcharts are exited immediately and proceed their execution until a junction without eligible outgoing transition is reached. This behaviour can be modelled in SyncCharts (and SCCharts) with immediate transitions and parallel regions.

Agrawal et al. [ASK04] transform Simulink Stateflow models to Hybrid Automata to ease model verification. Therefore, they used the model-based graph transformation tool GREAT to transform a defined subset of Stateflow, excluding procedural components, into the hybrid system interchange format (HSIF), which is an XML-based standard to represent dynamic networks of hybrid automata. The transformation is validated via testing. Future work for this approach lists the formal verification of the transformation.

B.3 Ptolemy II

The Ptolemy II project explores heterogeneous hierarchical models [EJL+03]. As before, different, e.g., engineering aspects of system development may comprise different characteristics for each domain. Usually, experts of a certain domain use means which are natural to them to design their subsystem. For example, modelling mechanical aspects is commonly done via ordinary differential equations in a continuous domain. In Ptolemy, every hierarchy level is managed by a (possibly different) *director*, which dictates the MoC.

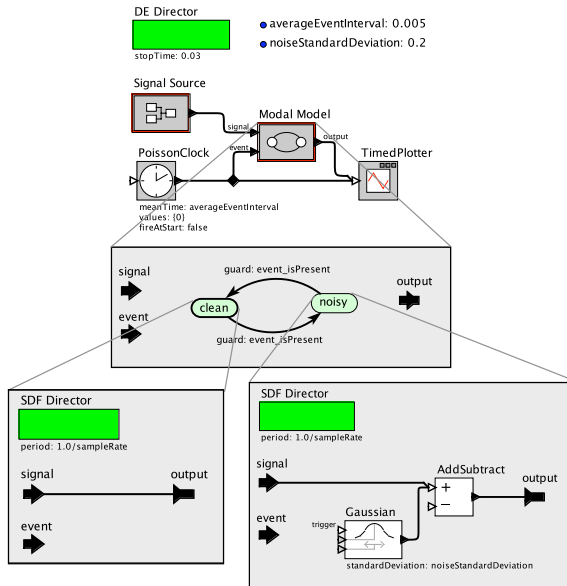


Figure B.3.1. Hierarchical Ptolemy II example with different directors (from [Lee09])

Within their hierarchical encapsulation the semantics are well-defined and usual analyses can be performed. Together with the directors, *receiver* manage the actor communication between the hierarchy levels. This hierarchical heterogeneity provides a more structured approach that can combine various MoCs. It also facilitates modularity via the use of polymorphic actors which can live in different domains. By default, Ptolemy supports domains such as continuous time (CT) for mechanical dynamics, discrete events (DE) for, e. g. digital circuits, synchronous/reactive (SR) for discrete time models, communicating sequential processes (CSP) for sequential message-passing models, Kahn's process networks (PN), synchronous dataflow (SDF), boolean dataflow and cyclo-static dataflow (CSDF).

Figure B.3.1 shows an example of a hierarchical ptolemy model. While the hierarchy is opened in different windows in the live editor, a common way of displaying this kind of diagrams is by *exploding diagrams*, where the

B. Related Modelling Tools

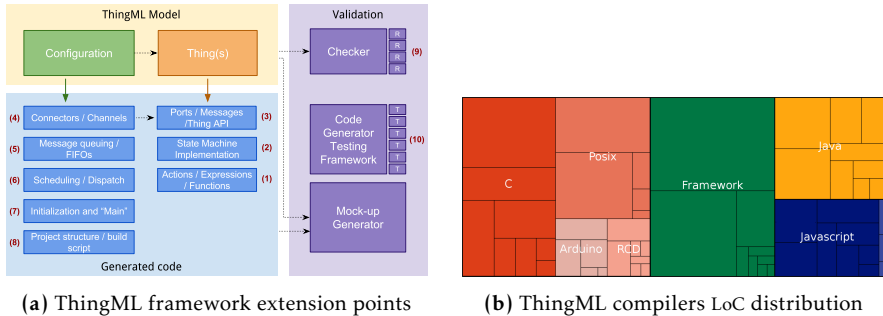


Figure B.4.1. ThingML framework extension points and compilers LoC distribution (from [HFM+16]).

inner behaviour is displayed on the side of the parent chart, connected by corner edges. On the top hierarchy level, a DE director directs a discrete event MoC. The level contains a finite state machine (FSM) actor, which can be expanded to inspect the included state machine. The states inside the machine contain refinements as well. These refinements are given by synchronous dataflow models.

SCCharts only follows the SCMoC. However, variations can be implemented easily by altering the compilation chain. One example where the behaviour of the control-flow towards the scheduling of the program is changed is explained in Section 5.2.6. Furthermore, SCCharts separation between core and extended SCCharts makes it easy to implement new extended features on top of the core language. Section 6.2 shows how the dataflow paradigm can be added to SCCharts as extended feature, enabling modelling concurrent assignments and also control-dataflow hybrid models. These models are then compiled to Core SCCharts control-flow models eventually and processed by the usual compilation chains.

B.4 ThingML

ThingML [HFM+16] is composed of a modeling language and a highly customizable compiler tool, primarily developed for embedded systems. It

was created in collaboration with the industry and is another attempt to bring the supposed benefits of MDE into practice. It has been used in medical systems for elderly people in the industrial context.

The modelling language is separated in *things*, which represent software components, and *configurations*, which describe their interconnections. Things' behaviour is implemented using a mix of imperative programming, simple if-then rules, and composite state machines, conforming the UML statecharts. The code generator in ThingML is a family of Model-to-Text Transformation (M2TT) each targeting a specific language. It has been used to generate code for C/C++, Java and JavaScript, targeting various different platforms ranging from micro controllers to servers. The compiler transforms a configuration into code. It is highly customizable and offers extension points for every major aspect of the compilation process, as can be seen in Figure B.4.1a. The framework is implemented in Java. The LoCs distribution for the different compilers is shown in Figure B.4.1b. Dialects of existing compilers, e. g. C and Arduino C, require a significantly smaller amount of LoCs.

While being highly customizable in every step of the compilation, the structure of the compilation and the steps involved are relatively fixed in ThingML and tailored to the needs of the ThingML DSL. KIELER does not restrict a compiler developer in the use of meta-models or the way artefacts are transformed. Interconnections between components can also be modelled and transformed in the same way, as is shown in Section 6.2, which renders specialized extension unnecessary in KIELER. The model-based compilation realized in KiCo stays in a meta-model as long as possible to facilitate understandability. Intermediate transformations should result in models which domain experts can read. Consequently, what is considered a code generator in KIELER is basically a serialization of the final model, which is usually a relatively simple step compared to the preceding transformations. For example, when compiling an SCChart model in KIELER to C, Java, or Arduino C, the transformations involved stay the same most of the time. Only the last serialization step has to change, with the Java and Arduino C processor extending the existing one for C. Moreover, the contributions of different compiler parts, similar to Figure B.4.1b, can be visualized automatically in the KiCo framework, because the compiler structure is modelled. Hence, tool

B. Related Modelling Tools

developers can get an instant overview over the current state of the compiler and all possible compilation paths, which is demonstrated in Section 4.2.

The ThingML team reflects that, contrary to the promises of MDE, automatic code generation is adopted relatively slow. As reasons they see that models and modelling languages are often not suited for code generation and code generators are often seen as blackboxes which are not easy to trust and produce sub-optimal code [HFM+16]. This has also been observed by Motika [Mot17] w.r.t. blackbox compiler. In the lessons learned, Harrand et al. reported that despite the fact that UML was encouraged by decision makers, the results in the embedded domain were impractical and counter-productive. On the other hand, textual DSLs proved easy to use by the developers. Graphically specifying complex state machines and writing actions in C was seen as drawback. This was also true for the high amount of customization needed for code generators for different target platforms, although it was written in Scala using a set of templates and helpers. Expert knowledge were indispensable and code duplication could not be avoided.

Harrand et al. concluded that having a single integrated language with a clear semantics together with dedicated tool support was a key to success. More research and tooling is necessary before combinations of different models and formalisms turn out to be practical. MDE should not introduce any overhead when no benefit is expected or desired and code generation is not popular among practitioners due to bad reputation based on experiences with tools producing code with low readability and high integration costs.

While KiCo is agnostic towards its artefacts, the KIELER SCCharts editor serves as example for a statecharts dialect with clear and strict synchronous semantics and a generic but custom-tailored tool support. The KIELER team mostly concurs with the observations and the conclusion of the ThingML team and we admit that by choosing a more generic approach, more knowledge is necessary on the compiler developers side. Hence, we thoroughly separated between the different roles of developers as is discussed in Section 3.2.3. MDE can increase productivity if done correctly. However, decision makers' choices, the tool jungle, black boxes and occasionally bad reputation make it difficult to choose what is the best tool (resp. language) and stay focused.

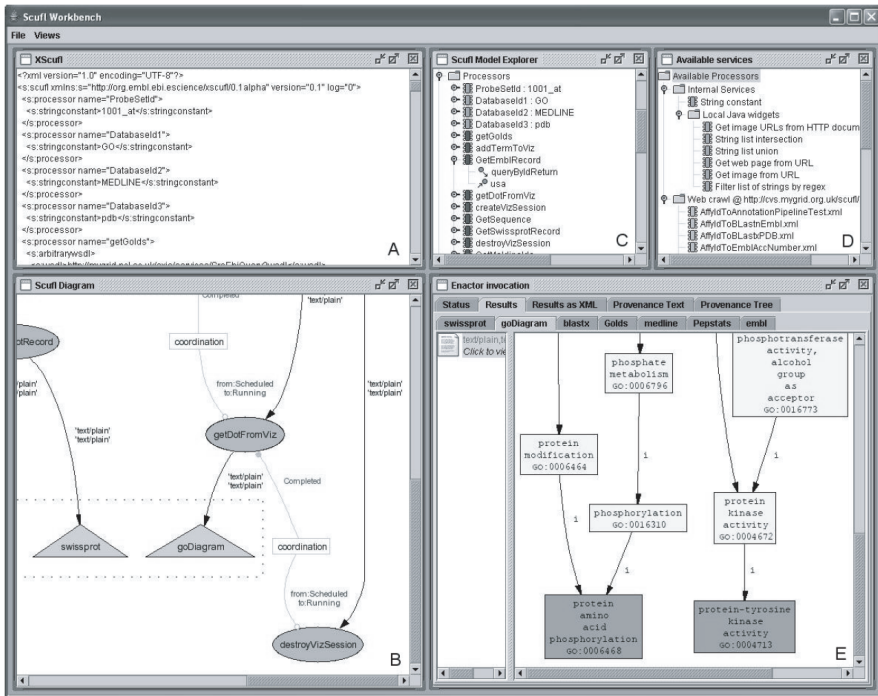


Figure B.5.1. Taverna Workbench [OAF+04]

B.5 Scientific Workflows

The process systems of the KIELER compiler can be seen as a variant of *scientific workflows* [CGG10] for M2MTs combined with state-of-the-art pragmatic modeling techniques. While the scientific workflows descriptions map into program executions, often for subtasks that work on remote resources. There are similarities in the orchestration, presentation and the overall life cycle of these workflows, even though KIELER is specialized in M2MTs and interactivity. Equal to the experiences gathered during the development of KIELER, Curcin et al. observed an iterative refinement process during the life cycle of a workflow. Developers review individual fragments of the workflow to

B. Related Modelling Tools

modify the functionality or to improve the performance. After the development completes, the workflow is deployed for execution. For KIELER, the refinement cycle is discussed in detail in Section 4.1.

An example of a scientific workflow system's implementation is TAV-ERNA [OAF+04; BWC+08]. Taverna is used to orchestrate local and web services in bioinformatics in so called workflows. To describe such a workflow the dedicated language Sculf (simple conceptual unified flow language) is used. Each process of a workflow represents an atomic task and can be categorized into different types, such as nested other workflows. A processor in Taverna is a classical transformation. Outputs are generated depending on the inputs.

