INSTITUT FÜR INFORMATIK

SCCharts: The Railway Project Report

Steven Smyth, Christian Motika,
Alexander Schulz-Rosengarten,
Nis Boerge Wechselberg, Carsten Sprung, and
Reinhard von Hanxleden

Bericht Nr. 1510
August 2015
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

SCCharts: The Railway Project Report

Steven Smyth, Christian Motika,
Alexander Schulz-Rosengarten, Nis Boerge Wechselberg,
Carsten Sprung, and Reinhard von Hanxleden

Bericht Nr. 1510
August 2015
ISSN 2192-6247

E-mail: {ssm, cmot, als, nbw, csp, rvh}@informatik.uni-kiel.de

Technical Report

Contents

1. Introduction **1**
- 1.1. Safety-Critical Systems . . . 1
- 1.2. The Model Railway . . . 1
- 1.3. SCCharts . . . 2
- 1.4. SCCharts Railway Controller . . . 2
- 1.5. Contributions . . . 3
- 1.6. Outline . . . 3

2. SCCharts Extensions **4**
- 2.1. KIELER Compiler . . . 5
  - 2.1.1. New SCCharts Features . . . 6
- 2.2. Declarations . . . 6
  - 2.2.1. Constants . . . 7
  - 2.2.2. Arrays . . . 8
  - 2.2.3. Extern . . . 9
- 2.3. Referenced SCCharts . . . 9
  - 2.3.1. Implementation Details . . . 12
- 2.4. Hostcode . . . 14
  - 2.4.1. Function Calls . . . 15

3. The Railway Project **16**
- 3.1. The Model Railway Installation . . . 17
  - 3.1.1. Railway Hardware . . . 17
  - 3.1.2. Programming Interface . . . 18
- 3.2. The Overall Approach . . . 20
  - 3.2.1. Station-2-Station Controllers . . . 20
  - 3.2.2. Train Controller . . . 23
  - 3.2.3. Mutual Exclusion Controllers . . . 24
  - 3.2.4. Complete Railway Controller . . . 24
- 3.3. Controller Implementation . . . 25
  - 3.3.1. Components . . . 25
  - 3.3.2. TCP Communication . . . 27
  - 3.3.3. GUIs . . . 29
- 3.4. Project Workflow . . . 29

4. Validation and Experimental Results **31**
4.1. Source-Code Size . . . 31
4.1.1. Compilation . . . 32
4.1.2. SCCharts Compilation . . . 33
4.2. Compiler Performance . . . 35
4.2.1. Remarks on Compiler Performance . . . 41
4.3. Tick Function Performance . . . 43

5. Survey Results **44**
5.1. Survey Setup . . . 44
5.1.1. Future Work . . . 44
5.1.2. Participants . . . 44
5.2. Language Aspects . . . 45
5.2.1. Deterministic Behavior . . . 45
5.2.2. Programming Paradigms . . . 46
5.2.3. Problem Solving . . . 47
5.2.4. Language Difficulty . . . 48
5.2.5. Modularity . . . 49
5.2.6. Project Revisions . . . 50
5.3. SCCharts features . . . 51
5.4. Modeling Aspects . . . 53

6. Related Work **55**
6.1. Model Railway Installation . . . 55
6.2. Model Railway Project 2007 . . . 56
6.3. Model Railway Project 2012 . . . 59

7. Wrap-Up **62**

Bibliography **63**

A. SCCharts Survey **65**

B. Raw Controller Model Analysis Results **79**

List of Figures

1.1. Reactive system cyclic control loop 2

2.1. SCCharts syntax overview 4
2.2. SCCharts Transformation Dependencies for Extended SCCharts Features 5
2.3. Example of the declaration and usage of arrays in SCCharts 8
2.4. Clock example: delay 10
2.5. Clock example: emitter 10
2.6. Clock example: clock 11
2.7. Clock example: fully expanded version of clock 12

3.1. Model railway installation 16
3.2. Model railway scheme 17
3.3. SCChart railway controller hierarchy 20
3.4. Railway stations and circles 21
3.5. Station-2-Station controller (without declarations and transition labels) . 22
3.6. Dynamic train controller (without declarations and transition labels) . . 23
3.7. Mutual exclusion controller (simplified) 24
3.8. Railway controller components 26
3.9. Example views from the GUIs 30
3.10. Project workflow 30

