A Domain-Specific Language for Railway Control

Philip Eumann

Bachelor's Thesis
September 2017

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth
M.Sc. Nis Börge Wechselberg
Prof. Dr. Reinhard von Hanxleden

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Computer science is a field crucial to all major companies. Computer scientists are needed everywhere, and the universities are always looking for new students in the field. Kiel University holds promotional events such as Girl's Day or taster courses ("Schnupperstudium") to incentivize young people to study computer science. The goal of these events is to present a tempting, but still realistic view of the way computer scientists work.

This thesis proposes a textual language called *RailSL* that can be used by the participants of such events to control a model railway installation. The language is designed to be easily usable by pupils with little to no programming experience. A synchronous language called *SCCharts* is used as host language to ensure determinacy of the programs written in RailSL. The programs are written in an Eclipse-based environment with a live visualization indicating the effects of a statement while it is typed.

Acknowledgements

I would like to thank Prof. Dr. Reinhard von Hanxleden for offering me the opportunity to work on this intriguing topic, as well as my advisors Steven Smyth and Nis Wechselberg for always lending me an ear and for all the helpful advice. Thanks as well to the railway project team for tips, tricks, explanations and debugging help and, last but not least, to Johanna for being there at all times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introduction will incrementally present the key components required to understand what this thesis is about. The first section will explain synchronous languages in general, beginning with classical approaches and concluding with SCCharts, the language that I used to realize this project. Section 1.2 will briefly introduce the KIELER project, the foundation on which the project has been built. The third section will introduce the model railway installation that I used before Section 1.4 states the goals and objectives of this thesis. Finally, Section 1.5 will present the outline of the further thesis.

1.1 Synchronous Languages

Synchronous languages offer determinacy and predictability under all circumstances as well as compile-time validation [BCE+03]. If a compiler for a synchronous language accepts a program, it is guaranteed to run without deadlocks and to behave exactly the same way each time it is given the same sequence of inputs. This makes these languages ideal for embedded and safety-critical real-time applications such as infrastructure systems (e. g., railway interlocking systems, traffic light controllers) or emergency systems (e. g., airbags in cars, reactor emergency shutdown in nuclear power plants).

However, the concepts of synchronous languages and their models of computation may seem unfamiliar and confusing to people who never worked with them before as they differ significantly from widely used languages such as Java or C.

1.1.1 Berry's Synchronous Model of Computation

The central aspect of the synchronous model of computation (MoC), as proposed in the 1990's [BCE+03] and used for the language *Esterel* [Ber00a], is the concept of *perfect synchrony*. Perfect synchrony means that physical time does not affect the execution of a program and that instead, time is divided into logical *ticks*. An execution of a program is a sequence of ticks, also called *macrosteps*, which in turn are composed of a finite number of *microsteps*. All microsteps within the same macrostep are considered instantaneous and take no logical time when executed. While each macrostep is thus guaranteed to terminate, the execution itself may well contain an infinite amount of macrosteps.

Another important part of Berry's synchronous MoC is the concept of *signals*. A signal may be either absent or present and must have a consistent state throughout each tick. A signal can be emitted through an `emit` statement, making it present for the entire duration of the current tick. If no such statement is executed in a tick, the signal remains absent. Programs with non-consistent signal states like Listing 1.1 are rejected by the compiler.

```
Present X else emit X;
```

Listing (1.1) A contradictory Esterel program. The signal changes state after it was already read in the same tick, making its state non-consistent.

```
Present X then emit X else emit X;
```

Listing (1.2) A logically correct, but non-constructive Esterel program.

Modern compilers go even further, considering a program valid if and only if it is *constructive*. Constructiveness, as Berry proposes, means that a signal is present in a tick iff an `emit` statement *must* be executed in that tick. It is absent iff no `emit` statement *can* be executed in the given tick [Ber02].

This forbids some programs, like Listing 1.2, which would be acceptable according to the signal coherence rule explained above as there is only one valid state for X: Having X present would emit it again, which is allowed as reemitting it does not change the state of the signal, making this a valid signal state. Having X absent would mean that an `emit` statement is executed in a tick where X has already been read as absent, which is not allowed. However, this reasoning is counter-intuitive as it moves against the direction of control flow; only the knowledge that a certain signal state will be invalid later in the execution determines the required state for X. The constructiveness analysis would, in the case of Listing 1.2, be unable to determine a signal state for X as at the time of evaluation of the `present` statement, there is no information about the state of X. This way, either of the two alternatives *may*, but neither of them *must* be executed. Therefore, the constructiveness analysis would reject such a program.

In short, constructiveness is a restriction of logical correctness in order to ensure determinacy. However, Berry's synchronous MoC also enforces consistent signal states throughout a tick. While this concept is grounded in the fact that the MoC was originally developed for use with hardware signals [Ber00a], it is not necessary to ensure determinacy; with a carefully chosen set of rules, sequential updates of signals can still be determinately scheduled.

1.1.2 The Sequentially Constructive Model of Computation

The *Sequentially Constructive Model of Computation (SC MoC)*, presented by von Hanxleden et al. in 2014 [HMA+14], introduces the *Initialize-Update-Read* (IUR) protocol. This protocol determines rules for scheduling concurrent accesses to the same variables. It states that all initializations (e. g., `x = 0`) have to be executed first, then all updates (e. g., `x = x + 1`), and finally all reads. So rather than forbidding all concurrent variable writes, as Berry's MoC does, a program can still be accepted if this set of rules permits statically scheduling all concurrent accesses to variables in a deterministic way.

It is also worth noting that these rules only apply to *concurrent* accesses. One thread by itself may well read the same variable and write it later in the same tick. In this MoC, the scheduler does not, as in classical synchronous languages, start from scratch, but uses the *scheduling information already expressed by the programmer*. Lines of code written for a thread are executed in that particular order (unless there are conditional jumps or preemptions of course), so if a programmer writes `if x == false then x = true`, the scheduler will have no trouble reading `x`, then updating its value if it was `false` before. This would also permit a program like Listing 1.1 as the scheduling order is already defined by the way the program was written.

The SC MoC conservatively extends Berry Constructiveness, meaning that any Berry-constructive program will also be sequentially constructive [HMA+14]. However, it allows a programming style more intuitive to those who are mainly used to imperative programming and removes restrictions that are not required to enforce determinacy.

1.1.3 SCCharts

SCCharts is a language proposed by von Hanxleden et al. in 2014 [HDM+13]. It is an extension of *SyncCharts*, which was proposed by Charles André in 1995 [And95]. SyncCharts is a state machine-based language with hierarchy, concurrency and modularity, among other features also found in SCCharts. SyncCharts can be translated to Esterel and vice versa and is thus a synchronous language.

SCCharts applies the SC MoC to the state-machine based approach. This section will only give a short overview on language features used in this project, shown in Figure 1.2.

For a full list of features available in SCCharts, see [HDM+14].

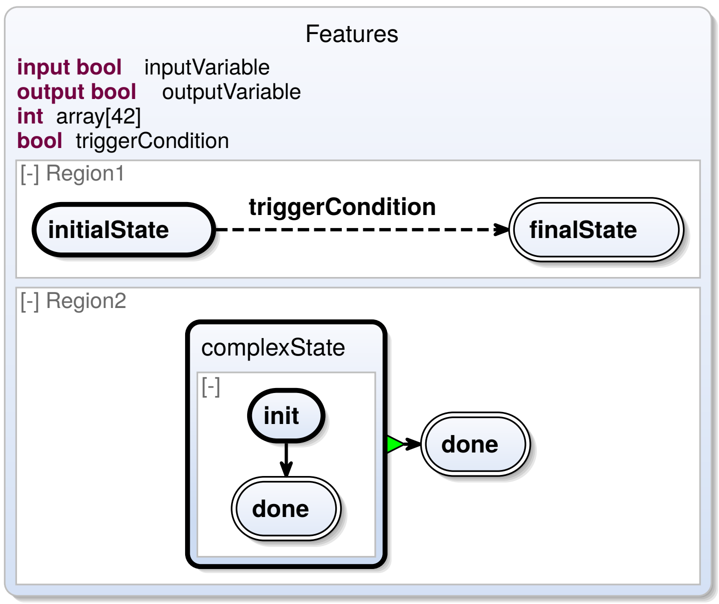


Figure 1.2. An overview of SCCharts features used in this project.

Input / Output variables
: Variables can be declared as inputs or outputs (or both). Input variables are used to feed information about the state of the environment to the SCChart while outputs are set in order to interact with the environment.

Simple and Complex States
: Simple states are only beginning and end points for transitions while complex states have inner behaviour. If a complex state is entered, all of its inner regions are executed in parallel.

Immediate Transition
: If the *trigger condition* is true within a tick, the transition can be taken and the execution is immediately continued in the next state.

Delayed Transition
: This type of transition will not be active in the same tick where its source is reached. Once a tick boundary has been crossed though, the transition becomes active and behaves like an immediate transition until taken.

Parallel Regions
Each parallel region with the same parent scope will be run in its own thread, starting in the region's initial state, which by definition is mandatory and has to be unique.

Normal Termination Transition
This transition is taken when all regions in a complex state have reached one of their final states. Normal termination transitions may be immediate or delayed, even though for this project, only immediate normal terminations are used.

Count Delay
Transitions with a count delay of n will not be triggered as soon as the trigger expression is true, but as soon as it has been true in n separate, not necessarily consecutive ticks.

1.2 The KIELER Project

The *Kiel Integrated Environment for Layout Eclipse RichClient (KIELER)*[1] is a research project developed by the Real-Time and Embedded Systems (RTSys) group[2] at Kiel University[3]. It is based on the Eclipse platform described in Section 3.1 and offers tools for automatic layout of diagrams[4] as well as a powerful compiler based on step-by-step transformation chains from complex source models to executable files [MSH14]. Developers can register their own transformations for custom languages and integrate them with KIELER.

KIELER also comes with an implementation of SCCharts. Rather than modelling the charts with a graphical editor, a textual language called *SCCharts Textual Syntax (SCT)* is used to specify them, from which in turn an actual SCChart can be derived and displayed in real-time using a transient diagram view [SSH13] and the automatic layout capabilities of KIELER. A series of model-to-model (M2M) transformations allows the KIELER compiler (KiCo) to transform SCCharts, via several intermediate formats, to either code in a common programming language (C or Java) or VHDL code for hardware synthesis.

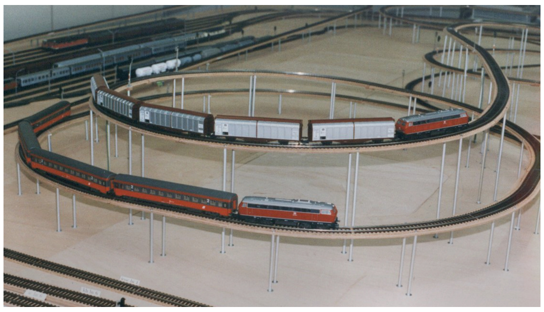
1.3 The Railway Installation

The Institute of Computer Science of Kiel University owns a railway installation consisting of about 127 meters of tracks on an area of 18 square meters[5]. As it is a simple and lucid example of a safety-critical real-time system, the installation is used for teaching and demonstration purposes, for both students and people foreign to the subject of computer science.

1.3.1 History

The first version of the installation was built by the working group for Computer Organization and ready for use in 1997. It was inspired by the *Kicking Horse Pass* in Canada, a railway infamous for having steep and sharp curves and a single track with many trains wanting to pass it. The model railway was designed in close accordance to the real railway (Figure 1.3a), however the installation also suffered

[1]http://rtsys.informatik.uni-kiel.de/kieler
[2]http://www.rtsys.informatik.uni-kiel.de
[3]http://www.uni-kiel.de
[4]https://www.eclipse.org/elk
[5]https://www.informatik.uni-kiel.de/~railway


(a) The central ascent in the first version of the railway installation.

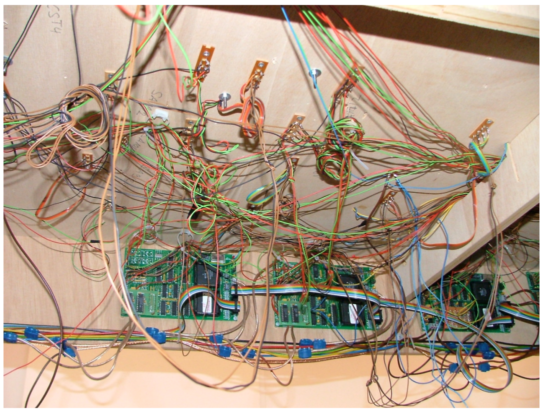

(b) Wiring of the second generation of railway hardware.

Figure 1.3. Images of the old railway installation. Pictures taken from [Wec15].

from the same problems. Some trains were unable to ascend the steep tracks and caused accidents or fell off the higher segments. Also, the control electronics was very susceptible to interference.

As the working group moved to the central campus, the installation remained unused for a while until it was refurbished in 1999, swapping out the old electronics for more robust parts and remodeling the track layout. Realism was traded for ease of use, a double-lane segment was introduced in the middle of the central pass to ease scheduling and keep the trains flowing. The steep ascents were swapped out for flatter segments to reduce the risks of accidents. The new electronics and their wiring, shown in Figure 1.3b, were however not very well documented and hard to maintain in case something broke.