The loose processor concept, type-checking and executable pre-defined workflows are comparable with the KIELER compiler concept, which will be introduced in the next chapter. However, the focus of KiCo lies on M2MT where every intermediate result is a fully functional artefact. KiCos processor system itself, including the environments in a running context, is also considered *just* a simple model here. There is no need for a specialized description language or special data storages, e. g. for processor meta-data such as processor run-time. In our approach, the system's model can be observed and modified during design- and run-time, which includes alterations by the contained processors. Furthermore, as long as the transient view framework supports the meta-model of the intermediate results, views can be generated instantaneously and there is no difference between the different artefacts, even if they are positioned on distinct meta-levels. This also includes the meta-model of the compiler itself. It is not handled differently.

To visualize the model, which is stored as Extensible Markup Language (XML) file, the Taverna workbench uses Dot. Dot is an automatic layout engine for graphs, which is similar to ELK and can also be utilized within KIELER. However, Taverna creates static images from the model description and does not employ an interactive, transient viewing framework, such as KLightD.

Other concepts such as web services and service discovery are not implemented in KIELER, but could prove to be useful additions. Especially as model-based compiler services are also moving from platforms such as Eclipse towards web applications as is discussed in Section 8.2. This is an-

other step towards unified models as envisioned by Steffen [Ste97]. While Model Checking (MC) plays a role in checking the consistency of scientific workflows themselves, MC on M2MT compiler descriptions in KIELER is usually not required, because the compilation chains are rather clear. The interacting compiler processors in KIELER are type-checked though. However, checking model properties on SCCharts models is useful and possible, e. g. by using NuSMV³ [CCG+02], in KIELER.

³<http://nusmv.fbk.eu>

Appendix C

SCCharts iMURD

For SCCharts, the views presented in Section 4.1 form an iMURD loop. It is depicted in the accompanying poster of the guidance presentation [[SSH18a](#)], which can be seen in Figure C.0.1.

C. SCCharts iMURD

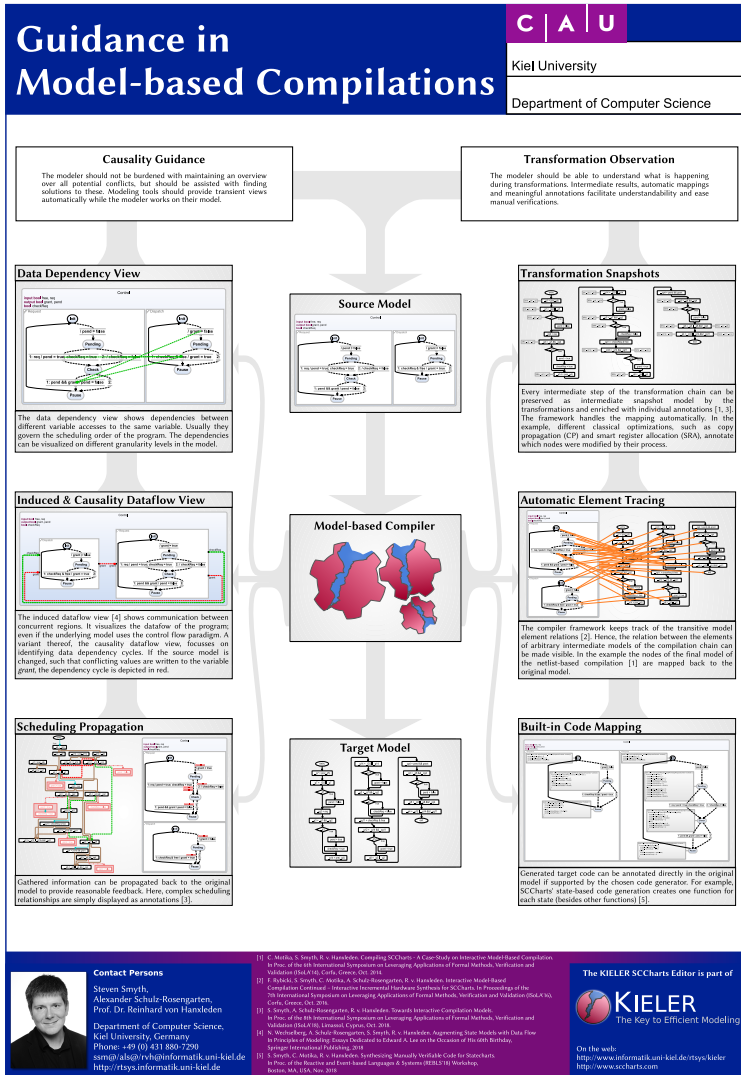


Figure C.0.1. Guidance in Model-based Compilation Poster [SSH18a], demonstrating the iMURD cycle in SCCharts

Extended SCCharts Features

This chapter discusses extended features of SCCharts which have not been covered in previous works. Specifically, Section D.1 explains *referenced SCCharts*, Section D.2 introduces *probabilistic priority transitions*, and *for regions* are shown in Section D.3.

D.1 Referenced SCCharts

Arguably one of the SCCharts additions with the biggest impact is the *referenced SCCharts* feature, the newly introduced modularity concept of SCCharts. Modules are indispensable to organize large projects, divide projects into smaller sub-projects and share work units between a team. SCCharts supports two different kinds of modules, namely *model expansion* and *module calls*. Expansion embeds referenced model in the root model and proceeds with the compilation as described in Chapter 5. While compiler analyses, such as the dependency analysis, can be applied easily as usual, this approach comes at the cost of inefficient scalability, because all instances of referenced SCCharts are expanded into one model. The *module calls* approach creates exactly one code base for each SCChart. Different instances of the models are called with their context stored in dedicated variables. However, since inter-thread communication between different instance of the same model may depend on the concrete context, dependency analyses is more complex. If already compiled modules should be supported as well, the inner behaviour of the model can be seen as a blackbox and is hidden during the compilation of the root model, and therefore, makes interleaving difficult. While potential write and read accesses can be managed via the input and output interface of a model, more complicated scheduling proto-

D. Extended SCCharts Features

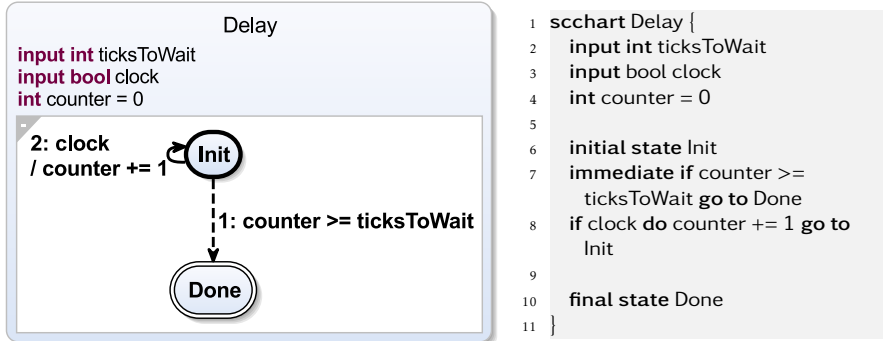


Figure D.1.1. Clock example: Delay SCChart

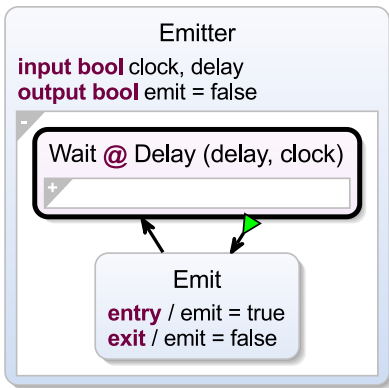
cols, such as the IURP, may need more sophisticated *contracts* between caller and callee.

D.1.1 Model Expansion

Referenced SCCharts using expansion are similar to the referencing mechanism provided by SyncCharts or Esterel. However, using an interactive modelling IDE, the tooling can help the modeller with necessary *bindings* of the interface. A *binding* binds a variable from the local scope which contains a reference to another SCChart to a remote variable of the interface of the referenced SCChart. The following clock example explains the different aspects of this feature. The first model delay, see Figure D.1.1, waits a certain amount of clock ticks before it proceeds. The number of ticks to wait in `ticksToWait` and the clock signal `clock` both come from the environment. Therefore, the waiting time and the frequency of the tick interval can be adjusted externally to the model. In the beginning counter is initialized with zero and the model starts at the `Init` state. If the counter value reaches `ticksToWait`, the `Done` states becomes active and the SCChart terminates. Otherwise, the `Init` state stays active and if a clock signal occurs, the counter is incremented.

Definition

D.1. Referenced SCCharts



```
1 import Delay
2
3 scchart Emitter {
4   input bool clock, delay
5   output bool emit = false
6
7   initial state Wait
8   is Delay(delay, clock)
9   join to Emit
10
11  state Emit {
12    entry do emit = true
13    exit do emit = false
14  }
15  go to Wait
16 }
```

Figure D.1.2. Clock example: Emitter SCChart

A second model Emitter, see Figure D.1.2, should wait a specific amount of clock ticks and emits a signal for one tick if the waiting time has passed. Therefore, Emitter can make use of Delay. A new state is declared in line 7 as usual. However, the state references another SCChart with the keyword `is` in line 8. The interface of that reference must be bound to variables of the local scope. In line 8, Emitter binds the clock of delay to the local clock variable and ticksToWait to the local object delay. Apart from the link to the referenced SCChart the state behaves like a normal *superstate*. Therefore, once the delay state finishes, the Emit state becomes active for one tick and sets the Emit variable to true and the cycle starts over again.

Although the clock to clock binding is done explicitly and recommended for clarity, bindings of objects with the same name is not mandatory. The reference transformation searches for valid objects and generates an implicit binding if the explicit binding is omitted. However, implicit bindings generate a warning in the IDE.

Finally, a third model clock, see Figure D.1.3, simulates a clock. It receives a millisecond signal in `msClock` as input and sends signals for seconds, minutes and hours itself. Therefore, it comprises three concurrent regions and uses the emitter model to accomplish the task. In the region seconds

D. Extended SCCharts Features

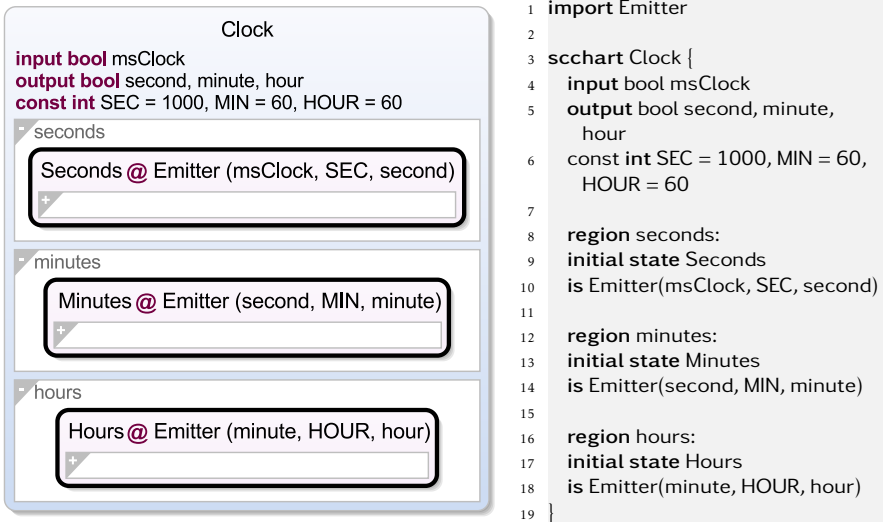


Figure D.1.3. Clock example: clock

the clock of the emitter is bound to the millisecond clock, the delay is bound to the necessary interval for one second stored in the constant SEC and the emit variable should be second. Similarly, the regions minutes and hours manage the outputs minute and hour. The clock is linked to the generated seconds impulses and delay is bound to MIN, respectively HOUR.