4.1. Distribution of self-written code in LoC 31
4.2. Distribution of the complete C code before compilation 32
4.3. Number of model elements during compilation 34
4.4. Compile time for rail regression test models 36
4.5. Code size for rail regression test models 36
4.6. Normalized compile time for rail regression test models 37
4.7. Compile time for rail regression test models and final dynamic SCCharts controller 37
4.8. Code size for rail regression test models and final dynamic SCCharts controller 39
4.9. Normalized compile time for rail regression test models and final dynamic SCCharts controller 39
4.10. Compile time for rail regression test models and final dynamic SCCharts controller (except June compiler version) 40
4.11. Normalized compile time for rail regression test models and final dynamic SCCharts controller (except June compiler version) 40

5.1. Language preferences 45
5.2. Deterministic behavior 45
5.3. Programming paradigms 46
5.4. Problem solving 47
5.5. Language difficulty 48
5.6. Modularity 49
5.7. Project revisions 51
5.8. SCCharts feature importance poll 52
5.9. Tools quality 53
5.10. Modeling aspects 53

6.1. Deadlock-free model of pass lane with turnout track 55
6.2. Net model of the kicking horse pass track 56
6.3. Railway project 2007 hardware schematics [from project website] 57
6.4. Railway project 2007 tool chain [from project website] 57
6.5. Railway project 2007 SCADE model part [from project website] 58
6.6. Railway project 2007 SCADE model part [from project website] 59
6.7. Controller communication from the railway project 2012 61
6.8. Segment scheme from the railway project 2012 61

Abstract

SCCharts is a visual language proposed in 2012 for specifying safety-critical reactive systems. We present the results of the first medium sized SCCharts case-study. The case-study was conducted in the context of the railway project performed by students at the Kiel University in the summer term 2014. The railway project is a regularly occurring student training project that teaches principles of concurrent *cyber-physical systems* on a complex live model railway demonstrator.

This report presents details of the first medium size SCCharts models created during the project. We explain what additional language extensions to SCCharts were necessary and how they were implemented. To handle performance issues that arose while developing the controller, compiler enhancements became necessary and are evaluated here. Furthermore, the participants completed a survey at the end of the project to confirm the goals that the SCCharts language and our SCCharts tool chain are suitable to build complex controllers. In the survey, the participants compared both, the SCCharts language and our SCCharts tools, with other modeling and classical programming languages and tools.

1. Introduction

1.1. Safety-Critical Systems

Reactive systems are computer systems that compute a function in a cyclic fashion. They continuously interact with their environment and react to inputs with computed outputs. Figure 1.1 visualizes a reactive system that repeatedly reads inputs from the environment, computes a reaction on the basis of the given inputs, and writes the reaction, i. e., the outputs to the environment. Therefore, the reactive system reacts with computed outputs, i. e., the reaction, to given inputs. We call one reaction computation a *tick*. Additionally reactive systems have to fulfill their computations in a certain amount of time, which is typically a requirement derived from (physical) properties of their environment. Thus, reactive systems often deal with deadlines, which makes them real-time systems. Reactive systems typically control their environment. Usually they are also embedded into their environment and hence termed *embedded systems*. Furthermore, reactive systems often control environments whose overall correct behavior is crucial. Failing in the sense of reactive systems not only means computing an incorrect output reaction but also not meeting the correct timing constraints, also called *deadlines*. The failure of such systems or parts of it may results in loss of human life or a financial disaster. It is essential that these systems do not fail and guarantee safety . Hence, many real-time and embedded systems also are *safety-critical systems*.

1.2. The Model Railway

The model railway[1] of the Department of Computer Science at Kiel University is an embedded system in the sense that the computer system which controls the trains is embedded and not visible from the top. The model railway is a reactive system in the sense that the controller controls the trains in a cyclic fashion. For each control cycle, it reads sensor information about the position of trains, then computes which tracks are set to which speeds and which switch points are set to straight or branch position. The model railway system is a real-time system because the dynamics of the trains imposes timing requirements/deadlines on the time for computation a reaction. The model railway further is good example for a safety-critical system because crashes of trains or electrical shorts must be avoided under any circumstance.