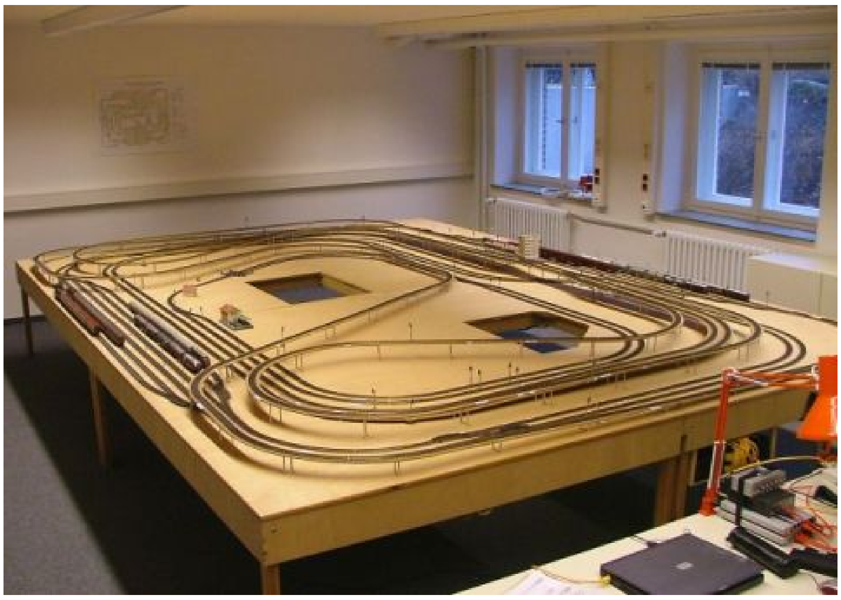


Figure 1.4. The current railway installation. Picture taken from [The06].

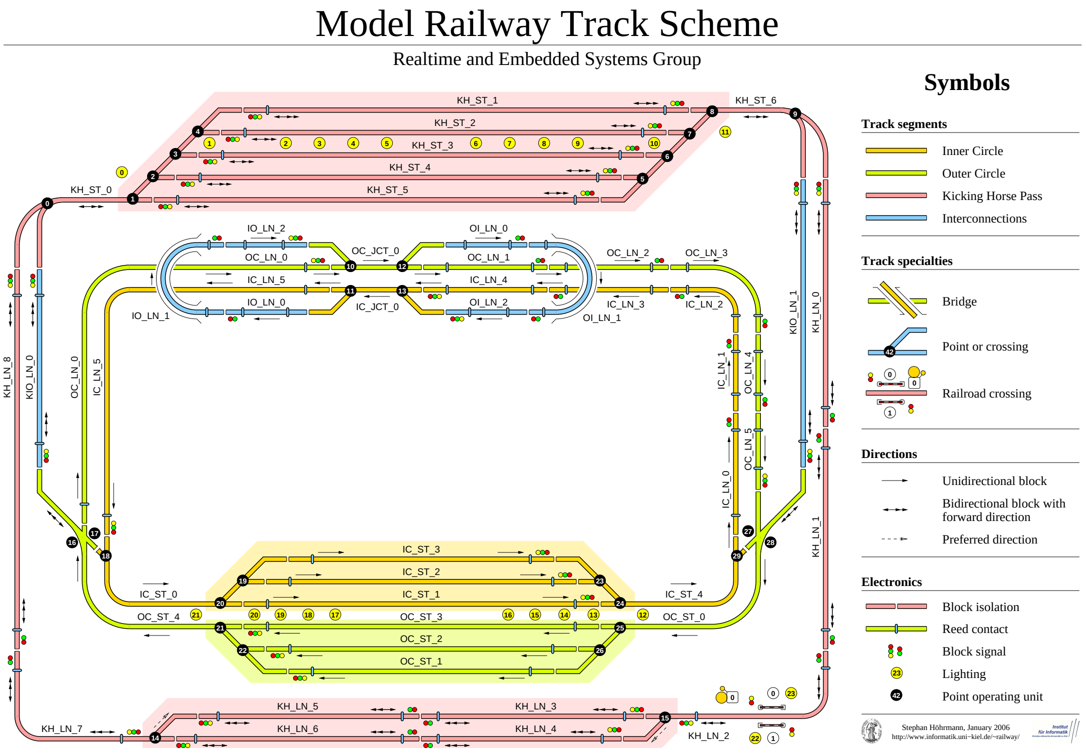


Figure 1.5. The (simplified) trackscheme of the current installation. Schematic taken from [The06].

1.3.2 Technology

In 2002, the Real-Time and Embedded Systems group took over the installation and in 2004 moved it to its current location. A third generation of control hardware was installed in 2005. It consisted of 24 custom-made circuit boards, each controlling a certain section of the installation, connected to a central computer. This time, the wiring was designed with low interference and high maintainability in mind, labelling each of the cables for easy replacement. This version of the railway was used for several lab courses as well as publicity events and presentations. A detailed description of the third generation railway installation can be found in [HFP+06].

Even though this version worked reasonably well for several years, the number of hardware faults started to increase. Parts of the network infrastructure failed repeatedly and several nodes did not boot properly. Therefore, Nis Börge Wechselberg introduced a fourth generation of railway hardware with his master's thesis in 2014 [Wec15].

The fourth generation of railway controller hardware relies mainly on commercial off-the-shelf (COTS) components to ensure an abundance of replacement parts. The low-level control of the railway sensors and actors is handled by Arduino microcontrollers[6], which are connected to Raspberry Pis[7] via USB. Those are in turn connected to the university's LAN and thus reachable from anywhere

[6]https://www.arduino.cc
[7]https://www.raspberrypi.org

within the network. This enables them to communicate with one another as well.

Since the third generation, a series of C header files exists that can be used to compile controllers for use with the model railway installation. They provide an Application Programming Interface (API) that allows for easy communication with the railway hardware as well as connection establishment and closing.

The intricate technical details of the installation are of no particular relevance to this thesis and are thus left out. For an in-depth explanation, please refer to Nis Börge Wechselberg's master's thesis [Wec15].

1.4 Problem Statement

This section defines the purpose of this thesis and describes the idea and motivation behind it.

1.4.1 Target Audience

The Department of Computer Science of Kiel University is involved in various regular promotional events, such as *Girl's Day*[8] or *Schnupperstudium*[9]. In these types of events, young people, typically pupils from local and regional schools, are supposed to get an insight into the subject of computer science. A few participants will have some programming experience from computer science classes in school, however most of them do not. At the very least, they usually lack advanced understanding of programming languages and thus the ability to grasp new programming concepts in the very limited time available during these types of events, as well as knowledge in theoretical computer science and terminology.

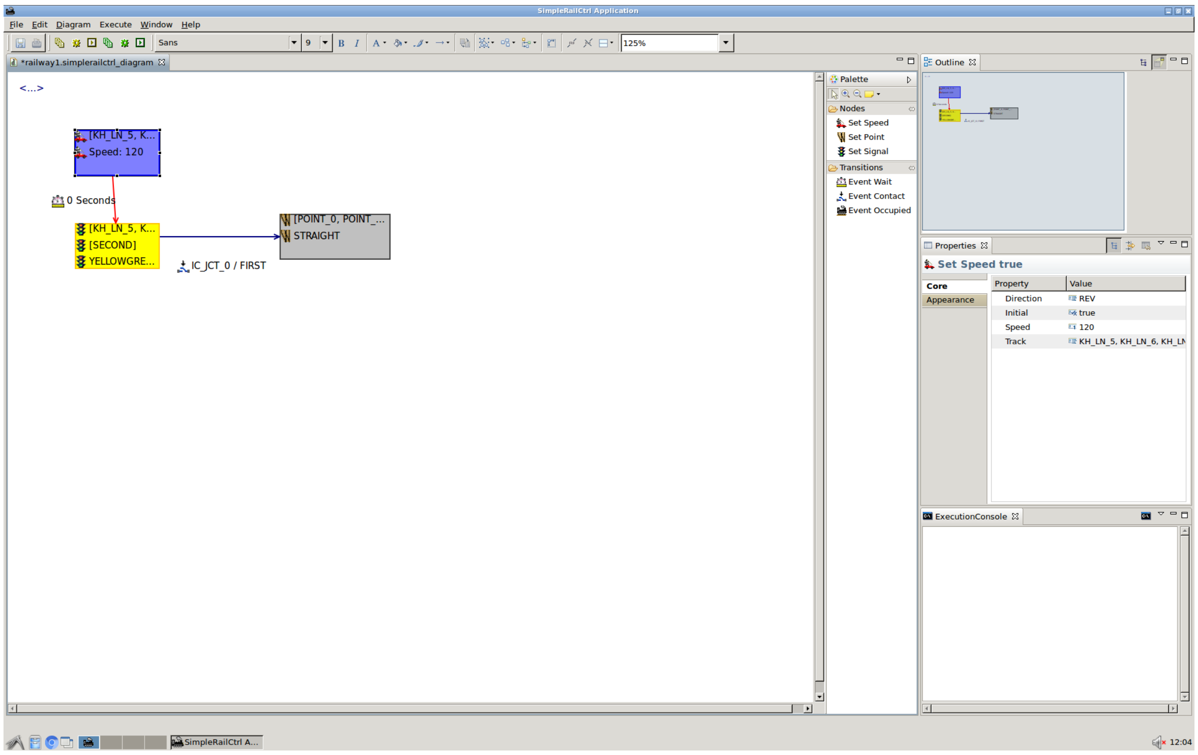
1.4.2 Currently Used Tool

The RTSys group has developed a tool in 2007 that is used in these types of events to allow the participants to control the railway installation. It features a graphical editor to create controlflow diagrams that can be compiled for use with either a SCADE[10]-based simulation or the actual railway installation. Figure 1.6 shows a screenshot of this tool.

The tool is rather outdated. It is not very comfortable to operate, due to being a graphical editor in general and due to some conceptual problems in particular. It uses, for example, an `initial` field for every node in the chart. If this field is set to true, a thread will be spawned that executes the chart, starting from this position. This field was introduced as it is not possible to find a unambiguous schedule for this type of program if it contains cycles, which the tool permits. The chart does not visually indicate that a node is set as initial though. This leads to very common issues with participants setting nodes as initial by accident or misunderstanding the concept and the tutors having difficulties finding the error.

Also, text fields, e. g., lists of track segments to be affected by a node, cannot be displayed properly if their content is too long, which happens regularly. The user then has to click the field, which opens a new window capable of displaying the full content. Additionally, the diagram itself may sometimes be hard to read due to the dark colors used, especially in bright sunlight or with weak projectors.

[8]https://www.inf.uni-kiel.de/en/computer-science-and-scool/girls-day
[9]https://www.inf.uni-kiel.de/en/computer-science-and-scool/taster-courses
[10]http://www.esterel-technologies.com/products/scade-suite



Set Speed true

Core
Appearance

Property	Value
Direction	REV
Initial	true
Speed	120
Track	KH_LN_5, KH_LN_6, KH_LN

Figure 1.6. The tool currently used for modelling railway controllers.

The tool does provide an auto layout option, however its effectiveness is rather limited. To achieve a pleasing layout, the button has to be pressed several times consecutively, moving the nodes a bit each time.

Visual problems aside, the tool uses six buttons, three each for deploying a program to the simulation or the railway. One button is used for code generation, one for compilation, and one for deployment. This is to be able to show the generated code, but it confuses people foreign to the subject, as they do not understand why they need to press three buttons to achieve what appears to be a single task. The generated C code is also a rather inadequate example to show to students, as its formatting and naming conventions obey no apparent rules. The C code is customly generated for this particular application, using semaphores for synchronization instead of synchronous concepts.

In conclusion, the currently used tool has major deficiencies. KIELER offers almost all the things necessary to greatly improve the usability and maintainability of such a tool, easy extensability, good automatic layout and a powerful and transparent compiler.

1.4.3 Language Idea

The goal of this thesis is to present a completely redesigned tool to develop software for use with the department's railway installation. I therefore created a domain-specific language (DSL) called *RailSL* (short for "railway-specific language").

The main objectives are easy usability for complete beginners and good readability of the programs. The language is integrated with KIELER using an Eclipse Xtext-based plugin, which is further explained in Section 3.2. The visual feedback, rather than using flowcharts or SCCharts, is based on a track scheme of the installation and immediate responses to changes in the program. To ensure determinacy, the KIELER compiler's extensability is used to compile RailSL to SCCharts, then down to C code using the standard SCCharts compilation.

1.5 Thesis Outline

Chapter 2 provides a brief look at related work, which will include other synchronous languages that may be considered appropriate for my requirements, other DSLs built for non computer scientists and other instances of model railways being used in teaching.

Chapter 3 will outline the central technologies I used for my project that have not yet been discussed, namely the Eclipse platform and its plugin architecture as well as the Xtext framework.

Chapters 4, 5, and 6 will describe the three main steps of developing the tool:

Language Design
: Chapter 4 explains the way I designed the language, its general concept and changes I made to it during the development process.

Implementation
: In Chapter 5, the implementation of the language is discussed. This includes technical decisions I made in the process, the notion of time in my language and concurrency issues typical to synchronous languages and their models of computation. Additionally, the actual implementation of the KiCo transformation and its corresponding wrapper code is explained.

Live Visualization
: Chapter 6 deals with the integrated visualization of the tool and the way it has been realized.

Finally, Chapter 7 will summarize my work and point out future work that may be tackled at another occasion.

The appendix of this thesis contains a user-level guide to the language, written in a style adapted to the target audience, as well as sample solutions for typical programming challenges used in promotional events.

Chapter 2

Related Work

This chapter presents existing publications dealing with topics similar to the one I have been working on. I will discuss other DSLs, some of which have been developed with objectives similar to those of RailSL, other synchronous languages that could have been used instead of SCCharts as well as other instances of model railway installations used in teaching.

2.1 Domain-Specific Languages

According to van Deursen et al., a DSL is "a small, usually declarative, language that offers expressive power focused on a particular problem domain" [DKV00]. In other words, domain-specific languages are tailored for a certain type of problems, a domain. They sacrifice versatility offered by general-purpose languages like C or Java and are usually not able to handle all types of problems imaginable. In turn, they are usually easier to learn and more powerful regarding the specific type of task they were made for.

While [DKV00] lists dozens of example DSLs from various domains, a particularly interesting topic for this thesis is the usability of DSLs for non-programmers, which has been investigated by Donohue [Don10]. He states that well-designed DSLs can be successfully used by non-programmer domain experts. He conducted a series of interviews with users of different DSLs and found that some, but not all of the users were able to write and maintain programs in the respective DSLs. However, he also found that there were cases where languages specifically designed to be used by non-programmers were not usable in practice.