The SCChart clock uses the emitter model three times, which itself uses the delay model. The fully expanded version of clock is depicted in Figure D.1.4. The transformation chain proceeds as usual after the expansion. Since a link to a referenced SCChart results in the expansion of the target SCChart into the actual model and therefore in an inclusion of arbitrary (extended) SCCharts features, it is mandatory that the referenced SCCharts transformation is executed before any other transformation is invoked.

D.1. Referenced SCCharts

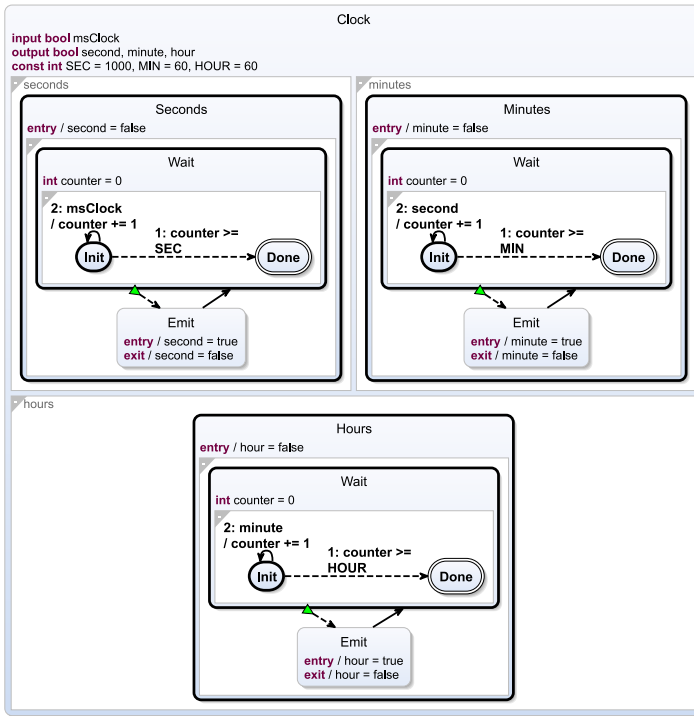


Figure D.1.4. Clock example: fully expanded version of clock

Details about Bindings There are three different kinds of bindings: *explicit bindings*, *order bindings*, and *implicit bindings*.

Explicit bindings state explicitly which local variable from the current scope should be bound to which variable in the remote interface. They are considered the most safe kind of binding, because if the interface or variable names change, the previous binding is not valid any longer. The content

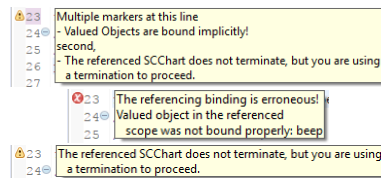


Figure D.1.5. Editor binding warnings

D. Extended SCCharts Features

assist should pick up erroneous bindings and the modeller must correct the configuration.

```
initial state wait is Delay(delay to ticksToWait, ^clock to ^clock) join to emit
```

Listing D.1.1. Explicitly bound local scope variables to the interface of the referenced SCChart

If the order of the variables is known (e. g. through the content assist), one can also use order bindings and simply state the list of local variables. They will be bound to the interface in the order of declaration in the remote SCChart. However, be aware of the fact that the bindings will change, when the declarations of the referenced SCChart change. This type of bindings was used in the clock example.

```
initial state wait is Delay(delay, ^clock) join to emit
```

Listing D.1.2. Local scope variables bound by order to the interface of the referenced SCChart

If the names of the local and referenced variables are identical, one can omit the binding completely. The variables are then bound implicitly. However, this kind of binding is considered unsafe and the editor warns the modeller about any implicit bindings. They can be convenient though when dealing with large interfaces, such as the interfaces of the railway projects explained in Section 7.3.2, which in parts consists of over 40 variables. In Listing D.1.3, `delay` is bound explicitly to `ticksToWait`. However, since the local and the remote clock variable share the same name, they are linked inherently and the binding can be omitted. This prompts an IDE warning.

```
initial state wait is Delay(delay to ticksToWait) join to emit
```

Listing D.1.3. Variables with the same name may be bound implicitly if not explicitly stated

As any state, referenced states are allowed to own transitions. The modeller must make sure that the referenced SCChart terminates if they use terminations; otherwise, the transition will never fire. The content assist helps with this issue. Different kinds of warning and error messages can be seen in Figure D.1.5.

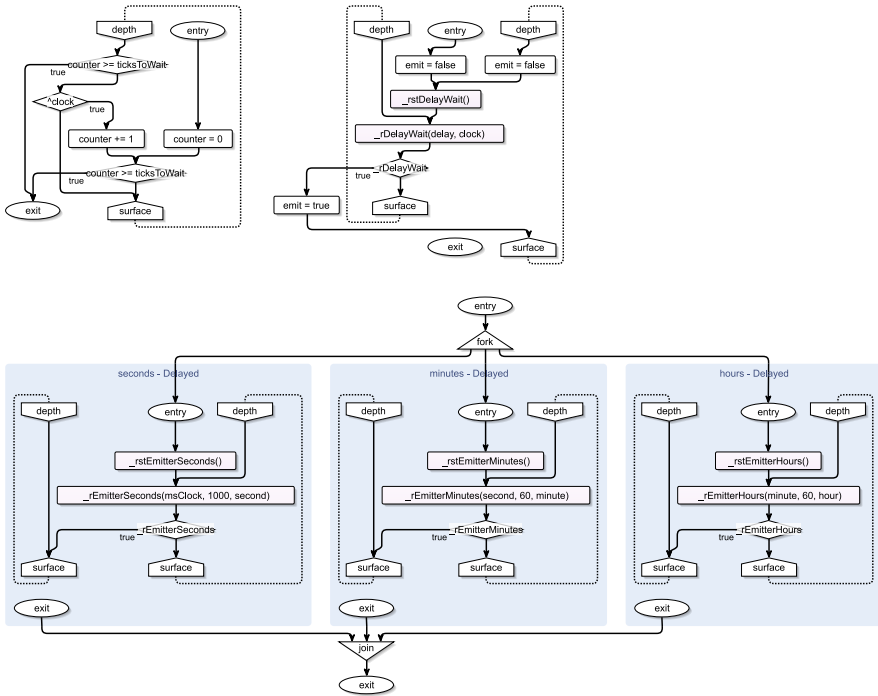
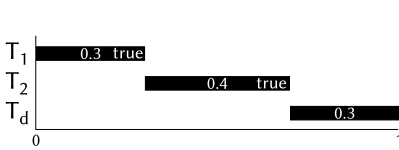


Figure D.1.6. Clock example: module call version of clock

D.1.2 Module Calls

Instead of expanding all instances into one model, the source codes for each model can be generated independently. Contrary to the immediate expansion described in Section D.1.1, the model contents will not be embedded into the root model. Instead, the references are carried down within the compilation chain until they reach the SCG. The generated SCG for the clock example is depicted in Figure D.1.6. A separate SCG exists for each model. References to other SCGs are colour-coded in the same way as before in the SCCharts synthesis. The module calls are prefixed with `_r`. The instance reset is prefixed with `_r`. After a reference call, the conditional checks if the

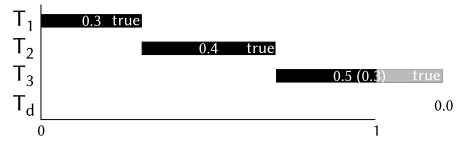
D. Extended SCCharts Features



(a) Two PrPT with modelled probability 0.3 and 0.4; if both trigger evaluate to true, the default transition has a inherent probability of 0.3



(b) If T_2 cannot fire due to its trigger, the T_d has a probability of 0.7.



(c) If the overall probability of all active transitions exceeds 1, the effective probability is lower than the modelled probability (here 0.3 instead of 0.5 for T_3) and the probability of T_d is 0.0.



(d) If the triggers of T_1 and T_2 evaluate to false, the effective probabilities of T_3 and T_d rise to 0.5.

Figure D.2.1. Modelled and effective probabilities in probabilistic priority transitions

instance has terminated to proceed with the potentially normal terminations, e. g. in Emitter. Whenever an instance is re-entered, the reset function for that instance must be called. Each SCG is compiled individually as explained in Chapter 5.

While this approach scales better w.r.t. code size if models are referenced multiple times and enables incremental compilation, it restricts bidirectional inter-thread communication between regions of different referenced SCCharts, because the references are called as monolithic units. In KIELER, the input and output variables in the interfaces of the referenced SCCharts define the scheduling order. They are not permitted to form cycles. More sophisticated scheduling approaches w.r.t. blackbox objects are researched by Schulz-Rosengarten [SSM19] and briefly described in Section 8.2.

D.2 Probabilistic Priority Transitions

Modelling chance is particularly helpful when creating models for environments. While some modelling tools, such a Ptolemy, support non-

D.2. Probabilistic Priority Transitions

deterministic transitions, giving concrete probabilities can help to create more realistic scenarios. Probabilistic properties can be added as extended features via KiCo as long as the underlying expression language supports some kind of randomness. The modeller can assign probabilities to transitions in SCCharts turning them into *probabilistic priority transitions* (PrPTs).

PrPTs extend SCCharts based on PAs [Rab63]. The transition matrix in PAs defines row vectors, whose components determine the probability of the outgoing transitions. The sum of these components is equal to 1. It is possible to implement exactly this behaviour in SCCharts, e. g. via connectors and a validation that ensures that the sum of all outgoing probabilities is 1. However, as transitions in SCCharts are guarded by triggers and ordered by priorities, I present an alternative intuitive approach here. A PrPT is guarded by its trigger and its probability. The probability is combined with the trigger of the transition. Additionally, the default transition is always $\max(0, 1 - \sum_t p(t))$ for all explicit outgoing transitions $t \in T$ with $p : T \rightarrow [0, 1]$ returning the probability of t if t is enabled or 0 otherwise. Thus, the default transition is taken if no other transition is eligible to run.

For each active state with eligible PrPTs an active transition will be selected in each tick. To keep things simple for the modeller, all probabilities are absolute values. The sum of all probabilities of a row vector can be greater than 1. However, the state will always decide for a probability within $[0, 1]$ and therefore the *effective probability* may divert from the probability modelled by the developer.

The *effective probability* of a transition determines the probability of a transition firing in the current tick. Definition

Figure D.2.1 illustrates how modelled and effective probabilities in PrPTs may differ. Figure D.2.1a shows three transitions: two PrPTs with modelled probabilities 0.3 for T_1 and 0.4 for T_2 . Therefore, if both trigger evaluate to true, the probability of the default transition T_d , which can be an implicit self-loop, is 0.3. In Figure D.2.1b the trigger of T_2 becomes false and the probability of T_d rises to 0.7. Since the overall probability is restricted to the interval $[0, 1]$, it is possible to achieve a lower effective probability than modelled. In Figure D.2.1c a third PrPT T_3 is added with a modelled probability of 0.5. However, the overall probability exceeds 1, and therefore, the effective probability of T_3 is 0.3 if all trigger evaluate to true. In this case, T_d

D. Extended SCCharts Features

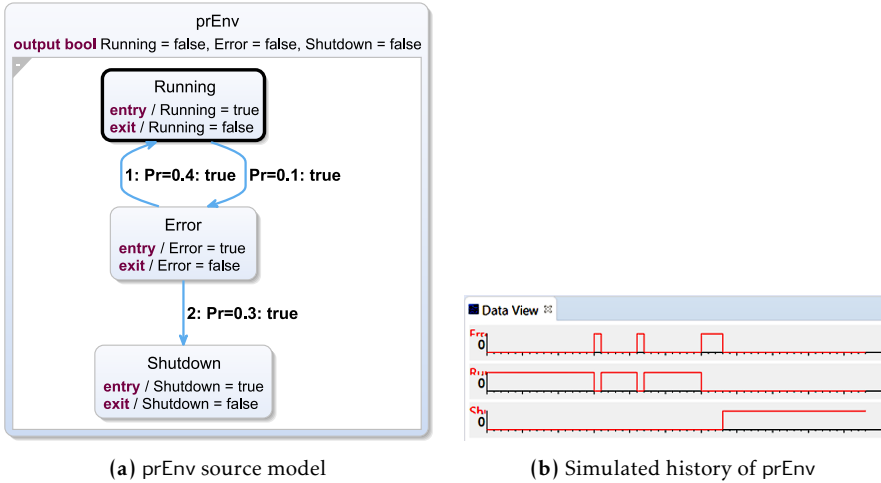


Figure D.2.2. PrPT example: prEnv

has a probability of 0.0 and is not considered further. If T_2 cannot fire due to its trigger, the T_d has a probability of 0.7. As soon as the triggers of T_1 and T_2 evaluate to false, the effective probabilities of T_3 and T_d rise to 0.5, which is illustrated in Figure D.2.1d.