[1] `http://www.informatik.uni-kiel.de/~railway/`

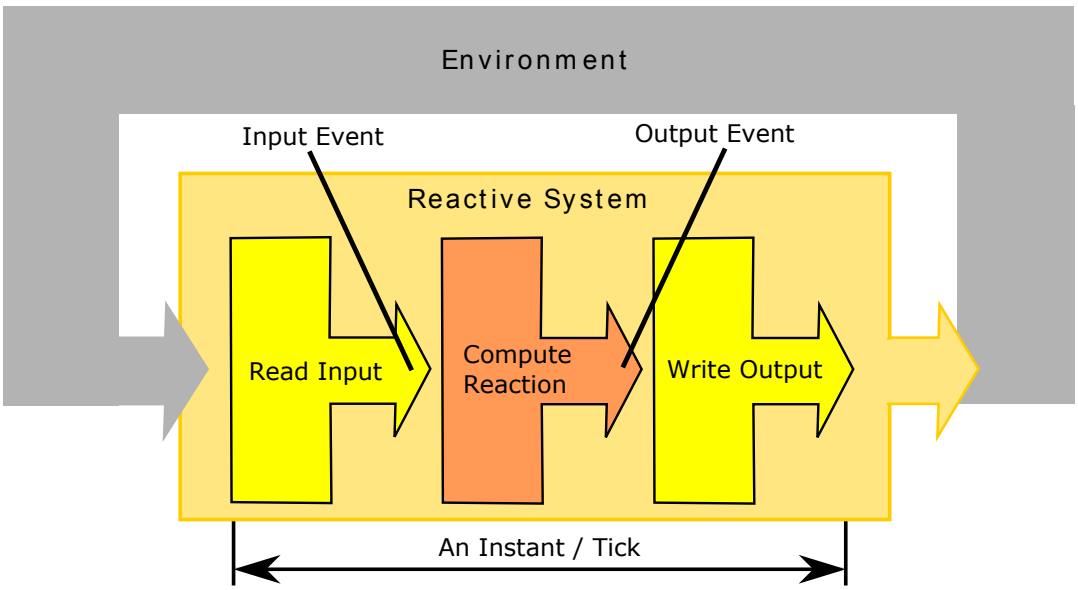


Figure 1.1.: Reactive system cyclic control loop [MvH14]

1.3. SCCharts

The recently proposed *Sequentially Constructive Statecharts* (*SCCharts*) are a synchronous modeling language for reactive systems [vHDM+14]. SCCharts are based on the sequentially constructive model of computation (SC MoC) [vHMA+13] and have been designed with a special emphasis on safety-critical systems where determinism and predictability play crucial roles.

An interactive and incremental design and modeling process has been proposed that includes an incremental compilation approach. In this Single-Pass Language-Driven Incremental Compilation (SLIC) approach SCCharts can be compiled interactively and incrementally in a single pass based on a well-defined sequence of model transformations down to C code. The interactive nature of the compilation lets the user chose from numerous model transformation options. Interactive compilation enables the modeler to understand not only the language features itself but also how each feature influences his specific model semantically. This helps in building more robust and reliable models which is crucial for safety-critical systems. [MSvH14]

1.4. SCCharts Railway Controller

The model railway installation contains 127 metres of tracks which can be controlled concurrently including switch points, signals, lights and a track crossing. The system is operated via a network of PC104 computers or TTP power nodes and is accessed via Ethernet, TTP or CAN bus. The details of the railway system are illustrated in Chapter 3.

In the summer term 2014 the participants of the railway project course[2] were given the task to design and implement a railway controller in SCCharts. The controller should manage the train schedules and should be able to control up to eleven trains simultaneously utilizing the underlying hardware. Testing the suitability of SCCharts to create a sophisticated controller of this size was also part of the goal.

1.5. Contributions

We present the results of the SCCharts case-study in the context of the railway project performed by students at the Kiel University in the summer term 2014. Specifically,

- we list which SCCharts language extensions were necessary to achieve the set project goals. Therefore we explain why this improvements were mandatory and how they were implemented.
- Secondly we present the first mid-to-large scale SCChart models and depict differences in the creation and debugging of these models in comparison to smaller models.
- We present the fully functional and remote controllable railway controller modelled in SCCharts.
- Furthermore, we analyse the suitability of SCCharts for large team projects with emphasis on composability, component testing, overall project overview and the performance of the compiler tool chain.