A successful example of a DSL designed for non-programmers is Quby [Vel12], a ruby-based DSL designed for the creation of medical questionnaires. Veldthuis developed this language as an alternative to modern commercial solutions like Questback (formerly GlobalPark)[1]. He believed that using a DSL could speed up the workflow of the questionnaire designers compared to using the graphical tool that was currently in use. In a trial with domain experts who were unfamiliar with the DSL and who had little to no programming skills, Quby performed quite well. Even though it was done with only three participants, the trial along with the fact that many other users unrelated to the research project have been actively using it shows that the language is easy to learn and quick to use and that it can even compete with commercial solutions.

[1]https://www.questback.com

2.2 Synchronous Languages

This section offers a brief look at various synchronous languages and discuss the advantages and disadvantages of using them in this project.

One such language is *Esterel* [Ber00b], a textual imperative controlflow language currently maintained by Esterel Technologies[2]. It uses the classical synchronous MoC and offers mechanisms for concurrency, preemption and suspension. As mentioned in Section 1.1, these mechanisms are inherently counter-intuitive even to advanced computer science students and therefore not very suitable for non-programmers.

There is a sequentially constructive version of Esterel called *Sequentially Constructive Esterel (SCEst)*, proposed by Rathlev et al. in 2015 [RSM+15]. SCEst extends Esterel to support the SC MoC explained in Section 1.1.1, removing some of the restrictions imposed by the classical synchronous MoC and focusing on software rather than hardware synthesis.

Another commonly used synchronous language is *Lustre* [HCR+91]. Lustre is dataflow-based, meaning that each variable or constant represents not a singular value, but a conceptually infinite *stream* of values and an associated clock, which is technically a boolean stream. In each tick of the program's master clock, it is determined for each variable's clock whether it is `true`. If that is the case, the next value for the associated stream is computed. Lustre, and especially its derivative *SCADE*[3], which provides graphical modeling tools and a formally certified compiler, are widely used in commercial safety-critical applications like train interlocking systems, power plant controls or plane flight controls.

However, SCADE could not be used in this project as it is only commercially available at a quite high cost. Plain Lustre, even though it is powerful and allows for a fast workflow after getting used to it, is just too hard to grasp without prior knowledge or a lengthy explanation.

The main reasons for choosing SCCharts over the alternatives presented here, however, were the easy integration with KIELER, the ease of using Xtext (rather than implementing a custom parser and compiler) as well as the vivid automaton-like visualization of KIELER SCCharts. I would consider the latter to be much more understandable to non-programmers than hundreds of lines of code.

2.3 Model Railways used in Teaching

2.3.1 Digital Process Control Laboratory

In a 1988 paper [McC88], McCormick describes the use of a model railway installation in teaching a course called *Digital Process Control*. Much like at Kiel University, the model railway has been made from commercially available tracks, but most of the electronics were self-made. McCormick presents a list of ten course objectives, including "Concurrency" and "Software Design Methodology for Real-Time Systems" [McC88]. As the computer used to control the installation had no mass storage as well as no operating system, the students had to write their own executive to provide the required services to run C programs. To keep the course size manageable, strict requirements were defined that everyone who wanted to participate in the course had to fulfill.

Even though there were several problems like frequent damage to the delicate trains, different levels of engagement of the students and the tedious process of debugging C code on a machine with limited abilities, McCormick considers the course to be "highly successful" [McC88]. He observed an overall high level of motivation among students taking the course and even students who did not

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

take the course discussing scientific topics in the laboratory room where the installation was located. He also noticed a positive change in the public image of his college, with newspaper articles and television reports featuring it.

He concludes that using a model railway installation for teaching has many advantages and is recommendable if an instructor is willing to take the extra workload required to set up and maintain the installation. However, he also suggests using a higher-level programming language than C to avoid unnecessary difficulties with coding and debugging.

2.3.2 Master's Project

As the multiple instances of renewed railway hardware show, the RTSys group is determined to do exactly as recommended by McCormick. During the time of writing this thesis, the RTSys group of Kiel University offered a project for master's students of which the objective was to develop a universal controller for the group's railway installation using SCCharts[4]. An SCCharts-based model of the railway installation has been created as well to test controllers prior to deploying them to the physical system in order to prevent damage. The students had to integrate their knowledge about concurrent scheduling, deadlock prevention, hardware interfacing and software engineering. This is only the most recent of several teaching events that Kiel University's model railway installation has been used for. The focus of the curricular teaching events at Kiel University lies on high-level model-based development, just as suggested by McCormick. The previous iteration of the Model Railway Master's Project, held in 2014, is considered a success by the tutors, as detailed in the project report [SMS+15].

2.3.3 Teaching Programming to Pre-School Children Using a Model Railway

Noma et al. proposed a model railway toy to teach the basics of programming to pre-school children [NSI+03]. They compare the concept of acquiring literacy through picture books to their concept of acquiring programming knowledge through a railway toy. They present custom components designed to represent statements and control structures and have trains as symbols for the program counter. A track forming a circle makes a loop, a component that sets switch points depending on the cargo of the train represents a conditional jump and so forth. The core idea is not to tell the children that the exercises are meant to be related to programming, but to just let them play and have fun and have them acquire skills needed for programming step by step.

Unfortunately, there is no record of the success or failure of this project; the paper cited here is a mere draft of the concept before even producing the toy and the associated assignments, and I was unable to find any follow-up publications.

2.3.4 Conclusions for this thesis

These examples of model railways being successfully used in or designed for teaching show that the general idea of catching the attention of students as well as providing motivation and physical feedback on what has been programmed using a model railway installation is not quite new, but well-tested and promising.

Students with different levels of knowledge can be taught scientific skills using this method. The availability of such an installation at Kiel University, along with the successful examples from other institutions, makes it almost imperative to use this potential.

[4]https://rtsys.informatik.uni-kiel.de/confluence/display/RP2/Railway+Project+-+Summer+Term+2017

Chapter 3

Used Technologies

3.1 The Eclipse Platform

The Eclipse Platform[1] is an open-source Integrated Development Environment (IDE) originally created by IBM[2] and currently maintained by the Eclipse Foundation. It was originally designed as a Java development platform, but due to its versatility, it is used across all domains today. Its extensability and configurability allow developers to customize the platform, adapting it to their needs. Extension plugins can be easily developed and seamlessly integrated with the existing components.

3.1.1 Plugin Architecture

A central feature of the Eclipse Platform is its plugin architecture. The core Eclipse application is very slim, all noticeable features are implemented through plugins that can be independently loaded and updated. Eclipse provides a simple system of *extension points* and *extensions* to allow for interdependent plugins.

An *extension point* is an interface that can be implemented by any number of *extensions*. The extension point declares fields for strings, booleans or Java class names. A plugin using such an extension point can access services provided by the extension point if it specifies values for all the fields required. Extensions and extension points are both defined using XML.

With extension points, any plugin can contribute functionality and have it cleanly integrated with the existing platform.

3.1.2 The Eclipse Modelling Framework

The *Eclipse Modelling Framework (EMF)*[3] is a framework to allow the use of structured data models in Eclipse plugins. A user can create metamodels describing the possible structures of models. The framework will then generate Java classes and adapters to allow the usage of these models in a Java context.

[1]https://eclipse.org
[2]https://www.ibm.com
[3]https://projects.eclipse.org/projects/modeling.emf

3.2 Xtext and Xtend

Xtext[4] is a framework designed to ease the development of new programming languages. It allows developers to specify a language in a notation very similar to the Extended Backus-Naur Form (EBNF). If desired, Xtext can now automatically generate an EMF metamodel for the language, for which in turn the EMF will generate Java code as described above. Alternatively, one can design a custom EMF model and import it into Xtext. Xtext will also provide a parser along with a completely functional Eclipse editor with syntax highlighting and auto completion for the language.

Xtend[5] is the language used by Xtext. It is used to generate Java code and is syntactically very similar to it, however there are some differences. For example, Xtend allows for extension methods, meaning that for a method `foo` that expects a parameter, instead of writing `foo(X)`, you can also write `X.foo()`. Furthermore, the language introduces the `?` operator to make any procedure call null-safe or makes empty pairs of parentheses optional. All in all, Xtend feels more usable than Java, but still works seamlessly with Java classes or libraries, making it ideal for Eclipse plugin development.

RailSL has been implemented using Xtext and all plugin code is written in Xtend.

[4]https://eclipse.org/Xtext
[5]http://www.eclipse.org/xtend

Chapter 4

Language Design

This chapter will talk about the language design process, its concept and realization and changes that have been incrementally made during the development.

4.1 Basic Concept

As stated in Section 1.4.1, the language is meant to be used by mostly young people with little to no programming experience, let alone knowledge of theoretical computer science. As many programming languages can appear intimidating and confusing due to their heavy use of symbols, brackets or even line alignment as part of their syntax, I decided to use as few symbols as possible and rely mostly on words, numbers and sentences to keep the language as understandable as possible. Most statements are therefore complete english sentences and also end with a full stop to emphasize the sentence character.

I wanted writing a program in RailSL to feel like writing a text. Generally, programming means telling the computer what to do. However, as our natural language is not concise and simple enough for the computers we have, programming languages were designed to be the half-way point between humans and computers, something that both can understand with some effort. To minimize the effort on the human side, DSLs like RailSL introduce another layer of language between other programming languages (C in our case) and the user with the intention to shift this point closer to the human.

There are some statements for which I chose a different syntax than the sentence-based structure described above. This is true for all controlflow statements, namely the conditional and parallel statements and the blocks. I made this decision because parsing the language requires unambiguous structures and making controlflow commands out of sentences seemed overly complicated and user-unfriendly to me.

The reasoning and possible alternatives will be discussed individually for each statement in the following section.

4.2 The Design Process

This section will give an overview of how the language evolved and explain design decisions made in the process.

4.2.1 Initial Version

The initial version of RailSL contained the statements listed in Table 4.1.

These statements allow basic control of all components of the railway installation as well as controlflow based on events from the installation itself or a timer.

Table 4.1. RailSL Statements in the initial version

Statement Type	Purpose	Example Syntax
TrackStatement	Manipulating speed and direction settings of an arbitrary number of track segments	Set track KH_ST_1 to go.
		Set track KH_LN_1, KH_LN_2, KH_LN_3 to reverse slow.
PointStatement	Switching an arbitrary number of points between `straight` and `branch`	Set point 1 to straight.
		Set point 2, 3, 4 to branch.
LightStatement	Turning an arbitrary number of lights on or off	Turn light 1 on.
		Turn light 2, 3, 4 off.
CrossingStatement	Open or close the railway crossing	Open crossing.
ContactWaitStatement	Delaying the execution until a certain contact is triggered	Reach first contact of KH_ST_1.
		Pass second contact of KH_ST_2.
TimeWaitStatement	Delaying the execution for a certain amount of physical time	Wait for 5 seconds.

```
Start:
  Set track KH_ST_1, KH_ST_2, KH_ST_3 to reverse go.
  Set point 5, 7, 8 to branch.
  Reach second contact of KH_ST_4.
  Turn light 4, 5, 6 off.
  Open crossing.
  Wait for 10 seconds.
Loop.
```

Listing 4.1. Examples for first version of RailSL syntax.

In order to avoid code repitition, it is possible to list multiple identifiers, separated by commas, rather than writing the same statement for each one of them.

Except for point and light indices, all numerical constants are hidden behind keywords as this way, it is not necessary to decide on an adequate speed or to memorize the standard numbers. This also prevents errors arising from number range problems and thus makes the language more stable. For example, the speed of a train can either be set to `go` (120), `slow` (45) or `stop` (0) instead of allowing freely entered integers. These particular values were adapted from the Girl's Day worksheets.

As we saw in Section 1.3.2, each train triggers a contact *twice* when passing it: once when the locomotive `reaches` it and again when the end of the last wagon `passes` it. So, for example, to make sure that a segment is empty, the only way is to wait until the closest contact of the next segment in the direction of travel has been triggered *twice*. However, to stop in a station, the contact must be triggered *for the first time* as the train needs to remain behind the contact. Again, in order to avoid code repition, writing the same statement twice to pass a contact and just once to reach it, the `Reach` and `Pass` keywords were introduced. Using them requires no knowledge of reed contacts and magnets in the physical railway, just a brief explanation of what each of them means.

Blocks are not commands for immediate interaction with the railway, but rather a means of structure that surrounds statements. To highlight this difference and for the sake on shortness, blocks are enclosed with simple `Start:` and `End.` markers rather than having full sentences on either side of the block. The End marker of the block can either be `End.`, in which case the block will be executed just once, or `Loop.`, which will cause the block to be repeated forever.

This structure emphasizes start and end of a block and also serves as a visual aid to quickly grasp the sructure of a program. This especially true if all statements within a block are properly indented as shown in Listing 4.1.

4.2.2 Syntax Changes

Instead of setting a track to `reverse slow`, it is now set to `slow reverse`. The position change of the `reverse` keyword made the language closer to natural English.

The word `go` as the constant for full speed has been replaced with `full`, as `go` did not convey the idea of full speed, but rather of not stopping. It might have led to confusion in combination with `slow`, as both of them made the train go, just at different speeds. `full` is clearer as to what it means.

Furthermore, the `and` keyword has been introduced to enumerations as in any natural sentence, one would always write `and` between the second-to-last and last words of an enumeration instead of just putting a comma there. However, to make the language as intuitive as possible, the possibility to use only commas has been preserved while also allowing the use of the Oxford comma.

4.2.3 The Introduction of Conditional Statements

During the implementation process, the problem arose that some desired programs were not feasible with the current version of the language. It was able to do everything necessary for a single train or even multiple trains without interactions. However, there was no way in which trains could synchronize with one another or even communicate at all. For example, a typical Girl's Day Challenge is to have two trains moving in opposite directions, meeting at a suitable point and then continuing at the same time. The *Conditional statement* was introduced to make this possible. With this, multiple alternative continuations can be specified that are chosen depending on the order in which contacts are triggered. If several contacts are triggered in the same tick, the textually first alternative is chosen.