Example Figure D.2.2 shows an example of the PrPT. In controller example there is often some kind of error state or handling. Figure D.2.2a shows a model with three state, Running, Error, and Shutdown. In its usual operation mode, the model resides in the Running state. However, there is a probability of 0.1 that the system changes to the Error state. In the error state, it is simulated that the issue could be fixed with a probability of 0.4 or that the system has to be shut down with a probability of 0.3. Otherwise, a decision must still be determined and the system remains in the Error state. A simulation run of prEnv can be seen in Figure D.2.2b.

Implementation The expansion of the probabilistic transitions is shown in Figure D.2.3. Technically, to model the probability, each state with outgoing probabilistic transitions generates a random value $r \in [0, 1]$. First, if the trigger of the transition is true, it is also checked whether the transition probability is

D.2. Probabilistic Priority Transitions

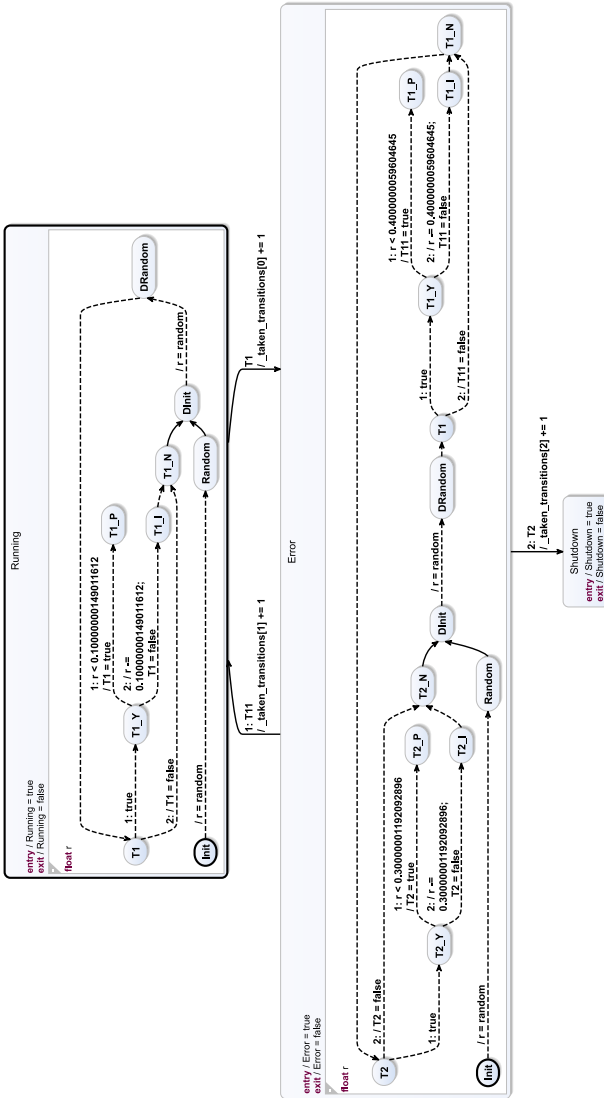


Figure D.2.3. Expanded prEnv model from Figure D.2.2a

D. Extended SCCharts Features

```

1 scchart For {
2   input bool I
3   output bool A[2] = {0, 0}
4
5   region set for N:0 to 1:
6
7   initial state Wait
8   if I do A[N] = true go to Set
9
10  final state Set
11 }

```

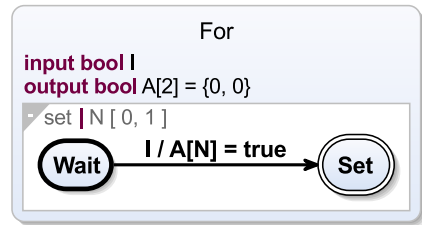
(a) for with an explicitly set span

```

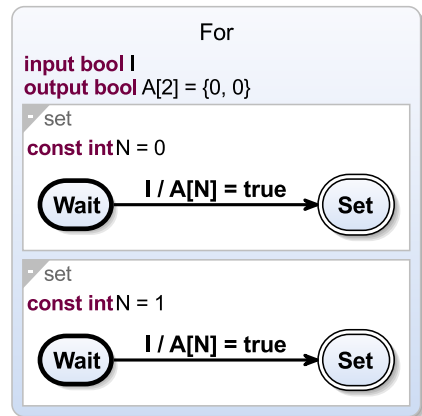
1 scchart For {
2   input bool I
3   output bool A[2] = {0, 0}
4
5   region set for N:A:
6
7   initial state Wait
8   if I do A[N] = true go to Set
9
10  final state Set
11 }

```

(b) for with a span defined by an array



(c) Graphical syntax of for



(d) for model after the expansion

Figure D.3.1. The *for* region feature in Extended SCCharts

matched. If so, the according transition is fired. Otherwise, the transition probability is subtracted from r and the next transition is checked.

D.3 For

For is a feature which saves repetitive work w.r.t. managing large variables fields. It duplicates the behaviour of a region. Each region receives its own counter variable.

Example Figure D.3.1 shows an example of *for*. In the textual syntax, the feature

can be activated by adding the keyword `for` to a region. It is followed by an iterator variable and either an integer span, as shown in Figure D.3.1a, or an array, illustrated in Figure D.3.1b, which defines the span inherently. The graphical syntax of `For` is identical, as shown in Figure D.3.1c. The `KiCo for` processor expands the feature into distinct regions. A new region with constant iterator variable is created for each iteration, which is illustrated in Figure D.3.1d

The feature is particularly helpful when dealing with large sets of data, e. g. in the context of the railway interface, which is discussed in Section 7.3.2. However, new object-oriented flavours added to `SCCharts`, which also permit the execution of instantaneous imperative loops, may render this addition w.r.t. to variable sets obsolete in the future. They are outlined in Section 8.2. Nevertheless, the feature can still be used to duplicate inner behaviour.

Bibliography

- [ABC16] Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy. “Formalising concurrent UML state machines using coloured petri nets”. In: *Formal Aspects of Computing* 28.5 (Sept. 2016), pp. 805–845. ISSN: 1433-299X. DOI: 10.1007/s00165-016-0388-9.
- [Abr96] Jean-Raymond Abrial. “Steam-boiler control specification problem”. In: *Formal Methods for Industrial Applications*. Springer, 1996, pp. 500–509.
- [All02] Tiziana Allegrini. “Code generation starting from statecharts specified in UML”. MA thesis. Università Degli Di Pisa, Facoltà di Ingegneria, 2002.
- [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. URL: <http://doi.acm.org/10.1145/800028.808479>.
- [Ame10] Torsten Amende. “Synthese von SC-Code aus SyncCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf>. Diploma thesis. Kiel University, Department of Computer Science, May 2010.
- [AMH+14] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. “Grounding synchronous deterministic concurrency in sequential programming”. In: *Proceedings of the 23rd European Symposium on Programming (ESOP’14), LNCS 8410*. Long version: Technical Report 94, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, Bamberg University, August 2014, ISSN 0937-3349. Grenoble, France: Springer, Apr. 2014, pp. 229–248.

Bibliography

- [And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.
- [And04] Charles André. “Computing SyncCharts reactions”. In: *Electr. Notes Theor. Comput. Sci.* 88 (2004), pp. 3–19.
- [And15] Lewe Andersen. “Quadrocopter flight control design using SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.
- [And19] Lewe Andersen. “Dataflow and statemachine extraction from C code”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, Dec. 2019.
- [And95] Charles André. *Synccharts: A visual representation of reactive behaviors*. Tech. rep. Université de Nice-Sophia Antipolis, Oct. 1995.
- [And96a] Charles André. “Representation and analysis of reactive behaviors: A synchronous approach”. In: *Computational Engineering in Systems Applications (CESA)*. Lille, France: IEEE-SMC, July 1996, pp. 19–29.
- [And96b] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.
- [ARG10] Sidharta Andalām, Partha S. Roop, and Alain Girault. “Deterministic, predictable and light-weight multithreading using PRET-C”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE’10)*. Dresden, Germany, 2010, pp. 1653–1656.
- [ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. “Semantic translation of simulink/stateflow models to hybrid automata using graph transformations.” In: *Electronic Notes in Theoretical Computer Science*. 109 (2004), pp. 43–56.

- [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.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — principles, techniques, and tools*. Reading, Massachusetts: Addison-Wesley, 1986, p. 796. ISBN: 0-201-10088-6.
- [AT00] Jauhar Ali and Jiro Tanaka. “Converting statecharts into Java code”. In: *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*. Society for Design and Process Science (SDPS). Dallas, Texas, June 2000.
- [BC84] Gérard Berry and Laurent Cosserat. “The ESTEREL synchronous programming language and its mathematical semantics”. In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.
- [BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. “The synchronous languages twelve years later”. In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.
- [BCH+08] Darek Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. “Clock-directed modular code generation of synchronous data-flow languages”. In: *Proceedings of the ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*. Tucson, AZ, USA: ACM, June 2008, pp. 121–130.
- [Bee02] Michael von der Beeck. “A structured operational semantics for UML-statecharts”. In: *Software and Systems Modeling* 1.2 (Dec. 2002), pp. 130–141. ISSN: 1619-1366. DOI: 10.1007/s10270-002-0012-8.

Bibliography

- [Bee94] Michael von der Beeck. “A comparison of statecharts variants”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by H. Langmaack, W. P. de Roever, and J. Vytopil. Vol. 863. LNCS. Springer-Verlag, 1994, pp. 128–148.
- [Ber02] Gérard Berry. *The constructive semantics of pure Esterel*. Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France: Draft Book, Version 3.0, Dec. 2002.
- [Ber92] Gérard Berry. “Esterel on hardware”. In: *Philosophical Transactions of the Royal Society of London*. A 339 (1992), pp. 87–104.
- [BK08] C Baier and Joost P. Katoen. *Principles of model checking*. MIT Press, May 2008. ISBN: 978-0-262-02649-9.
- [BPS06] Reinhard Budde, Axel Poigné, and Karl-Heinz Sylla. “synERJY An object-oriented synchronous language”. In: *Electronic Notes in Theoretical Computer Science* 153.4 (2006), pp. 99–115.
- [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. *An FPGA-based demonstrator for dynamic ticks*. Technical Report 2001. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2020.
- [Bus16] Jonas Busse. “SCCharts modeling for embedded systems with limited resources”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbus-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.

- [BW96] Robert Büssow and Matthias Weber. “A steam-boiler control specification with statecharts and z”. In: *Formal methods for industrial applications*. Springer, 1996, pp. 109–128.
- [BWC+08] K. Belhajjame, K. Wolstencroft, O. Corcho, T. Oinn, F. Tanoh, A. William, and C. Goble. “Metadata management in the taverna workflow system”. In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. May 2008, pp. 651–656. DOI: 10.1109/CCGRID.2008.17.
- [CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. “Register allocation via coloring”. In: *Comput. Lang.* 6.1 (Jan. 1981), pp. 47–57. ISSN: 0096-0551. DOI: 10.1016/0096-0551(81)90048-5. URL: [http://dx.doi.org/10.1016/0096-0551\(81\)90048-5](http://dx.doi.org/10.1016/0096-0551(81)90048-5).
- [CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. “The nuXmv symbolic model checker”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 334–342.
- [CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. “NuSMV 2: An open-source tool for symbolic model checking”. In: *Computer Aided Verification*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364. ISBN: 978-3-540-45657-5.
- [CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490.