1.6. Outline

Section 2 describes which features were considered helpful to finish the project successfully. It elaborates which of the features were implemented how and why we choose these features. Section 3 starts with a more detailed introduction of the railway installation and the general set-up of the railway project. In continues with the project goals that were set by the whole team and explains the approach the students have taken. The experience that could be gathered throughout the project is then analysed and rated in Section 4. Furthermore, the results of a conducted survey amongst the projects participants comparing the SCCharts language and tooling with other popular languages is presented in Section 5. Subsequently, Section 6 draws parallels to related work and different approaches. The report concludes with a wrap-up of the case-study in Section 7.

[2] `http://rtsys.informatik.uni-kiel.de/confluence/display/SS14Railway`

2. SCCharts Extensions

The SCCharts language is divided in two main categories. *Core SCCharts* contain the key ingredients of concurrent statecharts, namely concurrency and hierarchy, whereas *Extended SCCharts* add syntactic sugar to the language (e. g., history, aborts, actions). Figure 2.1 depicts all available SCCharts features as they were present at the beginning of the railway project. The upper part contains the core features, whereas the extended elements situate in the lower part.

A benefit of this language partitioning is that it is sufficient to ground the theoretical foundations of SCCharts on the minimal Core SCCharts sub-language, because every Extended SCChart can be transformed into a semantically equivalent Core SCChart [vHDM+14]. Therefore most of the new SCCharts features introduced in this section try to avoid any alterations of the core language. As explained elsewhere [MSvH14] this is particularly useful because we only need a transformation rule to handle a new feature and without compromising the correctness of the subsequent tool chain.

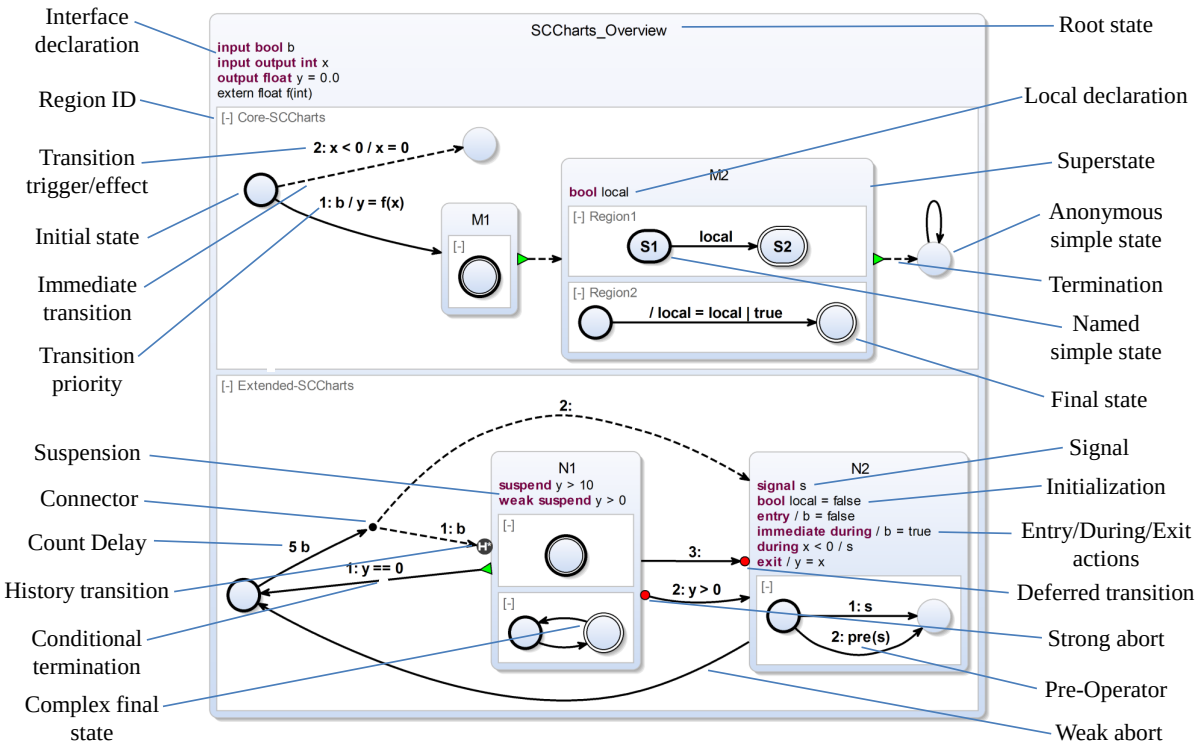


Figure 2.1.: SCCharts syntax overview [MSvH14]