The introducing `Branch:` marker is required to make the language unambiguous, marking which alternatives belong to the same conditional statement. The marker is not a full sentence because as with the blocks, it is not an actual command, but just a means of structuring. The statement allows one block per alternative that may contain any number of statements, including nested conditional statements.

Listing 4.2 shows the syntax and usage of the conditional statement.

However, the conditional statement turned out not to be very usable. It cannot be effectively used to synchronize the arrivals of two trains at their respective destinations as a train, controlled by a dedicated block, cannot check for the other train's arrival while it is still moving itself. Making a third thread as a kind of controller that would always listen for both contacts and act depending on their

```
Start:
  Branch:
    If first contact of KH_ST_1 is reached first, do:
      Start:
        Set point 1 to straight.
      End.
    If second contact of KH_ST_1 is reached first, do:
      Start:
        Set point 1 to branch.
      End.
End.
```

Listing 4.2. Example of a conditional statement.

```
1  Start:
2    Parallel:
3      Start:
4        Turn light 3 off.
5      End.
6
7      Start:
8        Set point 7 to straight.
9      End.
10
11   Wait for 5 seconds.
12 End.
```

Listing 4.3. Example of a parallel statement.

triggering order seems like a plausible solution. As the conditional statement only allows one block per alternative, this controller would, however, be unable to both start two blocks moving the two trains and notify the existing blocks that controlled the trains up to this point.

4.2.4 The Parallel Statement

In order to allow for better synchronization, an explicit *parallel statement* has been introduced. The main program now no longer consists of multiple, but just of a single block. In this block, there may be any number of statements, including parallel statements, which in turn contain two or more blocks. This makes solving tasks like the above one much simpler as there now is an option for sequentially executed blocks. It allows for multiple blocks to be executed (e.g. moving two trains to certain positions) and to start any number of new blocks after all of the previous ones were completely executed.

Listing 4.3 shows how the parallel statement can be used. The TimeWaitStatement in line 12 is executed only after *both* blocks within the parallel statement have terminated.

This, as the `conditional` statement, requires a marker to make it unambiguous, which again is just a word rather than a sentence to keep it simple. The parallel statement also solved some technical issues regarding schedulability, which are discussed in detail in Section 5.3.

4.3 The Final Language

This section contains a full EBNF of RailSL as it is now. For usage examples, please refer to Appendix B.

```
Program ::= Block

// Blocks
Block        ::= 'Start:' Statement {Statement} BlockEnd
BlockEnd     ::= 'Done.'
              |  'Loop.'

// Statements
Statement         ::= SetStatement
                   |  WaitStatement
                   |  OpStatement
                   |  ConditionalStatement
                   |  ParallelStatement
SetStatement      ::= TrackStatement
                   |  PointStatement
WaitStatement     ::= ContactWaitStatement
                   |  TimeWaitStatement
OpStatement       ::= CrossingStatement
                   |  LightStatement

// Track Statement
TrackStatement    ::= 'Set track' SegName
                      {',' SegName}
                      [('and' SegName) | (', and' SegName)]
                      'to' TrackSetting '.'
TrackSetting      ::= (('full' | 'slow') ['reverse'])
                   |  'stop'

// Point Statement
PointStatement    ::= 'Set point' Int
                      {',' Int}
                      [('and' Int) | (', and' Int)]
                      'to' PointSetting '.'
PointSetting      ::= 'straight'
                   |  'branch'

// Contact Wait Statement
ContactWaitStatement     ::= ContactEvent
                             ContactIndex
                             'contact of'
                             SegName'.'
ContactEvent             ::= 'Reach'
                          |  'Pass'
ContactIndex             ::= 'first'
                          |  'second'

// Time Wait Statement
TimeWaitStatement  ::=    'Wait for'
                          Int
                          'seconds.'

// Crossing Statement
CrossingStatement    ::= CrossingSetting
                         'crossing.'
CrossingSetting      ::= 'Open'
                      |  'Close'
```

```

// Light Statement
LightStatement   ::=  'Turn light' Int
                      {, Int}
                      [('and' Int) | (', and' Int)]
                      LightSetting
LightSetting     ::=  'on'
                  |   'off'

// Conditional Statement
ConditionalStatement    ::=  'Branch:'
                             ConditionalLine
                             ConditionalLine
                             {ConditionalLine}
ConditionalLine         ::=  'If' ContactIndex
                             'contact of'
                             SegName
                             'is reached first, do'
                             Block

// Parallel Statement
ParallelStatement    ::=  'Parallel:'
                          Block
                          Block
                          {Block}

// Segment names
SegName      ::=  'KH_ST_0' | 'KH_ST_1' | 'KH_ST_2' | 'KH_ST_3'
               |  'KH_ST_4' | 'KH_ST_5' | 'KH_ST_6'
               |  'KH_LN_0' | 'KH_LN_1' | 'KH_LN_2' | 'KH_LN_3'
               |  'KH_LN_4' | 'KH_LN_5' | 'KH_LN_6' | 'KH_LN_7'
               |  'KH_LN_8' | 'KIO_LN_0' | 'KIO_LN_1'
               |  'OC_ST_0' | 'OC_ST_1' | 'OC_ST_2' | 'OC_ST_3'
               |  'OC_ST_4' | 'OC_LN_0' | 'OC_LN_1' | 'OC_LN_2'
               |  'OC_LN_3' | 'OC_LN_4' | 'OC_LN_5'
               |  'IC_ST_0' | 'IC_ST_1' | 'IC_ST_2' | 'IC_ST_3'
               |  'IC_ST_4' | 'IC_LN_0' | 'IC_LN_1' | 'IC_LN_2'
               |  'IC_LN_3' | 'IC_LN_4' | 'IC_LN_5'
               |  'OC_JCT_0' | 'IC_JCT_0'
               |  'OI_LN_0' | 'OI_LN_1' | 'OI_LN_2'
               |  'IO_LN_0' | 'IO_LN_1' | 'IO_LN_2'

// Non-negative integers
Int              ::=  NonZeroDigit
                      {Digit}
NonZeroDigit     ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Digit            ::=  NonZeroDigit | 0
```

RailSL grammar

Chapter 5

Language Implementation

In this chapter, I will explain the implementation process of RailSL, including design decisions concerning the notion of time in the language as well as concurrency and present the way a RailSL program is transformed into an appropriate SCChart.

5.1 KiCo Compilation

The KIELER Compiler works with incremental model-to-model transformations. The programmer can write their own transformation class using XTend (see Section 3.2) and register it as a KiCo transformation. With a *compilation system*, a set of transformations and metrics can be combined and used like a single transformation.

The SCCharts compilation system contains 28 transformations that will remove complex features one at a time and replace them with more basic ones until the last transformation outputs C code.

In order to use the KIELER compiler for RailSL, an additional transformation was required that would generate an equivalent SCChart from a RailSL program and let the standard SCCharts compilation system take care of the generation of C code from that chart.

5.2 Logical and Physical Time

As already discussed in Section 1.1, the notion of time in synchronous languages is quite different from what Java or C programmers are used to. This section deals with design decisions that were necessary due to these differences.

5.2.1 Physical Time in Synchronous Languages

Synchronous languages deal with logical time, divided into discrete ticks, rather than with physical time divided into seconds or milliseconds. Therefore, synchronous languages like SCCharts offer no built-in option to suspend the program execution for a certain duration of physical time. They only allow to either wait for one logical tick, which can have any physical duration and is thus unsuitable to measure actual time, or to wait for a certain *event*, either a guard expression becoming `true` or a signal being present. Therefore, a typical approach to integrating physical time into synchronous programs is to have an input signal that will be emitted by the environment in regular intervals of physical time. This way, the program can assume that no matter how many ticks have passed, if the signal is present in a tick, this tick must be at least one physical time interval after the last time the signal has been present.

This mechanism may however produce intervals longer than intended. As opposed to interrupt-based systems like modern personal computers, a synchronous program will not react to a timer signal immediately. Instead, it will delay the reaction until the current tick has been completed as the program will typically read the inputs at the beginning of each tick. If a program has a computationally complex `tick` function or runs on a system with limited resources, it may take a relatively long time to process a single tick.

Still, this delay has a limited length. As stated in Section 1.1, each tick in a synchronous program may only contain a finite number of microsteps. Each of these microsteps is in turn an operation that runs in finite (and usually well-known) time. This way, it is generally possible to compute the *worst-case execution time (WCET)* of a program. The WCET is the maximum time it can take to compute a full tick. This value obviously depends on the program and its complexity, but also on the machine it is running on.

The delay caused by the current tick being completed before reacting to the timer signal can therefore be no longer than the controller's WCET. So far, no WCET measurements have been performed for any controller generated from a RailSL program. However, as typical RailSL programs are rather simple and will be executed on personal computers or the university's ThinLinc clients[1], delays will presumably be short. Also, the only place in which physical time is used in RailSL is with the `TimeWaitStatement` (see Section 5.4.5), where the execution is meant to be delayed anyway, and by several orders of magnitude more than a single tick.

5.2.2 Logical Time in RailSL

With synchronous programming, the question of tick boundaries always arises. Conceptually, everything that can or does happen physically at once should be within the same tick and tick boundaries should be introduced in the places where physical time passes between statements. In RailSL, there are tick boundaries in two different places:

- In a TimeWaitStatement
- Between the last and the initial state of a chart generated from a block with a `Loop.` end marker

The former is there for the reasons described above. A physical timeout should never terminate instantly. The latter does not obey the rule stated above as there does not necessarily have to be a physical delay between the last and first statements of a loop. However, as mentioned before, instantaneous loops are forbidden in synchronous languages like SCCharts. Without an artificially introduced tick boundary, it would be possible to write programs that would theoretically work on the railway installation, using contact events for safe operation and leaving out any timeouts, that could not be compiled as the KiCo would reject them as not schedulable due to an instantaneous cycle.

As discussed in the previous section, this delays the execution of the program by at most its WCET with each completed loop. As loops will typically run for several seconds or even minutes, this has no observable impact on the railway's behaviour.

[1]https://www.cendio.com/thinlinc/download

5.3 Concurrency

Another issue that had to be resolved was the question of concurrency. In early versions of RailSL, multiple parallel blocks at the top level were legal. The transformation turned them into parallel regions within the same root state so that they would be executed in parallel.

Some of the problems that this concept caused were already discussed in Section 4.2.3. Also, the KIELER compiler does not accept programs with concurrent and non-confluent accesses to the same variable. Two assignments are considered confluent iff the order in which they are executed does not affect the resulting variable value. For a formal definition, see [HMA+14]. As all regions potentially controlling different trains were concurrent by definition, it was impossible to have two regions manipulate the same track segment, switch point or even lamp, even if it was ensured that these two events could never happen at the same time.

5.3.1 Combine Functions

A possible fix for this problem would be Esterel-style combine functions. Every region would get its own set of output variables and write exclusively to those while sharing the inputs as usual. This way, each region could manipulate the variables without restrictions. A combine region would compare all variables set by the different regions and decide what value to propagate to the global output variables.

In practice though, designing the combine functions is complicated, which is why this approach turned out to be unusable. All regions would usually configure all switch points and tracks that their train needs for the next journey, then wait until the train reaches an end contact. If two regions set the same component to two different settings, there is no safe way to combine these two inputs; the combination will always cause unexpected behaviour for at least one of the two regions.

As described above, concurrent and non-confluent accesses to the same variable cause the KiCo to reject the program. This is actually beneficial to RailSL and its safety as multiple parallel regions accessing the same variable may mean that multiple trains will soon move across the same track segments or switch points. To avoid this, as well as potential crashes and damage to the installation resulting from it, RailSL still uses parallel regions without any combine function. This way, the KiCo will reject any program with potentially colliding trains.

5.3.2 Introducing Sequential Blocks Using the Parallel Statement

One of the problems described in Section 4.2.3 is that trains cannot move across the same track segment even if it is ensured that the segment is free whenever a train reaches it. The problem that causes this is not the fact that concurrent regions cannot access the same variables, but rather the general parallelity of *all* blocks, no matter whether their behaviour actually takes place at the same time.

With the introduction of an explicit parallel statement, this problem can be solved. As described in Section 4.2.4, the parallel statement allows the user to specify two or more blocks that should be executed in parallel. These blocks are translated into concurrent regions, where it is beneficial to have the KiCo reject programs that have conflicting variable accesses. However, if blocks are not part of the same parallel statement, they are no longer concurrent and therefore can use the same components of the railway.

This way, unsafe behaviour with multiple trains moving onto the same track segment cannot be modelled as before while at the same time, it became possible to express explicitly that blocks are in fact not parallel, but sequential.

5.4 RailSL statements and their translations to SCCharts

This section lists all statements currently supported by RailSL and explains how each of them is represented in SCCharts.

Generally, each statement is represented by a complex state's inner behaviour.

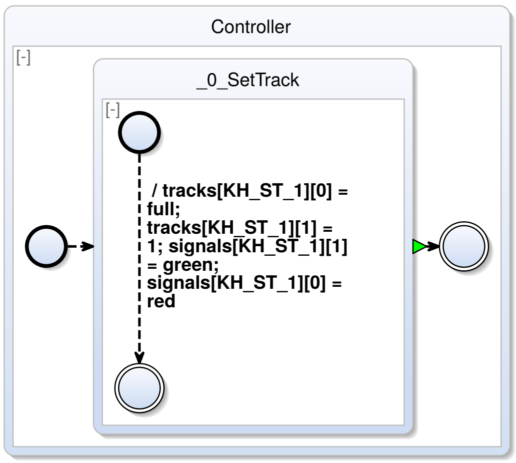
5.4.1 TrackStatement

The `TrackStatement` is used to set any number of tracks to a certain speed level. It can also determine the direction of travel and will automatically set the signals corresponding to the selected tracks appropriately.