Bibliography

- [CGG10] Vasa Curcin, Moustafa Ghanem, and Yike Guo. “The design and implementation of a workflow analysis tool”. In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 368.1926 (2010), pp. 4193–4208. ISSN: 1364-503X. DOI: 10.1098/rsta.2010.0157. eprint: <http://rsta.royalsocietypublishing.org/content/368/1926/4193.full.pdf>. URL: <http://rsta.royalsocietypublishing.org/content/368/1926/4193>.
- [CPH+87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. “Lustre: a declarative language for programming synchronous systems”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’87)*. Munich, Germany: ACM, 1987, pp. 178–188.
- [CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “A conservative extension of synchronous data-flow with state machines”. In: *ACM International Conference on Embedded Software (EMSOFT’05)* (Jersey City, NJ, USA). Jersey City, NJ, USA: ACM, Sept. 2005, pp. 173–182.
- [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.
- [Dom18] Sören Domrös. “Moving model-driven engineering from Eclipse to web technologies”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, Nov. 2018.
- [Dor08] Francois-Xavier Dormoy. “Scade 6: a model based solution for safety critical software development”. In: *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS’08)*. 2008, pp. 1–9.

- [Dou99] Bruce Powel Douglass. “UML statecharts”. In: *Embedded Systems Programming* (Jan. 1999), pp. 22–42.
- [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.
- [Eum17] Philip Eumann. “A domain-specific language for railway control”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-bt.pdf>. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2017.
- [Eum20] Philip Eumann. “Runtime debugging of SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-mt.pdf>. Masters’s thesis. Kiel University, Department of Computer Science, Mar. 2020.
- [EZ07] Stephen A. Edwards and Jia Zeng. “Code generation in the Columbia Esterel Compiler”. In: *EURASIP Journal on Embedded Systems* Article ID 52651, 31 pages (2007).
- [FBH+16] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. “Time for reactive system modeling: interactive timing analysis with hotspot highlighting”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: ACM, 2016, pp. 289–298. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997467.
- [FBS+14a] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. “Towards interactive timing analysis for designing reactive systems”. In: *Reconciling Performance and Predictability (RePP ’14), satellite event of ETAPS ’14*. Apr. 2014.
- [FBS+14b] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. *Towards interactive timing analysis for designing reactive systems*. Tech. rep. UCB/EECS-2014-26. EECS Department, University of California, Berkeley, Apr. 2014. URL:

Bibliography

- <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-26.html>.
- [FH10] Hauke Fuhrmann and Reinhard von Hanxleden. “On the pragmatics of model-based design”. In: *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development, Revised Selected Papers*. Vol. 6028. LNCS. Budapest, Hungary: Springer, 2010, pp. 116–140. doi: 10.1007/978-3-642-12566-9.
- [Fli16] Niclas Fliieger. “Comparison of compilation approaches in KIELER”. Master thesis. Kiel University, Department of Computer Science, Apr. 2016.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349. issn: 0164-0925.
- [FSK+05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. “29 new unclarities in the semantics of UML 2.0 State Machines”. In: *ICFEM*. Vol. 3785. LNCS. Springer, 2005, pp. 52–65.
- [Fuh11] Hauke Fuhrmann. “On the pragmatics of graphical modeling”. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.
- [Fuh17] Insa Fuhrmann. “Time for reactive system modeling”. Kiel Computer Science Series 2018/2. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2017.
- [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.
- [GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

- [GHL+13] John C. Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li. “Generating domain-specific visual language tools from abstract visual specifications”. In: *IEEE Transactions on Software Engineering* 39.4 (Apr. 2013), pp. 487–515. issn: 0098-5589. doi: 10.1109/TSE.2012.33.
- [Gon88] Georges Gonthier. “Sémantiques et modèles d’exécution des langages réactifs synchrones : application à esterel”. PhD thesis. Université Paris, 1988.
- [GR83] S. Moisan G. Berry and J-P. Rigault. “Esterel: Towards a synchronous and semantically sound high-level language for real-time applications”. In: *IEEE Real-Time Systems Symposium*. IEEE Catalog 83CH1941-4. 1983, pp. 30–40.
- [Gri16] Lena Grimm. “Debugging SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf>. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2016.
- [Gri19] Lena Grimm. “From Lustre to graphical dataflow programs”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, May 2019.
- [Gro15] Object Management Group. *Specification of the unified modeling language (UML)*. <http://www.omg.org/spec/UML/2.5/>. May 2015.
- [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.
- [Ham05] Grégoire Hamon. “A denotational semantics for Stateflow”. In: *EMSOFT’05: Proceedings of the 5th ACM International Conference on Embedded Software*. Jersey City, NJ, USA: ACM Press, 2005, pp. 164–172. isbn: 1-59593-091-4.

Bibliography

- [Han09a] Reinhard von Hanxleden. *SyncCharts in C*. Technical Report 0910. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.
- [Han09b] Reinhard von Hanxleden. “SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency”. In: *Proc. Int’l Conference on Embedded Software (EMSOFT ’09)*. Grenoble, France: ACM, Oct. 2009, pp. 225–234.
- [Har09] David Harel. “Statecharts in the making: A personal account”. In: *Commun. ACM* 52.3 (Mar. 2009), pp. 67–75. ISSN: 0001-0782. DOI: 10.1145/1467247.1467274. URL: <http://doi.acm.org/10.1145/1467247.1467274>.
- [Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.
- [Hat95] Les Hatton. *Safer C: Developing software for in high-integrity and safety-critical systems*. McGraw-Hill, Inc., 1995.
- [HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.
- [HDM+13a] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *Compiling SCCharts to hardware and software*. Presentation at Synchronous Programming (SYNCHRON ’13), Schloss Dagstuhl, Germany. Nov. 2013.
- [HDM+13b] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts*. Presentation at Synchronous Programming (SYNCHRON ’13), Schloss Dagstuhl, Germany. Nov. 2013.

- [HDM+13c] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.
- [HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for safety-critical applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.
- [HFM+16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. “ThingML: A language and code generation framework for heterogeneous targets”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS ’16*. Saint-malo, France: ACM, 2016, pp. 125–135. ISBN: 978-1-4503-4321-3. DOI: 10.1145/2976767.2976812. URL: <http://doi.acm.org/10.1145/2976767.2976812>.
- [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*. Ed. by Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 325–354. ISBN: 978-3-540-27863-4. DOI: 10.1007/978-3-540-27863-4_19.
- [HLM+12] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. “Multi-view modeling and pragmatics in 2020 — position paper on designing complex cyber-

Bibliography

- physical systems”. In: *Proceedings of the 17th International Monterey Workshop 2012 on Development, Operation and Management of Large-Scale Complex IT Systems, Revised Selected Papers*. Vol. 7539. LNCS. Oxford, UK, 2012, pp. 209–223.
- [HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. “Statemate: A working environment for the development of complex reactive systems”. In: *IEEE Transactions on Software Engineering* 16.4 (Apr. 1990), pp. 403–414.
- [HMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O’Brien. “Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation”. In: *Proc. Design, Automation and Test in Europe Conference (DATE ’13)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.
- [HMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O’Brien, and Partha Roop. “Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation”. In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.
- [HMM+17] Reinhard von Hanxleden, Michael Mendler, Christian Motika, Christoph Daniel Schulze, and Steven Smyth. “SCCharts, KIELER and the Eclipse Layout Kernel—Statecharts for safety-critical applications and a pragmatics-aware modeling environment”. In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE ’17)*. Lausanne, Switzerland, Mar. 2017.
- [HMP04] Jozef Hooman, Nataliya Mulyar, and Ladislau Posta. “Coupling simulink and uml models”. In: *Proceedings of Symposium FORMS/FORMATS*. 2004, pp. 304–311.

- [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.
- [HPS+87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. “On the formal semantics of statecharts”. In: *Proc. 2nd IEEE Symp. on Logic in Computer Science*. MODELS ’16. Ithaca, NY, 1987, pp. 54–64.
- [HR04] Grégoire Hamon and John Rushby. “An operational semantics for Stateflow”. In: *Fundamental Approaches to Software Engineering (FASE)*. Ed. by M. Wermelinger and T. Margaria-Steffen. Vol. 2984. LNCS. Barcelona, Spain: Springer, Apr. 2004, pp. 229–243.
- [IM14] Javier L. Cánovas Izquierdo and Jesús G. Molina. “Extracting models from source code in software modernization”. In: *Software and Systems Modeling* 13.2 (Sept. 2014), pp. 713–734. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0270-z.
- [Jür02] Jan Jürjens. “A UML statecharts semantics with message-passing”. In: *Symposium of Applied Computing (SAC 2002), Madrid*. <http://atbroy8/tmp/Jur02e.ps>. 2002.
- [Kös10] Niclas Köser. *SyncCharts in C auf Multicore*. Diploma thesis. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nks-dt.pdf>. Oct. 2010.
- [Lat02] Chris Lattner. “LLVM: An infrastructure for multi-stage optimization”. See <http://llvm.cs.uiuc.edu>. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.
- [Lat06] Chris Lattner. *Introduction to the LLVM compiler infrastructure*. Invited talk at Itanium Conference and Expo, San Jose, California. Apr. 2006.
- [Lee06] Edward A. Lee. “The problem with threads”. In: *IEEE Computer* 39.5 (2006), pp. 33–42.

Bibliography

- [Lee09] Edward A. Lee. *Finite state machines and modal models in ptolemy ii*. Tech. rep. UCB/EECS-2009-151. EECS Department, University of California, Berkeley, Nov. 2009. URL: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html>.
- [Lee17] Edward A. Lee. *Plato and the nerd, a creative partnership of humans and technology*. MIT Press, 2017. ISBN: 978-0262036481.
- [Len16] Stephan Lenga. “Model-based compilation of legacy C programs”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sle-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.
- [LNW03] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. “Actor-oriented design of embedded hardware and software systems”. In: *Journal of Circuits, Systems, and Computers (JCSC)* 12.3 (2003), pp. 231–260. DOI: 10.1142/S0218126603000751.
- [Mac15] Felix Machaczek. “Collision avoidance of safety-critical real-time systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fma-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.
- [Mea55] G. H. Mealy. “A method for synthesizing sequential circuits”. In: *Bell System Technical Journal* 34 (Sept. 1955), pp. 1045–1079.
- [MHH12] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. *Synchronous Java: Light-weight, deterministic concurrency and preemption in Java*. Technical Report 1213. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Oct. 2012.
- [MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. “Programming deterministic reactive systems with Synchronous Java (invited paper)”. In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and*

- Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Paderborn, Germany, June 2013.
- [MIS13] MISRA. *Misra c:2012: Guidelines for the use of the C language in critical systems*. Motor Industry Research Association, 2013. ISBN: 9781906400101.
- [MLA10] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. 2010.
- [Mot09] Christian Motika. “Semantics and execution of domain specific models—KlePto and an execution framework”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf>. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.
- [Mot17] Christian Motika. *SCCharts—Language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017.
- [MS10] Taziana Margaria and Bernhard Steffen. “Simplicity as a driver for agile innovation”. In: *Computer*. IEEE Computer Society, 2010.
- [MS15] Christian Motika and Steven Smyth. *Updates on SCCharts*. Presentation at the 22th International Open Workshop on Synchronous Programming (SYNCHRON ’15), Kiel, Germany. Dec. 2015.
- [MS18] Tiziana Margaria and Bernhard Steffen. *Leveraging applications of formal methods, verification and validation. modeling*. Springer International Publishing, Nov. 2018. ISBN: 978-3-030-03418-4.
- [MSH+13] Christian Motika, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. *Sequentially Constructive Charts (SC-Charts)*. Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF ’13), Berkeley, CA, USA. Nov. 2013.