2.1. KIELER Compiler

The SCCharts compilation implemented in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is based on model transformations as shown in Figure 2.2. These transformations have dependencies to each other that may either be produced or not-handled-by dependencies. The SLIC approach is based on these dependencies and explained in detail elsewhere [MSvH14]. A sequence of model transformations is derived from these dependencies where each transformation rule is applied just once for a model. Sometimes there may be more than one model transformation available with different optimization goals. The interactive fashion of the KIELER Compiler (KiCo) allows to select certain model transformation options or even single transformations only to inspect and optimize intermediate results. Because intermediate results are valid models they can be fully inspected and understood by the modeler. This helps the modeler in building reliable models because the modeler is able understand not only the language features itself but also what each usage of a language feature semantically means for his or her concrete model. As we learned from this project, it also helps the compiler writer to build a reliable compiler because he or she is able to inspect intermediate results where just certain model transformations were selected to be applied. Building reliable models and having a reliable compiler both is crucial for developing safety critical systems.

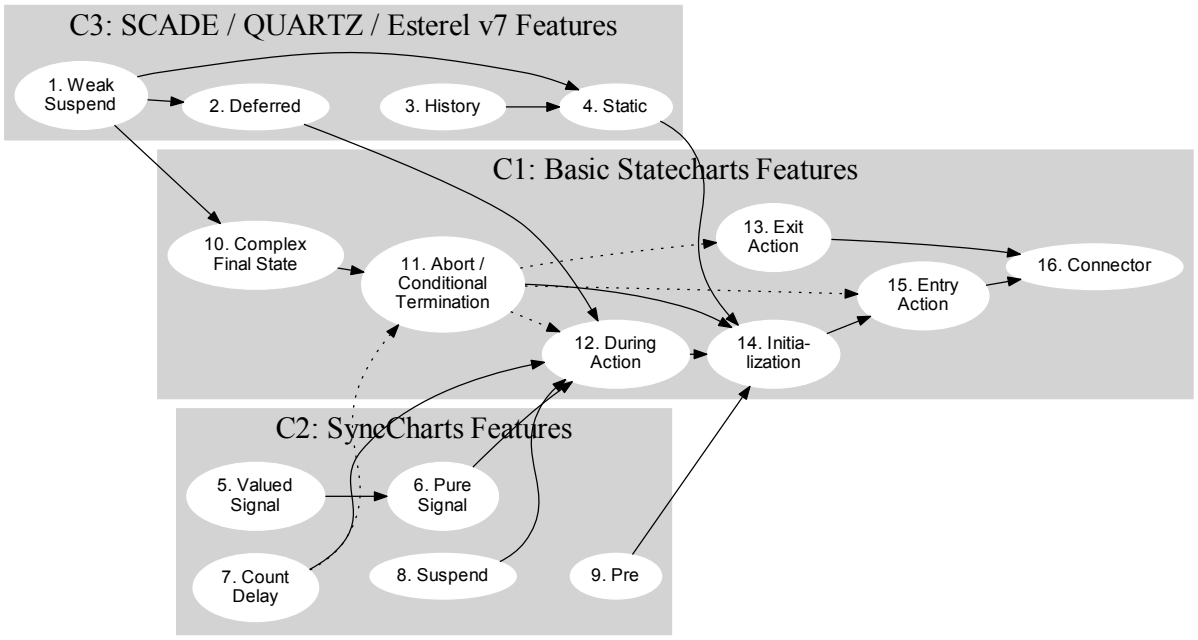


Figure 2.2.: SCCharts transformation dependencies for Extended SCCharts features [MSvH14]

2.1.1. New SCCharts Features

In preparation for the railway project course we proposed a list of new language features at the Oberseminar in March 2014[1]. The list contained five features we considered mandatory for SCCharts being used as the main tool to build the railway controller, namely *constants*, *arrays*, *referenced SCCharts*, aggregative functions such as *map*, and extensions on the hostcode interface. As optional improvement we also considered *structs* as an extension to the type system.