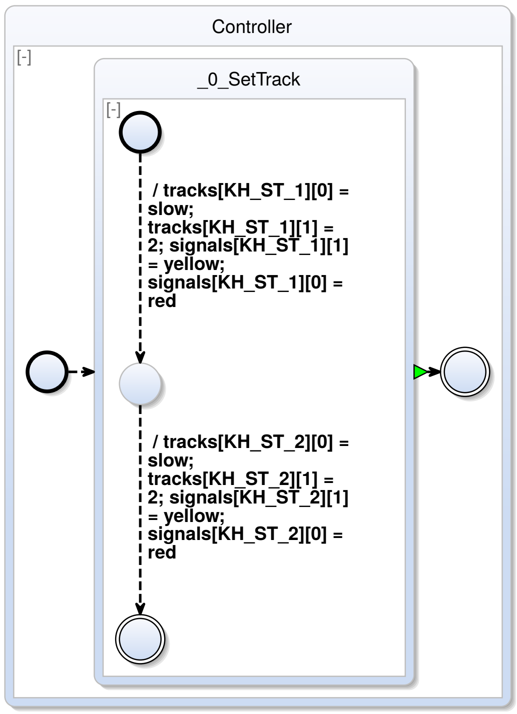
Figure 5.1 shows examples of `TrackStatements` with one (Figure 5.1a) and two (Figure 5.1b) track segments and the SCCharts generated from them.

As with several other types of statements, one anonymous state is created per track. An immediate transition is created for each track segment adressed in the statement, setting the four values associated with the segment as its effect. These are, in order of appearance: speed (first field of `tracks` array), direction of travel (second field of `tracks` array), second and first signals in the current direction of travel. The first signal, pointing towards where the train is coming from, is always set to `red`. For `full` speed, it is set to `green` and for `slow` travel to `yellow`. These, as well as all other constants, are defined in the root SCChart and not shown here.

The control flow within the state is linear and ends in the same tick the state was entered.

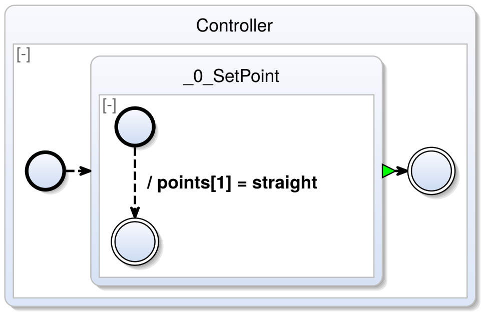

Controller
[-]
_0_SetTrack
[-]
/ tracks[KH_ST_1][0] = full; tracks[KH_ST_1][1] = 1; signals[KH_ST_1][1] = green; signals[KH_ST_1][0] = red

(a) `Set track KH_ST_1 to full.`

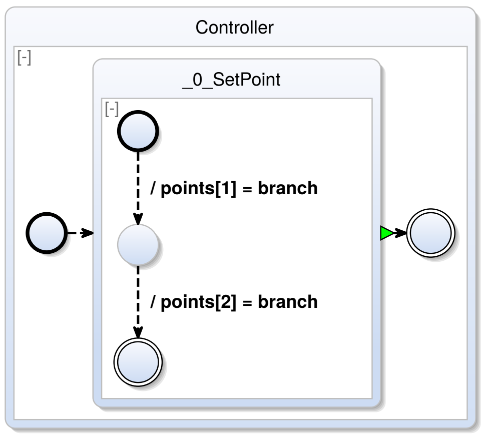

Controller
[-]
_0_SetTrack
[-]
/ tracks[KH_ST_1][0] = slow; tracks[KH_ST_1][1] = 2; signals[KH_ST_1][1] = yellow; signals[KH_ST_1][0] = red
/ tracks[KH_ST_2][0] = slow; tracks[KH_ST_2][1] = 2; signals[KH_ST_2][1] = yellow; signals[KH_ST_2][0] = red

(b) `Set track KH_ST_1 and KH_ST_2 to slow reverse.`

Figure 5.1. `TrackStatements` and the corresponding SCCharts generated from them.



(a) `Set point 1 to straight.`



(b) `Set point 1, 2 to branch.`

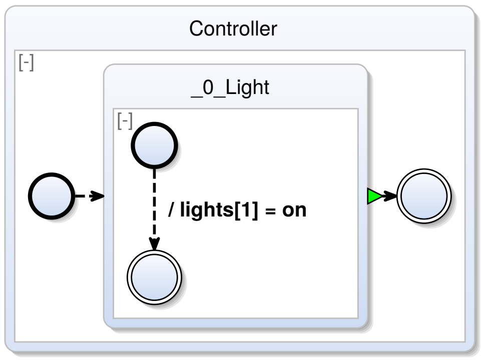
Figure 5.2. `PointStatements` and the corresponding SCCharts generated from them.

5.4.2 PointStatement

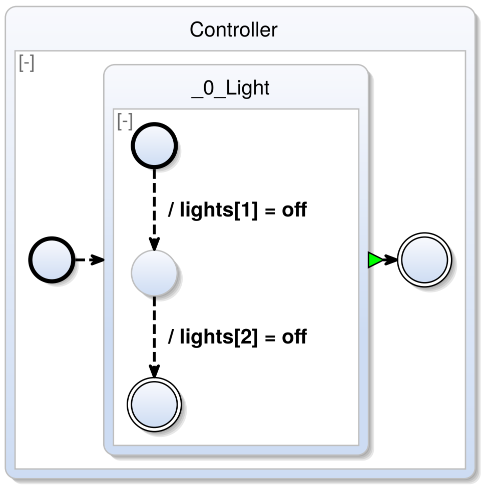
The `PointStatement` is used to toggle switch points between their straight and branched states. Figure 5.2 shows SCCharts generated from this type of statement. They function like the `TrackStatement` presented in the previous section. The only difference is that here, only one variable needs to be set as opposed to the four required in the previous statement.

5.4.3 LightStatement

The `LightStatement` is used to control the installation's street lights and other lights not related to the operation of the railway itself. Like the previous two, this type of statement creates one state per light index listed and sets the appropriate variable, as can be seen in Figure 5.3.

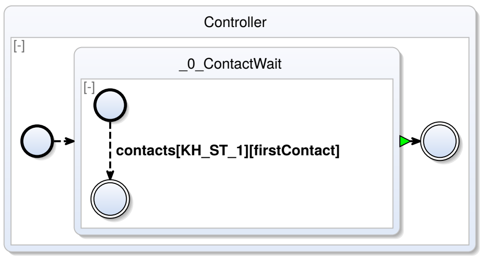


(a) `Turn light 1 on.`

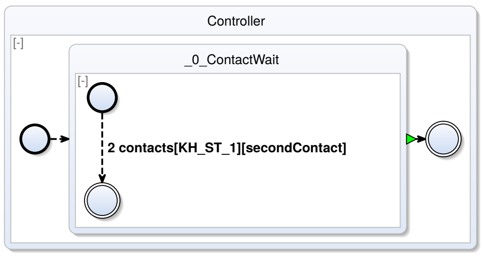


(b) `Turn light 1, 2 off.`

Figure 5.3. `LightStatements` and the corresponding SCCharts generated from them.



(a) `Reach first contact of KH_ST_1.`



(b) `Pass second contact of KH_ST_1.`

Figure 5.4. `ContactWaitStatements` and the corresponding SCCharts generated from them.

5.4.4 ContactWaitStatement

The `ContactWaitStatement` delays the execution until a certain contact is triggered once if the keyword `Reach` was used (Figure 5.4a), or twice if `Pass` was used (Figure 5.4b). This is realized with a count delay transition, indicated by the number 2 in Figure 5.4b. The way count delay transitions work is explained in Section 1.1.3.

5.4.5 TimeWaitStatement

The `TimeWaitStatement` delays the execution of the program by a certain number of seconds. This is realized very similarly to the previous statement, however this time, the variable used is set not in accordance with the state of the railway installation, but by a dedicated timer in the program's wrapper code. This is due to the issues discussed in Section 5.2. To see in more detail how the wrapper code works, see Section 5.5.

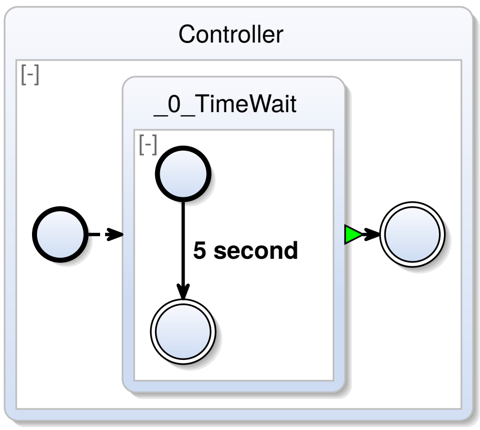


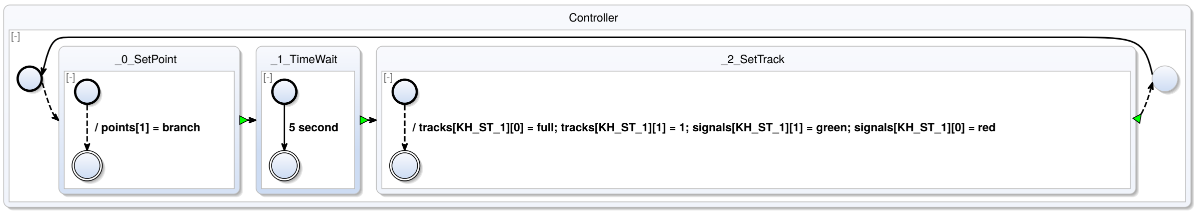
Figure 5.5. SCChart generated from `Wait for 5 seconds.`

5.4.6 Blocks

Figure 5.6a shows a program containing three statements and the SCChart generated from it.

To make the resulting SCChart as readable as possible and to visually show the separation of the different statements, generally one state is generated for each statement according to the rules explained above. An anonymous initial state is generated for each block with an immediate transition to the first statement. As the inner behaviour of each statement reaches a final state after the statement was executed, termination transitions can be used to connect the states with one another.

After the last statement was executed, its transition either leads to the chart's final state if the program ends with `End.` or returns to the initial state if it ends with `Loop.` (as shown in Figure 5.6a).


Controller
_0_SetPoint
/ points[1] = branch
_1_TimeWait
5 second
_2_SetTrack
/ tracks[KH_ST_1][0] = full; tracks[KH_ST_1][1] = 1; signals[KH_ST_1][1] = green; signals[KH_ST_1][0] = red

(a) Generated SCChart

```
Start:
  Set point 1 to branch.
  Wait for 5 seconds.
  Set track KH_ST_1 to full.
Loop.
```

(b) The corresponding RailSL code

Figure 5.6. An SCChart generated from a looped block with three statements.

5.4.7 ParallelStatement

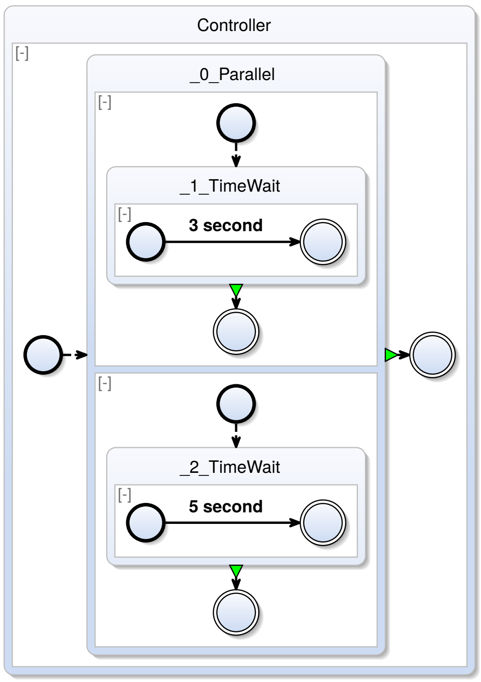
The parallel statement is used to start two or more blocks simultaneously that will be executed in parallel. The statement terminates once *all* of its blocks terminate

To realize this, the inner behaviour of the state representing the statement is generated with one region per block, as can be seen in Figure 5.7a. These regions are then generated according to the standard rules for blocks described above. By definition, the normal termination transition that leads out of the state will only be triggered once *all* regions have reached their final states.

5.4.8 ConditionalStatement

The `ConditionalStatement` will execute the block with the trigger expression that becomes true first. All other blocks are ignored. If multiple trigger expressions become true in the same tick, only the textually first one of them is executed. This is to ensure determinacy, however due to the short tick times and the relatively long time it takes the trains to go from one contact to the next, this case should occur very rarely in practice.

As Figure 5.8a shows, a state is generated for each block. Again, the inner behaviour of these states is generated according to the block rules specified above. Transitions with the appropriate triggers and the precedence specified by the textual order of the blocks are generated from the initial state to each block. A normal termination transition is added from each block to a joint final state.



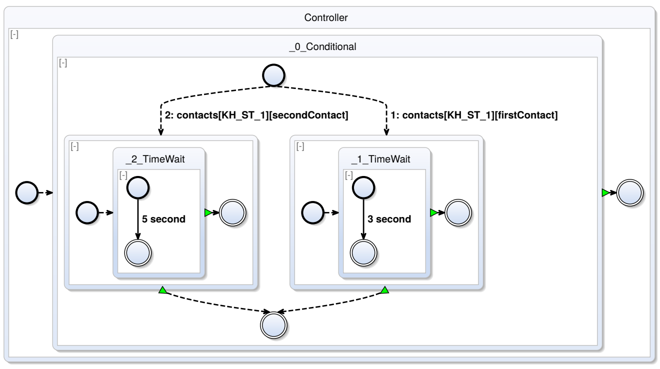
(a) Generated SCChart

```
Start:
  Parallel:
    Start:
      Wait for 3 seconds.
    End.

    Start:
      Wait for 5 seconds.
    End.
End.
```

(b) The corresponding RailSL code

Figure 5.7. An SCChart generated from a `ParallelStatement` with two blocks, each containing a `TimeWaitStatement`



(a) Generated SCChart

```
Start:
 Branch:
  If first contact of KH_ST_1 is
     reached first, do
    Start:
      Wait for 3 seconds.
    End.
  If second contact of KH_ST_1
     is reached first, do
    Start:
      Wait for 5 seconds.
    End.
End.
```

(b) The corresponding RailSL code

Figure 5.8. An SCChart generated from a `ConditionalStatement` with two blocks, each containing a `ContactWaitStatement`

5.5 Wrapper Code and Compilation

While the KIELER compiler is able to produce code in multiple languages, RailSL models are compiled to C. From the SCChart that has been created by the transformation described in the previous section, it generates a file containing a struct called `TickData` that saves the values of all signals, variables and guards occuring in the chart as well as a `tick` function taking a `TickData` struct as its parameter. This function contains the sequentialized logic of the chart and will, if called with a valid struct, perform a tick and update the struct to contain the new values for all variables. KiCo also provides a `reset` function that initializes all fields of the struct to their standard values.