Bibliography

- [MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. “Compiling SCCharts—A case-study on interactive model-based compilation”. In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 461–480. doi: 10.1007/978-3-662-45234-9.
- [MSS+16] Christian Motika, Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *KIELER SCCharts tutorial*. Interactive Tutorial performed at the 23th International Open Workshop on Synchronous Programming (SYNCHRON ’16), Bamberg, Germany. Dec. 2016.
- [Nas15] Stanislaw Nasin. “Transformaion from SCCharts to Esterel”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sna-mt.pdf>. Master Thesis. Kiel University, Department of Computer Science, Oct. 2015.
- [NLK+18] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. “CINCO: A simplicity-driven approach to full generation of domain-specific graphical modeling tools”. In: *International Journal on Software Tools for Technology Transfer* 20.3 (June 2018), pp. 327–354. ISSN: 1433-2787. doi: 10.1007/s10009-017-0453-6.
- [OAF+04] Tom Oinn et al. “Taverna: A tool for the composition and enactment of bioinformatics workflows”. In: *Bioinformatics* 20.17 (2004), pp. 3045–3054. doi: 10.1093/bioinformatics/bth361. eprint: /oup / backfile / content_public / journal / bioinformatics / 20 / 17 / 10.1093/bioinformatics/bth361 / 2 / bth361.pdf. URL: <http://dx.doi.org/10.1093/bioinformatics/bth361>.
- [Ols16] Lars Olsson. “Modellextraktion aus C code”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/leo-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2016.

- [PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.
- [Pei15] Lars Peiler. “Modeling simulations of autonomous, safety-critical systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.
- [Pei17] Lars Peiler. “Priority-based compilation of SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-mt.pdf>. Master thesis. Kiel University, Department of Computer Science, Oct. 2017.
- [Pet62] Carl Adam Petri. “Kommunkation mit Automaten”. MA thesis. Bonn: Mathematisches Institut, Universität Bonn, 1962.
- [PH95] A. Poigné and L. Holenderski. *Boolean automata for implementing pure Esterel*. Arbeitspapiere 964. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 1995.
- [Plo81] Gordon D. Plotkin. *A structural approach to operational semantics*. Technical Report DAIMI FN-19. <http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps>. University of Aarhus, Denmark, 1981.
- [PM03] Gergely Pintér and István Majzik. “Program code generation based on UML statechart models”. In: *Periodica Polytechnica* (2003), pp. 187–204.
- [Pot02] Dumitru Potop-Butucaru. “Optimizations for faster simulation of Esterel programs”. PhD thesis. Ecole des Mines de Paris, France, Nov. 2002.
- [PR09] Marc Pouzet and Pascal Raymond. “Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation”. In: *EMSOFT*. 2009, pp. 215–224.
- [PRS+16a] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. *Runtime enforcement of reactive systems using synchronous enforcers*. Presentation

Bibliography

- performed at the 23th International Open Workshop on Synchronous Programming (SYNCHRON '16), Bamberg, Germany. Dec. 2016.
- [PRS+16b] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. “Runtime enforcement of reactive systems using synchronous enforcers”. In: *ArXiv e-prints* (Dec. 2016). arXiv: 1612.05030 [cs.FL]. URL: <http://adsabs.harvard.edu/abs/2016arXiv161205030P>.
- [PRS+17a] Srinivas Pinisetty, Partha Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. “Runtime enforcement of reactive systems using synchronous enforcers”. In: *Proc. International SPIN Symposium on Model Checking of Software (SPIN '17)*. Santa Barbara, CA, USA, 713–14 2017.
- [PRS+17b] 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.
- [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, June 2006.
- [Rab63] Michael O. Rabin. “Probabilistic automata”. In: *Information and Control* 6.3 (1963), pp. 230–245. issn: 0019-9958. doi: [https://doi.org/10.1016/S0019-9958\(63\)90290-0](https://doi.org/10.1016/S0019-9958(63)90290-0). URL: <http://www.sciencedirect.com/science/article/pii/S0019995863902900>.
- [Rah17] Milad Rahimi-Barfeh. “Incremental compilation of SCEst”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mrb-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2017.

- [Rat15] Karsten Rathlev. “From Esterel to SCL”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/kratmt.pdf>. Master thesis. Kiel University, Department of Computer Science, Mar. 2015.
- [Ren18] Niklas Rentz. “Moving transient views from Eclipse to web technologies”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, Nov. 2018.
- [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.
- [RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. Vol. 9953. LNCS. Corfu, Greece, Oct. 2016, pp. 150–170. doi: 10.1007/978-3-662-45234-9.
- [Rüe11] Ulf Rüegg. “Interactive transformations for visual models”. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2011.
- [Rüe18] Ulf Rüegg. *Sugiyama layouts for prescribed drawing areas*. Kiel Computer Science Series 2018/1. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2018.
- [Sam02] Miro Samek. *Practical statecharts in C/C++*. CMP Books, 2002.
- [SBP+08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse modeling framework, second edition*. 2008.

Bibliography

- [Sch09] Klaus Schneider. *The synchronous programming language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.
- [Sch14] Alexander Schulz-Rosengarten. “Framework zum Tracing von EMF-Modelltransformationen”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2014.
- [Sch16] Alexander Schulz-Rosengarten. “Strict sequential constructiveness”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf>. Master thesis. Kiel University, Department of Computer Science, Sept. 2016.
- [Sch19] Christoph-Daniel Schulze. *Text in diagrams*. Kiel Computer Science Series 2019/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2019. ISBN: 9783734772689.
- [SDH19] Steven Smyth, Sören Domrös, and Reinhard von Hanxleden. *A case-study on manual verification of state-based source code generated by KIELER SCCharts*. Technical Report 1905. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.
- [SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling*. John Wiley & Sons, 1994. ISBN: 0-471-59917-4.
- [SHM+18a] 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.
- [SHM+18b] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. *Time in SCCharts*. Technical Report 1805. ISSN 2192-6247.

Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018.

- [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 Trans. Embedded Comput. Syst.* 16.4 (July 2017), 98:1–98:26. doi: 10.1145/3035544.
- [SLH16] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. “Model extraction for legacy C programs with SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA ’16), Doctoral Symposium*. Vol. 74. Electronic Communications of the EASST. With accompanying poster. Corfu, Greece, Oct. 2016. doi: 10.14279/tuj.eceasst.74.1044.
- [SMB97] Bernhard Steffen, Tiziana Margaria, and Volker Braun. “The Electronic Tool Integration platform: concepts and design”. In: *International Journal on Software Tools for Technology Transfer* 1.1 (Dec. 1997), pp. 9–30. issn: 1433-2779. doi: 10.1007/s100090050003. url: <https://doi.org/10.1007/s100090050003>.
- [SMH15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “A data-flow approach for compiling the sequentially constructive language (SCL)”. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörtlach, Austria, Oct. 2015.
- [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.

Bibliography

- [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 (TECS)—Special Issue on MEMOCODE 2015* 17.2 (Dec. 2017), 33:1–33:26. issn: 1539-9087.
- [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. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.
- [SMS+19a] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. *SCCharts: the mindstorms report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.
- [SMS+19b] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. *SCCharts: the mindstorms report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.
- [Smy13] Steven Smyth. “Code generation for sequential constructiveness”. <https://rtds.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf>. Diploma thesis. Kiel University, Department of Computer Science, July 2013.
- [SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing legacy systems: Software technologies, engineering processes, and business practices*. Boston, Massachusetts, USA: Addison Wesley, 2003. isbn: 0321118847.

- [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. Christian-Albrechts-Universität zu Kiel, 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: *2018 18th International Conference on Application of Concurrency to System Design (ACSD)*. 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.
- [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.
- [SSH18b] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Practical causality handling for synchronous languages*. Technical Report 1808. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, 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*

Bibliography

- (*ISoLA 2018*). Vol. 11244. LNCS. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260.
- [SSH18d] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Watch your compiler work — Compiler models and environments*. Technical Report 1806. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018.
- [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.
- [SSM+19] Steven Smyth, Alexander Schulz-Rosengarten, Christian Motika, and Reinhard von Hanxleden. “The KIELER SCCharts Editor—A modular open-source modeling suite with automatic diagram synthesis”. In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE ’19)*. Florence, Italy, Mar. 2019.
- [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.
- [SSS+19] Monty Santarossa, Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Using SCCharts models in Simulink to model an electric control unit*. Technical Report 1903. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2019.
- [Sta15] Andreas Stange. “Comfortable SCCharts modeling for embedded systems”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-bt.pdf>. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2015.

- [Sta19] Andreas Stange. “Model checking for SCCharts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf>. Master’s thesis. Kiel University, Department of Computer Science, May 2019.
- [Ste97] Bernhard Steffen. “Unifying models”. In: *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany*. Mar. 1997, pp. 1–20. DOI: 10.1007/BFb0023444. URL: <http://dx.doi.org/10.1007/BFb0023444>.
- [Sti92] Colin Stirling. “Handbook of logic in computer science (vol. 2)”. In: ed. by S. Abramsky, Dov M. Gabbay, and S. E. Maibaum. New York, NY, USA: Oxford University Press, Inc., 1992. Chap. Modal and Temporal Logics, pp. 477–563. ISBN: 0-19-853761-1. URL: <http://dl.acm.org/citation.cfm?id=162552.162565>.
- [SW01] K. Schneider and M. Wenz. “A new method for compiling schizophrenic synchronous programs”. In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’01)*. ACM. Atlanta, Georgia, USA, Nov. 2001, pp. 49–58.
- [Tar72] Robert E. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM Journal of Computing* 1.2 (1972), pp. 146–160.
- [TS03] Olivier Tardieu and Robert de Simone. “Instantaneous termination in pure Esterel”. In: *Static Analysis Symposium*. San Diego, California, June 2003.
- [TS04] Olivier Tardieu and Robert de Simone. “Curing schizophrenia by program rewriting in Esterel”. In: *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE’04)*. San Diego, CA, USA, 2004.

Bibliography

- [Uml15] Axel Umland. “Konzept zur Erweiterung von SCCharts um Datenfluss”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf>. Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015.
- [Wei15] Tibor Weiß. “Von Nebenläufigkeit zur Parallelität in SC-Charts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/twe-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Oct. 2015.
- [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. LNCS 11200. Springer International Publishing, 2018, pp. 504–523.
- [WZ91] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Transactions on Programming Languages and Systems* 13.2 (Apr. 1991), pp. 181–210.
- [YGR+16] Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. “The ForeC synchronous deterministic parallel programming language for multicores”. In: *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2016, Lyon, France, September 21-23, 2016*. 2016, pp. 297–304.
- [Zwe18] Philip Zweihoff. “Towards domain-specific graphical language server protocols”. In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA ’18), Doctoral Symposium*. Vol. 74. Electronic Communications of the EASST. Limassol, Cyprus, Nov. 2018.