A short survey conducted amongst the project participants subsequent to the initial tutorials of the course resulted in a similar list of desired features. A module concept, now known as referenced SCCharts, was considered most important. Secondly, more sophisticated functions in the development tools were desired. The KIELER compiler interface needed a possibility to persist intermediate transformation results as well as the generation of the final code in one step. Also, more meaningful error messages were desired. Furthermore, the participants also determined arrays as mandatory.

This section describes all new SCCharts features in detail grouped by category. First, all extensions to the declarations are explained. Then, the module concept of referenced SCCharts is elucidated and subsequently the section closes with the hostcode additions.

As most transformation rules in the KIELER SCCharts implementation are realized in XTend[2] [vHDM+13], all new transformations are also written in XTend.

2.2. Declarations

Each *scope*, which is either a *superstate* or a *region*, of SCCharts may include a declaration part. However, to emphasize the communication interface with the environment, the declaration of the outer-most state, the *root state*, is usually called *interface declaration*.

A central term in the context of declarations in the underlying KIELER expression language is a *Valued Object* (VO). VOs identify variables and all objects that derive from these (e. g., signals). Therefore, a specific value in the expression language used in the actual SCCharts implementation can either be a *literal* (e. g., true) or a VO.

Contrary to the first SCCharts implementation, VOs are now contained in *declaration groups* which hold the particular attributes of the included objects. They hold a list of VOs, their type and possible one or more of the keywords *input*, *output*, *signal*, *static*, *const*, and *extern*, with the latter two being new features. Therefore, variables may now be declared with an common type but still with individual *initialization part*. For example, Listing 2.2 is now a perfectly valid interface declaration.

The three main additions to the declarations, namely constants, arrays, and external declarations, are explained in the following sub-sections.

[1] http://www.informatik.uni-kiel.de/rtsys/teaching/ws13/osem1/
[2] http://eclipse.org/xtend/

Listing 2.1: Old declaration example

```
1 input bool I1;
2 input bool I;
3 output int O1 = 0;
4 output int O2 = 1;
```

Listing 2.2: New declaration example

```
1 input bool I1, I2;
2 output int O1 = 0, O2 = 1;
```

2.2.1. Constants

To avoid *magic numbers*, particularly in team projects, constants are mandatory. Moreover, they do not impair the subsequent analyses in the compile chain because the valued object is invariable. Additionally, they enable the modeler to use meaningful identifiers.

A VO is marked as constant if its declaration contains the keyword const. A constant VO must have a initialization part as constant value for the VO. Listing 2.3 depicts a constant definition of a VO.

Listing 2.3: Const declaration example

```
1 const float PI = 3.1415;
```

In the KIELER implementation, a single transformation rule transforms all constant VOs into their literals. Since the constant declaration makes use of the initialization part, following the SLIC approach the rule must be executed before the initialization rule resolves these parts. Although we suggest to invoke the const rule just before the initialization rule to maintain readability as long as possible during the SLIC, any point of time before the initialization part should suffice.

The transformation is depicted in simplified form in Listing 2.4. The transformConst method is called once for each scope. In lines 2-4 all const objects in this shallow scope are filtered and returned as list. Subsequently, in the loop following line 5 the VOs are replaced by their initialization value. Therefore, replaceAllReferencesWithCopy finds all *valued object references* in the scope, including nested references, and replaces them. Eventually the const VO is deleted in line 8.

Listing 2.4: Const transformation rule

```
1  def void transformConst(Scope scope) {
2    val constObjects = scope.valuedObjects.filter[
3      isConst && initialValue != null
4    ].toList
5    for (const : constObjects) {
6      val replacement = const.initialValue
7      scope.replaceAllReferencesWithCopy(const, replacement)
8      const.delete
9    }
10 }
```

As the railway project team depends heavily on hostcode because of the tight binding to the C API, we decided to provide a possibility to use const VOs in plain hostcode descriptions. Hence, the rule was also extended by an annotation to perform a straightforward text replacement (not depicted in the listing). A const declaration annotated with @alterHostcode also replaces textual occurrences of the VO name in hostcode. Since this annotation extension is considered a work-around for the current project, usage is not endorsed as it may be deactivated in future versions.