However, this code cannot be compiled directly as it requires an adequate `main` function. This so-called *wrapper code* is required to compile any SCChart into an executable file. KIELER offers a module called *Project Management (PROM)* [Sta15] that allows for automatic generation of wrapper code based on variable annotations and a template engine. However, PROM has not been used in this project as the wrapper code is static and does therefore not need to be generated individually for each program.

The API for the railway and thus most of the functions used in the wrapper code are documented in Stephan Höhrmann's diploma thesis [Höh06].

The following paragraphs will give an overview of the wrapper code's structure.

The `main` function first performs several setup and initialization steps such as establishing the network connection to the railway and creating the `TickData` struct and resetting it, then runs in a loop that will remain active until the `_TERM` field of the struct is set.

The wrapper code also keeps track of all output variable values from the previous tick, the so-called `pre` values. This is to reduce load on the railway's network infrastructure because this way, API calls have to be made only for the values that changed in the current tick.

The loop performs the following steps in each iteration:

Scan contacts
At the beginning of each tick, all contacts from tracks that are currently active and that have contacts are scanned via an API call. The values are stored in the struct's `contacts` field and can be accessed by the chart's logic. To reduce load, contacts of inactive tracks are not scanned as they cannot be triggered. Contacts are reset after being scanned, ensuring that for each event, the corresponding variable in the struct will be true for exactly one tick. As the railway hardware will sometimes produce ghost contact events shortly after powering on, one of the inactive tracks is scanned each tick in round robin mode as well to flush out these events.

Compute time
Then, it is checked whether at least one second has elapsed since the last time the `second` signal was set to true. If that is the case, `second` is set to true for the next tick and the timer is reset.

Perform tick
At this point, the `tick` function is called, passing the `TickData` struct to it that contains, among other data, the freshly computed inputs. The `tick` function computes the new state of the chart and writes it back to the struct.

Write outputs
For each output variable, it is checked whether it changed during the last tick. If and only if that is the case, an API call is performed to update the railway installation according to the new value.

Update `pre` values
Set the locally saved `pre` values to the newly computed values from the `TickData` struct.

Sleep
Pause the execution for 100 microseconds. This is because on modern computers, tick times are so short that the railway installation cannot, even with the measures to reduce load, handle the requests fast enough. Testing has shown that with no delay, the Raspberry Pis handling the requests will stop responding within seconds, causing timeouts in the controller. This delay is just long enough that in tests the Rapsberry Pis could keep up with the number of requests.

When `_TERM` becomes true and the `while` loop terminates, the chart has reached its final state and no further steps are required. The wrapper code then closes the connection to the railway and terminates.

To compile the generated C code, two different makefiles are available, depending on whether the controller should run on a simulation or the physical railway installation. One of the two makefiles was written in 2007 by Christian Motika and compiles the controller for simulation along with the simulator itself. The other makefile, the one used for deployment to the actual railway, was developed during the railway project described in Section 2.3.2.

At the time of writing this thesis, there is no automatic workflow for the compilation. The possibilities and requirements will be further discussed in Section 7.2.

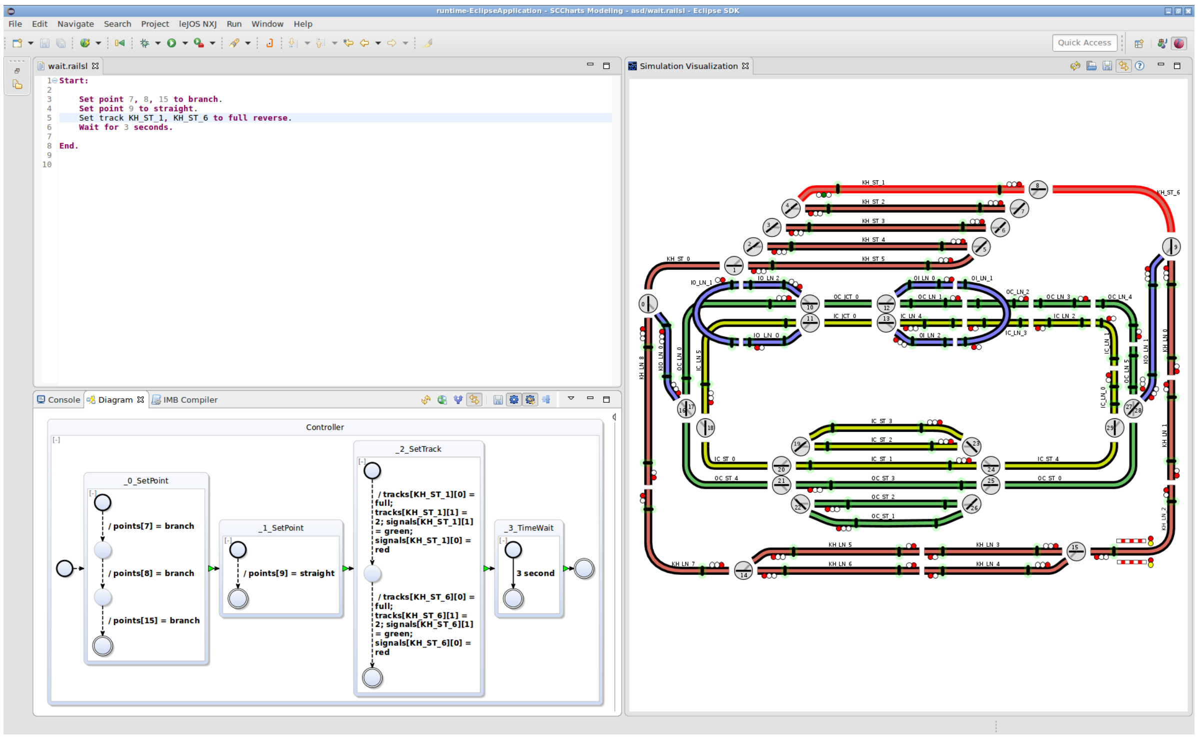
Chapter 6

Live Visualization

This chapter describes the idea of on-the-fly visualization of the program being written in the editor. The concept of transient views responding to editor changes is not new in KIELER; the *KIELER Lightweight Diagrams (KLighD)* view [SSH13], which is used to display the SCCharts modeled in the editor, originally responded directly to changes in the editor, however this approach has been abandoned due to performance issues with large models.

6.1 General Idea

The general idea of the visualization is to give immediate visual feedback on what is being modelled to reduce the probability of programming errors. The idea to show the track layout and highlight the components being adressed came to mind quickly, as this concept can be seen in the old simulator. KIELER offered a component called *KIELER Environment Visualization (KEV)*, which was replaced by the *KIELER Visualization (KiViz)* during the development process. The visualization of RailSL was


runtime-EclipseApplication - SCCharts Modeling - asd/wait.railsl - Eclipse SDK
File Edit Navigate Search Project leJOS NXJ Run Window Help
Quick Access
wait.railsl
Simulation Visualization
Console Diagram IMB Compiler
Controller
_0_SetPoint
/ points[7] = branch
/ points[8] = branch
/ points[15] = branch
_1_SetPoint
/ points[9] = straight
_2_SetTrack
/ tracks[KH_ST_1][0] = full; tracks[KH_ST_1][1] = 2; signals[KH_ST_1][1] = green; signals[KH_ST_1][0] = red
/ tracks[KH_ST_6][0] = full; tracks[KH_ST_6][1] = 2; signals[KH_ST_6][1] = green; signals[KH_ST_6][0] = red
_3_TimeWait
3 second
KH_ST_1
KH_ST_2
KH_ST_3
KH_ST_4
KH_ST_5
KH_ST_6
KH_ST_0
KH_LN_8
KH_LN_7
KH_LN_6
KH_LN_5
KH_LN_4
KH_LN_3
IC_ST_3
IC_ST_2
IC_ST_1
IC_ST_4
IC_ST_0
OC_ST_4
OC_ST_3
OC_ST_2
OC_ST_1
OC_ST_0

```
Start:

    Set point 7, 8, 15 to branch.
    Set point 9 to straight.
    Set track KH_ST_1, KH_ST_6 to full reverse.
    Wait for 3 seconds.

End.
```

Figure 6.1. Example of a program and its visualization in KIELER

realized using KiViz.

KIELER offers tools to simulate and interact with SCCharts and to demonstrate the behaviour of the model being simulated. The KiViz view is able to display an image that can, through a customized mapping from model variables to image elements, react to changes in the simulation data pool. This mapping works with a data pool from other sources as well and with a matching mapping, the data pool can be visualized.

The KiViz view updates as a line of RailSL code is entered to show on the track layout what would change if that line was executed. Setting the speed of a track highlights it in the appropriate color according to the palette used by the new simulation developed during the master's project while setting the signals appropriately. Setting a point should flip it on the scheme, scanning contact events should highlight the adressed contact. Figure 6.1 shows the RailSL editor (left), where a `TrackStatement` is being edited, and the corresponding visualization (right) with the matching tracks highlighted appropriately.

6.2 Implementation

In order to realize this visualization, changes to the editor's content need to be detected as they should trigger an update of the visualization. For this purpose, Eclipse offers the `IDocumentListener` interface. An instance of a class implementing this interface can be attached to an editor's active document and will then be notified whenever a change is about to be made to the document, then once again when the change has been performed. The (anonymous) Listener class of RailSL ignores the first event and only reacts once the update is complete by notifying the singleton instance of the `Visualizer` class, which contains the core logic for the visualization.

To ensure that the `IDocumentListener` is always attached to the correct document, the Eclipse interface `IPartListener2` was used. Instances of this interface are notified whenever parts of the Eclipse window (the so-called *workbench*) are opened, closed or activated, among others. Closing and reopening the editor or opening a new document disables the currently active `IDocumentListener` and switching between multiple open editor tabs may cause it to be attached to the wrong document. Therefore, the `IPartListener2` was configured to attach a new listener to a document whenever an editor is activated, which happens when opening the view or when loading a new file, and to remove it when the editor is deactivated or closed. This ensures that the `Visualizer` will be notified reliably whenever the content of the editor changes.

Xtext's `EObjectAtOffsetHelper` is used to determine the `EObject` generated at a particular cursor position in an editor. With this, it is simple to find the statement being currently written or edited and to generate an appropriate `DataPool` object for the KiViz view. This pool contains variable names and their values. An additional mapping from these variables to image elements had to be created.

The `Visualizer` uses a default `DataPool` with all tracks set to off, all contacts deselected and all switch points set to straight that is cloned with each update. The specific values needed to represent the statement currently being edited are added and the pool is passed to the `updateView` function of the KiViz view.

From a performance perspective, it would be sensible to keep track of the values that have been added to the pool to visualize a statement in order to be able to remove them again, going back to the original pool, and then readd values for the next statement. This would reduce the reaction time of the visualization as the (rather large) data pool would not have to be cloned each time. However, the simulation creates a new `DataPool` object each time a value changes and the view will ignore it if the same pool as before is passed as in this case, no update would be required in a simulation scenario.

Chapter 7

Conclusion

7.1 Summary

The objective of this thesis was to develop a DSL to allow non-programmers to write controller programs for use with the department's model railway installation. The language, called RailSL, is primarily meant to be used in promotional events with very limited time frames in which the participants need to understand the concept of the language and produce working code solving different challenges. Due to these requirements, the language has to be simple and intuitive, which was achieved by keeping the language as close as possible to English. It uses few symbols like semicolons or brackets and concludes every statement with a full stop in order to emphasize the natural language character. There is a manageable number of statement types and it is quite clear what each one of them does. The use of the language is eased further by the autocomplete options offered by the editor and the integrated visualization.

To implement the language, available features of KIELER, the Eclipse platform and Xtext were used as the foundation of the newly developed components. After specifying the grammar for RailSL, Xtext generated an Eclipse editor with syntax highlighting and autocompletion along with a parser and EMF model generator. Thanks to the EMF, the model generated by the Xtext parser can be used in a Java context. A custom KiCo transformation converts the RailSL model to an SCCharts model, which is then compiled with the standard SCCharts compilation chain, generating C code. This C code can then be compiled together with a file containing wrapper code as an interface between the controller program and either the railway installation or a simulation using a predefined makefile.

The language is ready to be used and creates fully functional controllers, however some improvements are left for future work.

7.2 Future Work

This section lists topics that are left for future projects.

7.2.1 Improved workflow

At the time of writing, the language, though fully functional, does not offer a very user-friendly workflow. It is currently necessary to copy and paste the C code generated by KIELER into a correctly named file along with the makefile, wrapper code, required headers and, in case it is compiled for simulation, the files needed for that as well. The compilation itself currently has to be manually run as well, then the resulting binary has to be started by hand via a command line.

Eclipse does offer options to greatly improve this workflow. It would be possible to automatically configure a *run configuration* for each project created with the *New RailSL project* wizard. This run configuration should execute a KiCo configuration (`kibuild`) file that generates the C code, then saves it to an appropriate output directory. The project creation wizard should also create all files required for the compilation of the generated code so that the run configuration can call `make` and afterwards run the generated executable file.