List of Figures

1.0.1	Model-driven software development example with interactive compilation from the KIELER SCCharts project	5
1.1.1	Outline of the development ecosystem	11
2.1.1	Reactions in synchronous languages	14
2.1.2	Syntax and behaviour of the <i>Hello World!</i> of sequential constructive statecharts: ABO	17
2.1.3	The Extended and Core SCCharts variants of AO with signals	20
2.1.4	SCCharts feature overview	21
2.2.1	KIELER SCCharts Editor – Simulation perspective	24
2.2.2	The KIELER SCCharts development timeline	25
2.3.1	Previous work done regarding the compilation of SCCharts by Motika	28
2.3.2	Previous work done regarding interactive timing analysis by Fuhrmann	31
2.3.3	LLVM Architecture	32
2.3.4	The UNIX coordination universe	34
2.3.5	CINCO generates ready-to-run modelling tools from abstract tool specification.	35
2.3.6	Petrinet modelling tool automatically generated by CINCO .	36
2.3.7	Developer role layer from CINCO’s point of view	37
2.3.8	Web-based CINCO product development	38
2.3.9	Part of the Citizen Quartz clock example	39
2.3.10	Conflicting Transitions in Rhapsody and STATEMATE	41
2.3.11	The “Hello World” of synchronous languages – ABRO	44
2.3.12	Combining statecharts with Esterel: SyncCharts	45
2.3.13	SCADE Suite user interface showing ABRO	48
2.3.14	Different design patterns for Statechart code generation . .	50
2.3.15	Different code generation approach for Esterel	52

List of Figures

2.3.16 SyncChart's meta model	53
2.3.17 SyncCharts to SC translation	53
3.1.1 Three alternative development processes	60
3.1.2 Different model layer and user roles of interactive compilation systems	64
3.2.1 Different parts of a model-based compilation	65
3.2.2 Compilation workflow example – ABRO for Arduino	67
3.2.3 Different views of the same program, AO: C, SCT, SCCharts, SCG	69
3.2.4 Model layer of the AO examples in Figure 3.2.3	70
3.2.5 User roles' focus w.r.t. the chapters of this book	71
3.2.6 Different role's interactions with the framework	72
3.3.1 Processors and processor instances	74
3.3.2 Compilation systems and compilation contexts	76
3.3.3 Concept of a compilation context with two processors.	78
3.3.4 To save resources, several processors can be grouped together. Generally, everything which happens between two environments is commonly called a transformation.	78
3.3.5 Schema of a co-processor relationship	79
3.3.6 Example of an automatically generated graphical view of a compilation system	83
3.3.7 Complete example of a running KIELER instance during simulation.	84
3.3.8 When joining different branches, model measures rate the quality of the preceding results to determine the new source environment.	85
3.3.9 A processor can access the compilation context during runtime and change the subsequent compilation chain.	86
3.3.10 Illustration of a dynamically extended compilation context	87
3.3.11 Immediate join of branched processors in KIELER	88
3.3.12 Complete system with LoC measurement of branches with different internal optimizations determine the quality of the optimizations by measuring the LoC of the generated assembler code.	89

3.3.13	LoC measurement of branches with different external optimizations in a compilation system	90
3.3.14	Instantaneous visual feedback of the quality of different compilation branches	91
3.4.1	SLIC dependency table	92
3.4.2	Generated SLIC schedule re-using existing processor modules	93
4.1.1	Schema of the Interactive Model-Understanding-Refinement-Documentation feedback loop (iMURD)	98
4.1.2	Interactive models in a KiCo compilation system	99
4.1.3	Example of different unmodified transformation snapshots: LeanStrongAbort	100
4.1.4	Data dependencies	102
4.1.5	Causal dataflow	103
4.1.6	Scheduling propagation	104
4.1.7	Transformation snapshots	105
4.1.8	Automatic element tracing	106
4.1.9	Automatic element tracing of selected elements	106
4.1.10	Built-in code mapping	107
4.2.1	KiCos compilation system universe	108
4.2.2	Top-level view on the KIELER systems universe	109
4.2.3	Example of the KiCo Registry	110
4.3.1	The SCCharts base model for complexity approximations .	111
4.3.2	Complexity ratio for expressions and valued objects	112
5.1.1	Surface and depth w.r.t. a pause statement	119
5.1.2	Matrix showing the entire mapping throughout the transformation process from SCCharts to circuits	121
5.1.3	The ABO program in two different meta-models. Note that both views use the same OSM. They show two different points of time during the same compilation.	123
5.2.1	Core processors of the netlist-based compilation	125
5.2.2	Concurrent accesses share an LCAF	128
5.2.3	Write-Read and write-write dependencies	128
5.2.4	Dependencies shown in the SCG of ABO	130

List of Figures

5.2.5	Further dependency views on the SCG of ABO	131
5.2.6	Visualization of a detected loop within the compilation chain	132
5.2.7	Two different examples of basic blocks	134
5.2.8	Example of two SBs dividing a BB	135
5.2.9	SB partitioning of ABO	137
5.2.10	Two different examples of dead BBs	139
5.2.11	Two different examples of BB guard expressions	140
5.2.12	SB partitioning of ABO with guard expressions	143
5.2.13	Different synchronizers	145
5.2.14	Schizophrenia in SCCharts	147
5.2.15	Curing schizophrenia via SDJ	148
5.2.16	SCPDG excerpt with different kinds of dependencies	151
5.2.17	Two examples of the SCPDG	152
5.2.18	The sustain example	153
5.2.19	Scheduling of the ABO netlist	155
5.2.20	Illustration of the SC+ semantics	157
5.2.21	Sequentialized version of ABO	158
5.2.22	While structural reasonable, forks with just one outgoing control-flow are semantically superfluous for the downstream compilation.	161
5.2.23	While semantically valid on model-level, threads with no behaviour are superfluous for the downstream compilation.	162
5.2.24	Example result of the copy propagation processor which eliminates copy statements	163
5.2.25	The Conditional Merger merges control-flows which are executed on the same condition.	164
5.2.26	The halt state optimization removes halt states, which are ineffective.	166
5.2.27	Register relocation: Used guard variables can be recycled after their life span.	167
5.2.28	Persistent state optimization, replaces stateful register patterns with boolean true expressions.	169
5.2.29	Partial Assignment Evaluation scans the SCG sequentially and applies configurations to variables.	170
5.2.30	_GO MoC optimization example	171

5.2.31	Second <code>_GO</code> MoC optimization example	172
5.2.32	<code>!_GO</code> MoC optimization example	173
5.2.33	Sequentialized optimized SCG of ABO	174
5.2.34	Serialized C code of ABO in the netlist-based approach as it interactively appears in the KiCo compilation chain	176
5.2.35	Serialized Java code of the sequentialized SCG of ABO	178
5.2.36	Sequentialized optimized SSA SCG of ABO	179
5.2.37	Synthesized circuit of ABO	180
5.2.38	Synthesized circuit of ABO during simulation	181
5.3.1	SC Classes and programs	183
5.3.2	The SCG of ABO annotated with priorities, thread segment IDs and prioIDs	185
5.3.3	ABO variant compiled with the priority-based compilation	188
5.3.4	C code of ABO generated by the priority-based compilation approach	189
5.3.5	Java code of ABO generated by the priority-based compila- tion approach	192
5.4.1	Requirements Overview	198
5.4.2	Requirements checklist for KIELER	198
5.4.3	Topology-preserving code-generation: The state-based ap- proach	200
5.4.4	Thread status in priority-based concurrency	202
5.4.5	Generated code of the topology-preserving code generation	204
5.4.6	Lean state-based compilation system	206
5.4.7	State-based example SCChart with region dependencies	206
5.4.8	Superstate code of the lean state-based approach	207
5.4.9	Leaving superstates: AbortSAWA	208
5.4.10	Lean CS State-based abort serialization	209
5.4.11	Entry action code for superstates in Lean CS	210
5.5.1	Initial benchmarks of the netlist-based approach	211
5.5.2	Benchmarks of the initial SCCharts contribution	211
5.5.3	Comparison of code sizes according to Peiler	212
5.5.4	Comparison of execution times according to Peiler	213
5.5.5	Jitter comparison according to Peiler	214
5.5.6	The Disk-and-Sticks demonstrator	215

List of Figures

5.5.7	Evaluation of the Disk-and-Sticks demonstrator	216
5.5.8	Mean time and confidence of the state-based compilation trials with vs. without comments regardless of trial ordering	219
5.5.9	Mean time and confidence of all trials.	220
5.5.10	Comment's influence rating (light: SB with comments first; dark: SB without comments first)	222
5.5.11	Mean tick execution time (μs)	222
5.5.12	Steam boiler model in SCCharts	224
5.5.13	Inner behaviour of the degraded state	227
5.5.14	Steam boiler case-study results	228
5.5.15	Schematic of the test suite benchmark compilation system .	231
5.5.16	Comparison between all presented compilation approaches	234
5.5.17	Comparison of individual tick times	235
6.1.1	Dependencies induced by either a MoC or an SD	239
6.1.2	Concurrent Counter Reset program in SCCharts	241
6.1.3	Data dependencies are visualized as coloured, dashed edges between the regions.	242
6.1.4	Counter Reset example with SDs	243
6.1.5	Example WW5 depicts a strict order across three regions. . .	244
6.1.6	Cyclic count delay	244
6.1.7	Expanded cyclic count delay	245
6.1.8	Cured expanded Cyclic Count delay	246
6.2.1	Sequentially constructive write operators in the SCCharts dataflow extension	249
6.2.2	Sequentially constructive dataflow example	250
6.2.3	Explicit sequential constrains in dataflow SCCharts	252
6.2.4	Dataflow SCCharts limitations imposed by circular dependencies	252
6.2.5	Transformation of the example from Figure 6.2.2 to control-flow SCCharts	253
6.2.6	Transformation of the counter example from Figure 6.2.2 .	254
6.2.7	Immediate ABRO as dataflow SCCharts	256
6.2.8	Delayed ABRO as dataflow SCCharts	257
6.2.9	Inducing a schedule for the init operation	258

6.2.10	Mixing dataflow-based and control-flow-based modelling styles	260
6.2.11	Simulation of DFAbro from Figure 6.2.10	260
6.2.12	Dataflow once transformation of the Counter example from Listing 6.2.1	262
6.2.13	Different action views of the same SCCharts model	263
6.2.14	Counting node example in Lustre, SCADE and four different views synthesized by KIELER	264
6.2.15	Using the netlist-based code generation approach to compile Lustre programs in KIELER	266
6.3.1	Excerpt from the extended SCCharts compilation system with additional actions	268
6.3.2	Step-wise transformation of the WeakEntry example	269
6.4.1	The Control example, illustrating the sequential modification of shared variables.	271
6.5.1	Full Fibonacci example	275
6.5.4	Clocked computation of the Fibonacci number	276
6.5.2	Generated netlist of the extracted SCChart depicted in Figure 6.5.1b	277
6.5.3	Possible out-of-the-box compilation targets for the Fibonacci example in Figure 6.5.1 in the KIELER SCCharts tools	278
6.6.1	Language preferences	282
6.6.2	Deterministic concurrency	283
6.6.3	Sequentiality	284
6.6.4	Separate timing & functionality	284
6.6.5	Solving abstract problems	285
6.6.6	Solving low-level problems	286
6.6.7	Understandability	287
6.6.8	Simplicity	287
6.6.9	Composability	288
6.6.10	Maintainability	289
6.6.11	Debugging	289
6.6.12	Essential transition features	291
6.6.13	History and local action features	293
6.6.14	Concurrency, declarations, and data types	294

List of Figures

6.6.15	Additional extended features	295
6.6.16	Future features	297
6.6.17	Tool quality	299
6.6.18	Model creation & debugging	300
6.6.19	Tooling aspects	300
7.1.1	Automatically generated SCChart for the high-level specification shown in Table 7.1.1	305
7.1.2	Illustration of the LTL semantics	307
7.1.3	NuSMV compilation system in KIELER	307
7.1.4	Model Checking for SCCharts	309
7.1.5	Diagram of a panel member cycle for DIN EN 13725 limit y/n311	
7.1.6	Corrected behaviour of the DIN EN 13725 panel member cycle in a y/n odour determination measurement according to the done model check	312
7.2.1	Runtime enforcement in a pacemaker device	313
7.2.2	Example safety automaton of the P_2 property in SCCharts	314
7.3.1	An assembled NXT	316
7.3.2	Barcode encoding	317
7.3.3	Barcode reader simulation visualization	318
7.3.4	An example for a barcode reader model	319
7.3.5	The pathfinder mat	320
7.3.6	The first SCCharts railway project	322
7.3.7	SCChart generated from a looped RailSL block	323
7.3.8	RailSL IDE with reasonable modeller feedback	324
7.3.9	Quadrocopter student project 2015	325
7.3.10	Using SCCharts Models in Simulink to Model an Electronic Control Unit	327
8.2.1	Meta tool engines moving into the web.	332
8.2.2	The semantic gap	333
8.2.3	SCCharts graph rewriting example	334
8.2.4	KIELER user interface with components relevant to debugging highlighted	335
8.2.5	inheritance in SCCharts	336