7.2.2 Simulation in KIELER

As mentioned earlier, the team of the master's railway project that took place recently (see Section 2.3.2) developed an SCCharts-based simulation of the railway installation that can be used directly with the simulation mechanism of KIELER. However, at the time of development, it was not possible to integrate this simulation with the RailSL system as it had not yet been migrated to the latest version of KIELER and was thus not compatible with my code. For future work, it would be desirable to use this new simulation instead of the old standalone version as it would greatly simplify the simulation of RailSL programs. This way, a single button could start the simulation right from the code in the editor, removing a lot of overhead.

Appendix A

User Documentation

A.1 Introduction

RailSL is a textual language designed for quick and simple programming. It allows the construction of controller software for the model railway installation maintained by the Department of Computer Science of Kiel University. It is meant to be easily usable even by people with little to no programming experience. This manual explains the usage of the language and, for the sake of understandability, leaves out a lot of technical detail, which the interested reader can find in Philip Eumann's Bachelor's thesis[1]. If any questions remain unanswered by this manual, please feel free to contact Philip Eumann (stu121235@mail.uni-kiel.de).

A.2 The Installation

The Department of Computer Science's model railway installation is, as can be seen in Figure A.1, divided into four parts:

Kicking Horse Pass
This part of the installation (highlighted red in the track scheme) can be used by trains travelling in both directions. However, travelling *clockwise* is considered the main direction of travel. The *Kicking Horse Station* (at the top of the scheme) has enough space for five trains and can also be used in both directions. The two segments at the bottom have little arrows next to them that indicate the preferred direction of travel. This means that if a train travels clockwise, it should preferably take the bottom route and the top route if it travels the other way round.

Outer Circle
The green Outer Circle and its station may only be used by trains travelling *clockwise*.

Inner Circle
Trains using the yellow Inner Circle and its station must travel *counter-clockwise*.

Connections
There are several connecting segments between the three circles, highlighted in blue. Each one of them may be used according to the arrows shown next to them.

An interesting property of this installation, which it shares with most other model railways, is the fact that the trains cannot be controlled directly. Instead, the track segments they run on are set to different speed settings.

The details of the different components are described in Section A.4.

[1]http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/peu-bt.pdf

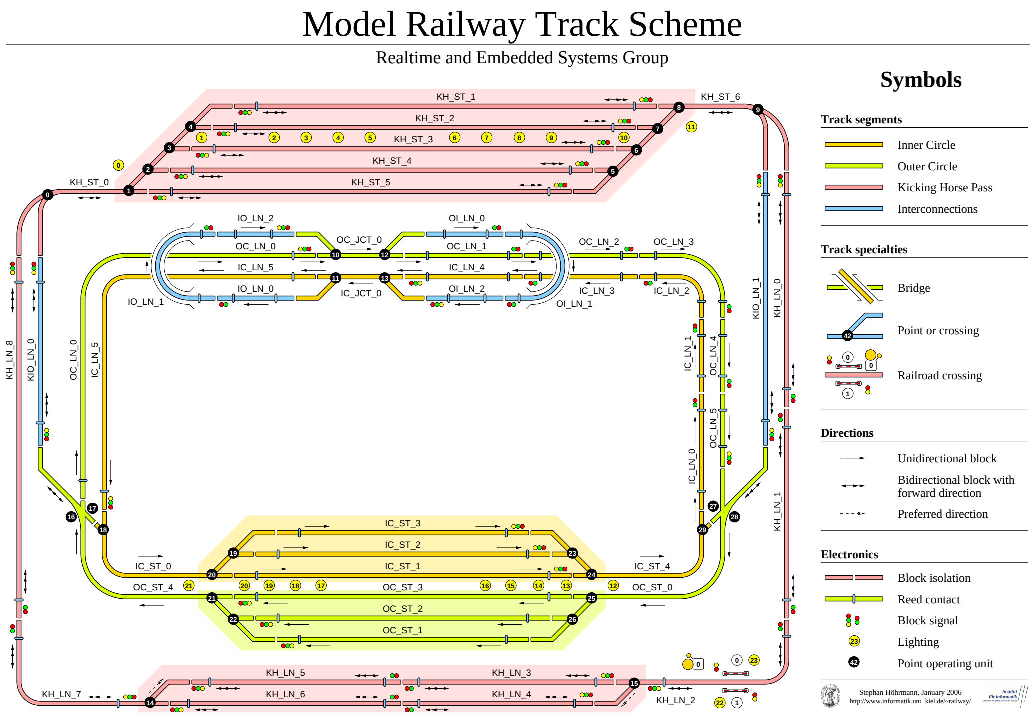


Figure A.1. The (simplified) trackscheme of the installation. Schematic taken from [HFP+06].

A.3 Program Structure

A.3.1 Blocks

Programs written in RailSL are structured into *blocks*. Blocks start with the `Start` marker and end with either an `End.` or a `Loop.` marker. The dots are part of the marker and cannot be left out.

A program consists of a single block, so the first line of any program will be `Start:` and the last line will be one of the two possible end markers. Each block must contain *at least one* statement. Statements are basically commands capable of manipulating the installation or reacting to events. The different types of statements are explained in the following sections.

When a block runs, the statements are executed one by one, in the order in which they were written. Statements that control the components of the railway installation run in practically no time, whereas some statements that wait for certain events may take a lot longer to run (see Section A.5).

Once all statements in a block have been executed, two things can happen:

1. If the block ends with the `End.` marker, the next step depends on the block's position:

   If the block is the *root* block, so the program itself, the program ends here.

   If the block is part of a `parallel` statement and was the last one to terminate, the execution is continued after the parallel statement in the parent block.

2. If the block ends with the `Loop.` marker, it is restarted and runs again from the beginning.

A.3.2 Parallel Blocks

To run multiple trains at once, it is necessary to have multiple blocks that can be executed at the same time, parallel to each other. This way, each train can have its own block that sets switch points and track segments for it.

To accomplish this, you can write `Parallel:` anywhere within a block and follow it up with *two or more complete blocks* (including start and end markers). There must not be any other statements in between these blocks. These blocks may contain any statements, including other parallel statements. After the blocks, you can just continue writing more statements. These belong to the same block as the `parallel` statement. When *all* blocks have terminated, as described in the previous subsection, the execution will resume here.

Be aware that if a `parallel` statement contains at least one block with a `Loop.` end marker, the `parallel` statement will run forever and no statements will be executed that are located after it.

Two parallel blocks cannot control the same components as this may cause not only conflicts on a software level, but also collisions on the hardware level. It is possible to model this kind of behaviour, however the compiler will later reject such models. It is usually possible to divide blocks into several chunks and have only those executed in parallel that should actually have simultaneous behaviour and put the other chunks in a second `parallel` statement later on.

A.4 Controlling the Components

When controlling the model railway installation, the following rules must be obeyed *under all circumstances* as parts of the installation might be damaged otherwise.

1. **Track Speeds**

   The speed setting of a track segment must not be reversed while a train is on it.

   If a train passes from one segment to the next, both segments must be set to the same speed and direction.

   The speed of a train must never be changed from `full` to `stop` as this may cause it to skid past the stopping point and causing collisions.

2. **Stopping**

   A train may only stop on a segment that has contacts in it and has to stop completely between the two contacts.

   Stopping on segments without any contacts is not allowed as these are either too short to accomodate a train or connector tracks that must not be blocked.

3. **Direction of travel**

   Trains must obey the travel directions for the segments stated above.

   Trains may only ever travel forward; trains driving backwards may be derailed or otherwise damaged.

   To turn a train around, the loops between Inner and Outer Circle must be used.

4. **Switch points**

   Trains must not run into switch points that have not been appropriately set

   Switch points must not change orientation while a train is crossing them

5. **Free segments**

   Whenever a block starts, it has to be ensured that all segments and switch points used by the block are free of trains.

   Each block exclusively uses the components for the entire duration of its execution.

A.4.1 Statements and their Effects

This section explains each type of statement that is available in RailSL. Generally, most statements are complete sentences. If the examples show a full stop at the end, this is part of the statement and must not be left out.

TrackStatement

The TrackStatement can be used to set speed and direction of track segments. The statement has the general form

```
Set track [segments] to [setting].
```

One or more segment can be listed according to their name in the track scheme shown in Figure A.1. When listing more than one segment, the setting is applied to all of them at once. There are three options on how to list them. They all have the same effect and you can choose whichever one you like the most.

Comma-separated KH_ST_1, KH_ST_2, KH_ST_3

Using "and" KH_ST_1, KH_ST_2 and KH_ST_3

Using the Oxford comma KH_ST_1, KH_ST_2, and KH_ST_3

There are three different speed settings available: `full`, `slow` and `stop`. If the train should move in the main direction of travel, as described above, this already is the complete setting. If the train should move *against* the main direction of travel, which is only legal on Kicking Horse Pass, add `reverse` *after* the speed setting.

Signals will be automatically set appropriately.

PointStatement

PointStatements change the orientation of switch points. The general form is

```
Set point [index / indices] to [setting].
```

As indices, the numbers next to the points in Figure A.1 are used. Again, to list multiple indices, the rules listed in the previous section apply.

There are only two settings available, `straight` and `branch`. `straight` does not necessarily mean completely straight; the `straight` setting is always the one closer to an actually straight line while `branch` is further away from it.

LightStatement

With LightStatements, the (purely decorative) lights of the installation can be turned on and off. A LightStatement looks like this

```
Turn light [index / indices] [setting].
```

For indices, the same rules as above apply. The possible settings are `on` and `off`.

CrossingStatement

With the CrossingStatement, the railway crossing outside the Kicking Horse Pass station can be opened and closed. The statement itself is very simple:

```
[command] crossing.
```

The command can either be `Open` or `Close`.

A.5 Delays

Until now, all statements already presented were *instantaneous*, meaning that they are executed immediately and take practically no time. Usually however, the program will have to wait for a certain event and do something only after this event occured.

These events can either be *timer events*, meaning that a certain amount of time elapsed, or *contact events*. As mentioned before, most of the track segments have *contacts* at either end. These contacts can detect trains passing over them. RailSL differentiates between *reaching* a contact, which means that the tip of the locomotive of a train is crossing the contact, and *passing* it, meaning that the end of the last wagon is over the contact. It is possible to wait for either of these events on any track that has contacts.

TimeWaitStatements are relatively simple:

```
Wait for [number of seconds] seconds.
```

ContactWaitStatements have the general form

```
[Event] [index] contact of [segment],
```

where the event can either be `Reach` or `Pass`, the index is one of `first` and `second` and segment is the track segment's name according to Figure A.1. Remember that the index is counted in the *main direction of travel*, which is not necessarily the way the train is currently moving!

A.6 Compilation

To compile a RailSL program, there are several steps required.

1. Open the file that you want to compile in KIELER's RailSL editor by double-clicking it from the *Project Explorer* view.
2. If it is not yet open, open the KIELER compiler selection by clicking *Window →Show View →Other...*, selecting *Other →IMB Compiler* and clicking *OK*.
3. If it is not yet open, open the KLighD Diagram view by clicking *Window →Show View →Other...*, selecting *KIELER Lightweight Diagrams →Diagram* and clicking *OK*.
4. In the compiler selection, open the dropdown menu and select *RailSL*
5. Press the *compile* button on the left of the dropdown menu
6. The Diagram view now shows C code. Click the blue *[Open in Editor]* button.
7. In the C code editor, press *Ctrl* + *A* to select everything and *Ctrl* + *C* to copy it.

The following steps depend on whether the code should run on the railway installation or be simulated locally.

Compilation for Simulation

To compile the controller for simulation, follow these steps:

1. Open the file *Controller.c* located in */railsl/simulation/RailwayControllerC*
2. Replace all possibly existing content with your new code by pressing *Ctrl* + *A* and then *Ctrl* + *V*
3. Save and close *Controller.c*
4. Open a terminal and navigate to the *simulation* directory by entering `cd ~/railsl/simulation`
5. Compile your code along with the simulation by entering `make all`
6. Start the ModelGUI in an asynchronous task by entering `gui &`
7. In the ModelGUI window, click *Open* and select the file *railway-6b1_fullsimsa.svg*, then click *Open*
8. In the terminal window, enter `Executables/SampleController`
9. In the ModelGUI window, press the *Play* button.

The simulation now shows the behaviour of your controller. If there are any problems with the simulation like collisions, error messages or unexpected behaviour in general, *do not* deploy your code to the installation. Please ask a supervisor if you cannot solve the issue yourself.

Compilation for the Installation

Before deploying your controller to the installation, please make sure that it does not cause any problems when simulated, as described in the previous section.

To deploy a controller, follow these steps:

1. Open the file *Controller.c* located in */railsl/girlsday*
2. Replace all possibly existing content with your new code by pressing *Ctrl* + *A* and then *Ctrl* + *V*
3. Save and close *Controller.c*
4. Open a terminal and navigate to the *girlsday* directory by entering `cd ~/railsl/girlsday`
5. Compile your code by entering `make all`
6. Turn on the switch labelled *12V* on the physical railway installation and wait for 30 seconds
7. Start the controller by entering `./RailwayController`

If you want to terminate a controller that is running in an infinite loop, press *Ctrl* + *C* in the terminal window.

Please rearrange all trains exactly the way they were before you used the installation. Do not push locomotives around as they may be damaged. Instead, carry them and push the carriages.

A.7 Comments and Formatting

While it is not required for the program to run, clean formatting and comments are important when writing programs.

Formatting helps understand the program more quickly. Here are some general formatting rules to keep in mind:

▻ Start a new line for every statement

▻ Do not put more than one space between words

▻ Leave empty lines where it makes sense (e. g., between blocks of a ParallelStatement)

▻ Indent your lines.

Keep everything at the same indentation that is on the same hierarchy level

For example, indent everything inside a block by one tab more than the `Start:` and `End.` markers of the block.

Use tabs instead of spaces to indent

A.8 Troubleshooting

This section lists common issues and possible solutions to them.

Editor text highlighted in red / Red “X” showing on the left

▻ Move your cursor over the highlighted word or the error marker. The tool will give you an error description that is usually enough to solve the problem.