8.2.6	Deterministic usage of host language object Counter using an SD	337
8.2.7	Traditional modelling flow vs. modelling pragmatics	338
B.1.1	Example of an UML statechart modelled in Rhapsody	360
B.2.1	MathWorks Simulink examples	361
B.2.2	Stopwatch example in Stateflow	362
B.3.1	Hierarchical Ptolemy II example with different directors	363
B.4.1	ThingML framework extension points and compilers LoC distribution.	364
B.5.1	Taverna Workbench	367
C.0.1	Guidance in Model-based Compilation Poster, demonstrating the iMURD cycle in SCCharts	372
D.1.1	Clock example: Delay SCChart	374
D.1.2	Clock example: Emitter SCChart	375
D.1.3	Clock example: clock	376
D.1.4	Clock example: fully expanded version of clock	377
D.1.5	Editor binding warnings	377
D.1.6	Clock example: module call version of clock	379
D.2.1	Modelled and effective probabilities in probabilistic priority transitions	380
D.2.2	PrPT example: prEnv	382
D.2.3	Expanded prEnv model from Figure D.2.2a	383
D.3.1	The <i>for region</i> feature in Extended SCCharts	384

List of Tables

5.2.1	KIELER netlist-based optimizations and their code sizes . . .	159
5.5.1	Lines of code for the different trials: netlist-based, priority-based, state-based without comments and state-based with comments	218
5.5.2	Order of code generation approaches for each group	226
6.1.1	Results of the different count delay approaches in SCCharts when compiling the Cyclic Count Delay model	246
6.2.1	Selected results of the benchmarks comparing the sizes and reaction times of the SCCharts dataflow extension and Verimag Lustre V6	267
7.1.1	Excerpt of a high-level specification for a user interface written in a common spreadsheet software	304
7.1.2	Behaviour of an EN 13725 panelist cycle in a y/n odour determination measurement	310

List of Listings

1.0.1	Classical results of different compilation optimizations . . .	2
1.0.2	Illustration of intermediate compilation result in the GCC .	3
2.1.1	Typical sequential programming pattern forbidden in classical synchronous languages: set flag x to true after some work has been done.	15
2.3.1	Possible ABRO implementation in Lustre	48
3.3.1	Compilation invocation excerpt from the KIELER project . .	77
3.3.2	Model description of the netlist-based SCCharts compilation	82
3.3.3	Nested statements in incremental SCEst compilations . . .	86
3.4.1	SLIC dependency excerpt in SCL	92
3.4.2	SLIC Schedule system	93
4.1.1	A small C program	96
4.1.2	Code snippet of the SSA intermediate representation in GCC generated with the <code>-fdump-tree-ssa</code> option.	96
5.2.1	Textual description of the netlist-based compilation system in KiCo	127
5.2.2	Joined Depth-First Search (JDFS) algorithm	138
5.2.3	Simple guard expression algorithm	142
5.2.4	Sequentialization algorithm	158
5.2.5	Assembler code of ABO with different compiler optimizations	177
5.3.1	Longest weighted path algorithm	186
5.3.2	Tick function synthesized for ABO with expanded SCL_p macros	190
5.5.1	Complete benchmark configuration for the final run-time benchmarks	232

List of Listings

6.2.1	Dataflow extension syntax	248
6.4.1	ReEmit is not valid in Esterel due to the emission of A after its value is read with $?A$; it is valid in SCEst.	270
6.4.2	The Control example in SCEst with variables, and with signals, including transformation to sCL.	273
D.1.1	Explicitly bound local scope variables to the interface of the referenced SCChart	378
D.1.2	Local scope variables bound by order to the interface of the referenced SCChart	378
D.1.3	Variables with the same name may be bound implicitly if not explicitly stated	378

List of Acronyms

ACM	Association for Computing Machinery An international scientific society for computing.
ASC	Acyclic-Schedulable Constructive A program class of SC, which allows structural cycles without reachable data dependencies.
AST	Abstract Syntax Tree A program representation.
BB	Basic Block A standard concept of program structuring, where statements that are executed together form a logical unit.
BDD	Binary Decision Diagram A data structure which represents boolean functions.
BFS	Breadth-First Search A graph traversal algorithm which explores all of the neighbouring nodes before moving on to the next depth level.
CASE	Computer-Aided Software Engineering Software tools used to design and implement applications.
CDT	C Development Tooling C development tooling embedded in Eclipse.
CEC	Columbia Esterel Compiler An Esterel compiler developed at the Columbia University.
CFR	Control-flow Region A region that comprises a set of states, which are connected via transitions. Only one state is active at a time. Conceptionally, a CFR resembles a single thread.
COTS	Commercial Off-The-Shelf Commercial, common, packaged hardware or software solutions.
CPS	Cyber-Physical System A modern term for embedded systems underlining the fact that the overall system must also include the physical aspects of the environment.

List of Acronyms

CPU	Central Processing Unit The processing core of a modern computer.
LCS	Lean Common Set A language subset, which comprises language elements/features that constitute a trade-off between modelling convenience and compilation complexity.
DFR	Dataflow Region A region that comprises a list of assignments, which are evaluated in every tick when the enclosing state is active. Conceptionally, each assignment in a DFR resembles an immediate during action.
DFS	Depth-First Search A graph traversal algorithm which explores as far as possible along each branch before backtracking to another branch.
DSL	Domain-Specific Language A (usually small) language developed for a specific purpose, as opposed to general-purpose languages.
DTA	Discrete Timed Automata A timed automata used for expressing safety properties in safety-critical CPS
ELK	Eclipse Layout Kernel An Eclipse project that provides an infrastructure for layout algorithms.
EMF	Eclipse Modeling Framework A modeling framework for generating code for a class model based on a model specification. Available at: http://www.eclipse.org/emf
ECU	Electric Control Unit Central part of an engine control unit for electric cars.
ETI	Electronic Tool Integration An online service for experimentation and coordination of tool functionalities. Available at: http://eti.cs.uni-dortmund.de
FS	Flexible Schedule A flexible schedule uses Scheduling Directives to augment the scheduling rules of the underlying Model of Computation
FSM	Finite State Machine A mathematical model of computation with a finite number of states.
FPGA	Field Programmable Gate Array Integrated circuit for configurable gates.

GCC	GNU Compiler Collection The name of the GNU compiler suite.
GLSP	Graphical Language Server Protocol A variant of the LSP for graphical languages.
GNU	GNU is not Unix A unix-like, free operating system.
GRC	Graph Code A control-flow-oriented intermediate program representation.
IASC	IUR-Acyclic-Schedulable Constructive A program class of SC, which allows cycles without reachable data dependencies.
IDE	Integrated Development Environment An editor that provides advanced features for developing software.
iMURD	Interactive Model-Understanding-Refinement-Documentation feedback loop An interactive way of modeller guidance and transformation observation, which helps to refine the source model.
IR	Intermediate Representation Program representations that are often used in compilers as common basis.
ITA	Interactive Timing Analysis A pragmatic modeling approach to aid the modeler at timing constraints.
IURP	Initialize-Update-Read Protocol A scheduling protocol that describes the order in which different types of variables accesses may proceed.
JDFS	Joined Depth-First Search A graph traversal algorithm which explores as far as possible along each branch before backtracking to another branch that does not proceed from SCG exit nodes right away. Instead, the join node will continue after all child threads have been searched.
JSON	JavaScript Object Notation A common and compact open standard for data interchange.
KiCo	KIELER Compiler A framework that handles arbitrary model transformations.
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client An environment for experimenting with pragmatic modelling concepts.

List of Acronyms

LCAF	Least Common Ancestor Fork The least common ancestor fork node of two potentially concurrent child nodes.
LLVM IR	LLVM Intermediate Representation The intermediate program representation of the LLVM compiler infrastructure.
LoC	Lines of Code A measure for source code.
LSP	Language Server Protocol A unified protocol to support development in programming languages.
LTL	Linear Temporal Logic A formal logic for model verification.
MC	Model Checking Automatic verification if a model meets a given specification.
MoC	Model of Computation Given an input, a model of computation describes how an output of a function is computed.
M2MC	Model-to-Model Compilation A chain of M2MT that form a compilation. Also see M2MT.
M2MT	Model-to-Model Transformation A transformation that is performed on a given model and returns another model.
M2TT	Model-to-Text Transformation A transformation that is performed on a given model and returns text.
MDE	Model-driven Engineering A software development methodology that exploits domain models to increase the productivity of developers.
OOP	Object-Oriented Programming A programming paradigm that centers around the creation of object, often instantiated from classes.
OS	Operating System A software system, which manages resources and provides common services for applications.
OSM	Original Source Model The source model that comes first in an M2MT.
PA	Probabilistic Automaton Generalization of nondeterministic finite automaton including probabilities.

PDG	Program Dependency Graph A program representation that facilitates maximum parallelism.
PID	Proportional–Integral–Derivative Short form of a PID controller, often used in control systems.
PrPT	Probabilistic Priority Transition A transition that is only fired with a certain probability if its trigger evaluates to true. Multiple transitions are checked according to their priority.
ROOM	Real-Time Object-oriented Modeling A methodology for the analysis and design of real-time systems.
RTW	Real-Time Workshop An integrated code synthesis for Matlab/Simulink.
SASC	Structurally Acyclic-Schedulable Constructive A program class of SC, which disallows cycles in the control-flow representation of a program.
SB	Scheduling Block A unit of statements within an Basic Block (BB) which combines statements that are scheduled together.
SLIC	Single-Pass Language-driven Incremental Compilation A model-based incremental compilation approach.
SC	Sequentially Constructive A synchronous program class that lifts the restriction of unique signal values in the sequential context.
SCChart	Sequentially Constructive Statechart A statechart dialect designed to provide deterministic concurrency with less restrictions than previous synchronous languages.
SCC	Strongly Connected Component A directed subgraph in which every node can reach every other node.
SCCP	Sparse Conditional Constant Propagation A compiler optimization for propagating constants and eliminating dead code.
SCEst	Sequentially Constructive Esterel An sequentially constructive Esterel dialect.
SCG	Sequentially Constructive Graph A control-flow representation of synchronous programs.

List of Acronyms

SCL	Sequentially Constructive Language A minimal, imperative language that captures the SCMoC. SCL can be seen as the kernel language of SCCharts.
SCMoC	Sequentially Constructive Model of Computation The computation model that is used by sequentially constructive programs.
SCPDG	Sequentially Constructive Program Dependency Graph A sequentially constructive program representation that facilitates maximum parallelism.
SCT	Textual SCCharts Language A textual language for defining SCCharts.
SD	Scheduling Directive A (modelled) directive that influences the scheduling of a given program.
SDJ	Structural Depth Join A method to duplicate surface instructions in synchronous models to cure schizophrenia.
SIASC	Structurally IUR-Acyclic-Schedulable Constructive A program class of SC, which allows structural cycles as long as they do not comprise any data dependencies.
SJ	Synchronous Java A class-based Java extension for synchronous models developed by Motika et al. [MHH13]
SKP	SCCharts Kernel Pattern Every SCChart can be expressed by an semantically equivalent SCChart that only uses the normalized pattern.
SOS	Structured Operational Semantics A semantics following strict structural replacement rules explored by Plotkin [Pl081].
SSA	Static Single Assignment A code representation in which every variable is assigned exactly once.
SVG	Scalable Vector Graphic A standardized graphics format for scalable images.
UI	User Interface The panel or console the user interacts with.

USB	Universal Serial Bus An industry norm for cables, connectors and communication protocols.
UML	Unified Modeling Language A collection of graphical languages for visualizing the design and the behaviour of (software) systems.
VDSL	Visual Domain-Specific Language A DSL distinguished by its visual representation.
VHDL	Very High Speed Integrated Circuit Hardware Description Language Hardware description language for describing digital and mixed-signal systems.
WCET	Worst-Case Execution Time The worst amount of time a tick instance cycle need under all circumstances.
XML	Extensible Markup Language A format for representing documents in a structured and well-defined way readable by both humans and machines. Available at: http://www.w3.org/TR/xml11/