▻ If you are unsure what the message means, delete the highlighted part and press *Ctrl* + *Space* where it used to be. You will see a menu with possible words that may be used in this position. If you are still unsure, carefully read the explanations above and make sure that the statement you are writing obeys the rules listed there.

▻ If the message says something like “Extraneous Input [X], expected EOF”, you either have an extra block end marker somewhere or you wrote code outside the main block, which is not allowed.

The Visualization is not working

▻ Reload the mapping by pressing the *Open KVis File* button and selecting the file *Mapping.kvis*, then clicking *OK*

▻ Make sure that your editor contains no input highlighted in red

▻ Make sure that the statement you are entering has an effect that should be visible in the Visualization. For example, setting a point to `straight` will not be visible as points are displayed as straight by default.

The simulation does not behave correctly

▷ Check your RailSL program for possible programming errors. Most of the time, the reason that a program does not behave as expected is a bug in the code. Especially check for contact indices; look at the visualization and make sure that the right contact is adressed.

▷ Check the *IMB Compiler* view after compiling.

Expand group labelled *SCG* by double-clicking it

Look for the *Scheduler* transformation. If there is a red box inside it, this means that your program contains two or more parallel blocks that access the same component of the installation. This may cause damages to the installation.

To solve this issue, think about splitting your program into multiple sets of blocks that can be executed one after the other to avoid this problem.

Appendix B

Sample Solutions to typical Programming Challenges

This chapter presents the typical challenges used for promotional events as well as sample RailSL programs that solve them.

B.1 Challenge 1 (Easy)

Challenge one is as follows:

1. Train starts on **KH_ST_2**
2. Train moves at full speed to the beginning of **KH_LN_6**
3. Train slows down and stops at the end of **KH_LN_6**
4. Wait for 5 seconds
5. Train continues at full speed to its original track
6. Train slows down and stops at the end of its original track.

Listing B.1 shows a simple approach to solving this challenge. For this challenge, an alternative solution is listed in Listing B.2, where each segment is only active when required and segments are turned back off as soon as they are not needed anymore. However, this is more complex and thus rarely to be found in practice.

```
/*
 * Sample solution to Girl's Day Challenge 1
 * - Simple version -
 *
 * Philip Eumann, stu121235@mail.uni-kiel.de
 */

Start:

  // Set points
  Set point 7, 8, 15 to branch.
  Set point 9 to straight.
  Wait for 3 seconds.

  // Travel to first contact of KH_LN_6
  Set track KH_ST_2, KH_ST_6, KH_LN_0, KH_LN_1, KH_LN_2, KH_LN_4, KH_LN_6 to full.
  Reach first contact of KH_LN_6.

  // Slow down
  Set track KH_LN_4, KH_LN_6 to slow.
  Reach second contact of KH_LN_6.

  // Stop and wait
  Set track KH_LN_6 to stop.
  Wait for 5 seconds.

  // Set points
  Set point 14, 0, 2, 3 to straight.
  Set point 1, 4 to branch.
  Wait for 3 seconds.

  // Travel to first contact of KH_ST_2
  Set track KH_LN_6, KH_LN_7, KH_LN_8, KH_ST_0, KH_ST_2 to full.
  Reach first contact of KH_ST_2.

  // Slow down
  Set track KH_ST_0, KH_ST_2 to slow.
  Reach second contact of KH_ST_2.

  // Stop
  Set track KH_ST_2 to stop.

End.
```

Listing B.1. A simple solution to Challenge 1.

```
/*
 * Sample solution to Girl's Day Challenge 1
 * - Advanced version -
 *
 * Philip Eumann, stu121235@mail.uni-kiel.de
 */

Start:

  // Set points
  Set point 7, 8, 15 to branch.
  Set point 9 to straight.
  Wait for 3 seconds.

  // Travel to first contact of KH_LN_6
  Set track KH_ST_2, KH_ST_6, KH_LN_0 to full.
  Pass first contact of KH_LN_0.

  Set track KH_ST_6 to stop.
  Set track KH_ST_2 to stop.
  Reach second contact of KH_LN_0.

  Set track KH_LN_1 to full.
  Pass first contact of KH_LN_1.

  Set track KH_LN_0 to stop.
  Reach second contact of KH_LN_1.

  Set track KH_LN_2 to full.
  Pass first contact of KH_LN_2.

  Set track KH_LN_1 to stop.
  Reach second contact of KH_LN_2.

  Set track KH_LN_4 to full.
  Pass first contact of KH_LN_4.

  Set track KH_LN_2 to stop.
  Reach second contact of KH_LN_4.

  Set track KH_LN_6 to full.
  Reach first contact of KH_LN_6.

  // Slow down
  Set track KH_LN_4, KH_LN_6 to slow.
  Reach second contact of KH_LN_6.

  // Stop and wait
  Set track KH_LN_4, KH_LN_6 to stop.
  Wait for 5 seconds.

  // Set points
  Set point 14, 0, 2, 3 to straight.
  Set point 1, 4 to branch.
  Wait for 3 seconds.

  // Travel to first contact of KH_ST_2
  Set track KH_LN_6, KH_LN_7 to full.
  Pass first contact of KH_LN_7.

  Set track KH_LN_6 to stop.
  Reach second contact of KH_LN_7.

  Set track KH_LN_8 to full.
  Pass first contact of KH_LN_8.

  Set track KH_LN_7 to stop.
  Reach second contact of KH_LN_7.

  Set track KH_ST_0, KH_ST_2 to full.
  Reach first contact of KH_ST_2.

  // Slow down
  Set track KH_ST_0, KH_ST_2 to slow.
  Reach second contact of KH_ST_2.

  // Stop
  Set track KH_ST_2 to stop.

End.
```

Listing B.2. An advanced solution to Challenge 1.

B.2 Challenge 2 (Advanced)

The second challenge is as follows:
This challenge requires two trains.
The first trains moves exactly as described in Section B.1.
The second train moves as follows:

1. Train starts on **IC_ST_3**
2. Train does a lap at full speed to the beginning of its original track
3. Train slows down and stops at the end of its original track.

This challenge can, for example, be solved by Listing B.3.

```
/*
 * Sample solution to Girl's Day Challenge 2
 *
 * Philip Eumann, stu121235@mail.uni-kiel.de
 */

Start:

  Parallel:

    // Train 1 - same as A1
    Start:

      // Set points
      Set point 7, 8, 15 to branch.
      Set point 9 to straight.
      Wait for 3 seconds.

      // Travel to first contact of KH_LN_6
      Set track KH_ST_2, KH_ST_6, KH_LN_0, KH_LN_1, KH_LN_2, KH_LN_4, KH_LN_6 to full.
      Reach first contact of KH_LN_6.

      // Slow down
      Set track KH_LN_4, KH_LN_6 to slow.
      Reach second contact of KH_LN_6.

      // Stop and wait
      Set track KH_LN_6 to stop.
      Wait for 5 seconds.

      // Set points
      Set point 14, 0, 2, 3 to straight.
      Set point 1, 4 to branch.
      Wait for 3 seconds.

      // Travel to first contact of KH_ST_2
      Set track KH_LN_6, KH_LN_7, KH_LN_8, KH_ST_0, KH_ST_2 to full.
      Reach first contact of KH_ST_2.

      // Slow down
      Set track KH_ST_0, KH_ST_2 to slow.
      Reach second contact of KH_ST_2.

      // Stop
      Set track KH_ST_2 to stop.

    End.

    // Train 2
    Start:

      // Set points
      Set point 24, 20 to branch.
      Set point 23, 29, 13, 11, 18, 19 to straight.
      Wait for 3 seconds.

      // Travel to first contact of IC_ST_3
      Set track IC_ST_3, IC_ST_4, IC_LN_0, IC_LN_1, IC_LN_2, IC_LN_3, IC_LN_4, IC_JCT_0,
    IC_LN_5, IC_ST_0 to full.
      Reach first contact of IC_ST_3.

      // Slow down
      Set track IC_ST_0, IC_ST_3 to slow.
      Reach second contact of IC_ST_3.

      // Stop
      Set track IC_ST_3 to stop.
    End.

End.
```

Listing B.3. A possible solution to Challenge 2.

B.3 Challenge 3 (Hard)

The third and hardest challenge is uses two trains.

Train A moves as follows:

1. Train starts on **KH_ST_2**
2. Train moves *clockwise* at full speed and does a loop of Kicking Horse Pass
3. Train returns to its original track
4. Train slows down and stops at the end of its original track.

Train B moves as follows:

1. Train starts on **KH_ST_4**
2. Train moves *counter-clockwise* at full speed and does a loop of Kicking Horse Pass
3. Train returns to its original track
4. Train slows down and stops at the end of its original track.

In this challenge, it is important to make sure that the two trains can *never* collide. Simulate the controller before running it on the installation!

This challenge was difficult to do with the formerly used tool, however a RailSL program like Listing B.4 is relatively simple to write and can solve the challenge.

```
/*
 * Sample solution to Girl's Day Challenge 3
 *
 * Philip Eumann, stu121235@mail-uni-kiel.de
 */

Start:

  Parallel:

    // Train A
    Start:

      // Set points
      Set point 7, 8, 15 to branch.
      Set point 9 to straight.
      Wait for 3 seconds.

      // Travel to first contact of KH_LN_6
      Set track KH_ST_2, KH_ST_6, KH_LN_0, KH_LN_1, KH_LN_2, KH_LN_4, KH_LN_6 to full.
      Reach first contact of KH_LN_6.

      // Slow down
      Set track KH_LN_4, KH_LN_6 to slow.
      Reach second contact of KH_LN_6.

      // Stop and wait
      Set track KH_LN_6 to stop.
    End.

    // Train B
    Start:

      // Set points
      Set point 2, 1, 14 to branch.
      Set point 0 to straight.
      Wait for 3 seconds.

      // Travel to second contact of KH_LN_3
      Set track KH_ST_4, KH_ST_0, KH_LN_8, KH_LN_7, KH_LN_5, KH_LN_3 to full reverse.
      Reach second contact of KH_LN_3.

      // Slow down
      Set track KH_LN_5, KH_LN_3 to slow reverse.
      Reach first contact of IC_ST_3.

      // Stop
      Set track IC_ST_3 to stop.
    End.

  // Both trains reached the bottom lanes.

  Parallel:

    // Train A
    Start:

      // Set points
      Set point 14, 0, 2, 3 to straight.
      Set point 1, 4 to branch.
      Wait for 3 seconds.

      // Travel to first contact of KH_ST_2
      Set track KH_LN_6, KH_LN_7, KH_LN_8, KH_ST_0, KH_ST_2 to full.
      Reach first contact of KH_ST_2.

      // Slow down
      Set track KH_ST_0, KH_ST_2 to slow.
      Reach second contact of KH_ST_2.

      // Stop
      Set track KH_ST_2 to stop.

    End.

    // Train B
    Start:

      // Set points
      Set point 8, 5 to branch.
      Set point 15, 9, 7, 6 to straight.
      Wait for 3 seconds.

      // Travel to second contact of KH_ST_4
      Set track KH_LN_3, KH_LN_2, KH_LN_1, KH_LN_0, KH_ST_6, KH_ST_4 to full reverse.
      Reach second contact of KH_ST_4.

      // Slow down
      Set track KH_ST_4, KH_ST_6 to slow reverse.
      Reach first contact of KH_ST_4.

      // Stop
      Set track KH_ST_4 to stop.
    End.
End.
```

Listing B.4. A solution proposal to Challenge 3.

Bibliography

[And95] Charles André. *Synccharts: a visual representation of reactive behaviors*. Tech. rep. Oct. 1995.

[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.

[Ber00a] Gérard Berry. *The Esterel v5 language primer, version v5_91*. `ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf`. Centre de Mathématiques Appliquées Ecole des Mines and INRIA. 06565 Sophia-Antipolis, 2000.

[Ber00b] Gérard Berry. "The foundations of Esterel". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.

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

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: an annotated bibliography". In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: `10.1145/352029.352035`. URL: `http://doi.acm.org/10.1145/352029.352035`.

[Don10] Aron Donohue. "Debugging domain-specific languages". `http://www.arandonohue.com/writing/MSc/ut-thesis.pdf`. Master thesis. University of Toronto, Graduate Department of Computer Science, 2010.

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

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

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

[HFP+06] Stephan Höhrmann, Hauke Fuhrmann, Steffen Prochnow, and Reinhard von Hanxleden. "A versatile demonstrator for distributed real-time systems: using a model-railway in education". In: *Proceedings of the ERCIM/DECOS Dependable Embedded Systems Workshop at Euromicro 2006*. Cavtat/Dubrovnik, Croatia, Aug. 2006.

[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.

[Höh06] Stephan Höhrmann. "Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sho-dt.pdf. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mar. 2006.

[McC88] John W. McCormick. "Using a model railroad to teach digital process control". In: *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '88. Atlanta, Georgia, USA: ACM, 1988, pp. 304–308. ISBN: 0-89791-256-X. DOI: 10.1145/52964.53039. URL: http://doi.acm.org/10.1145/52964.53039.

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

[NSI+03] H. Noma, H. Sasamoto, Y. Itoh, Y. Kitamura, F. Kishino, and N. Tetsutani. "Computer learning system for pre-school-age children based on a haptized model railway". In: *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.* Jan. 2003, pp. 118–119. DOI: 10.1109/C5.2003.1222343.

[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.

[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.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[Sta15] Andreas Stange. "Comfortable SCCharts modeling for embedded systems". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.

[The06] The model railway. *Project homepage*. Group of Real-Time and Embedded Systems, Department of Computer Science, Kiel, Germany. 2006. URL: http://www.informatik.uni-kiel.de/~railway.

[Vel12] M. Veldthuis. "Quby - a domain-specific language for non-programmers". http://www.cs.rug.nl/~aiellom/tesi/veldthuis.pdf. Master thesis. University of Groningen, faculty of mathematics and natural sciences, Aug. 2012.

[Wec15] Nis B. Wechselberg. "Model railway 4.0". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbw-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Mar. 2015.