2.2.2. Arrays

To handle large amounts of data, arrays (or at least some kind of aggregate concept) are essential. We decided in favour of arrays because this enabled us to directly support the data format of the underlying host language (in our case C) and the data structures of the railway API. More on the API will follow in Section 3.

The dependency analysis must be aware of different cardinalities and accesses on array types. Therefore, if an array field is accessed with a literal (or a const VO) the dependency analysis is able to determine a potential conflicting concurrent *variable access*. If the field is accessed in a dynamic manner, the whole access is seen as scalar and distinction of the unique array cells is not possible statically. The characterization of different variable accesses is explained elsewhere [vHMA+13].

Hence, with exception of the extension of the dependency analysis, the implementation of arrays is mainly a question of grammar modification. Besides the additions to the expressions grammar, the SCCharts meta model had to be modified. The grammar rule for VOs in the SCCharts Textual Language (SCT) is depicted in Listing 2.5. After the name of the object the grammar in line 3 allows arbitrary many array cardinalities of type int for multi-dimensional arrays. Nevertheless, the definition of these cardinalities is optional. If they are omitted, the VO will stay a scalar.

Same is true for the SCT action grammar, which is responsible for the trigger and effect handling of transitions. Here, assignments to variables must also be able to write to single array elements. The index of such element may be an arbitrary expression and is not restricted to a literal. The corresponding grammar rule is shown in Listing 2.6. General expressions and therefore also triggers are handled by the underlying KIELER expressions and do not need SCCharts-specific adjustments. An example of array declaration and usage in SCCharts can be seen in Figure 2.3.

Although it is entirely possible to manage arrays on extended SCCharts level and let a transformation handle the conversion of all array objects into scalar VOs, we decided against this approach and implemented arrays as core extension embedded in the underlying expression language. Therefore, arrays are also available in the low-level syntheses of the tool chain (e. g., dependency analysis, guard generation in the circuit approach, etc.). Furthermore, the array declaration can be copied nearly one-to-one to the host language and named as defined in the host code.

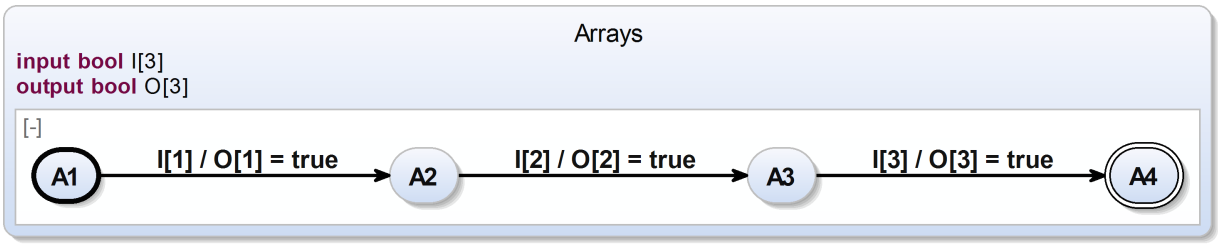


Figure 2.3.: Example of the declaration and usage of arrays in SCCharts

Listing 2.5: SCT VO grammar rule

```
ValuedObject returns kexpressions::ValuedObject:
    name=ID
    ('[' cardinalities+=INT ']')*
    ('=' initialValue=Expression)?
    ('combine' combineOperator = CombineOperator)?;
```

Listing 2.6: SCT assignment grammar rule

```
Assignment returns sccharts::Assignment:
    valuedObject=[kexpressions::ValuedObject]
    ('[' indices+=Expression ']')*
     "=" expression = Expression ;
```

2.2.3. Extern

One useful mechanism when working tightly with an external API is the possibility to tell the compiler not to care about the generation of a certain VO but still let it know that it is present. Roughly speaking, this is what the extern keyword tells the compiler. The VO is treated as any other VO and accessible in all analyses but ignored in the final code generation step and not generated automatically. Instead, the programmer himself is able to define it in a manually written code part and can use it in the generated code of the SCCharts model.

External VOs are introduced by the extern keyword and are allowed to be combined with any other combination of declaration keywords. They are not obliged to have a type. An example of a valid external declaration is depicted in Listing 2.7.

Listing 2.7: Extern declaration example

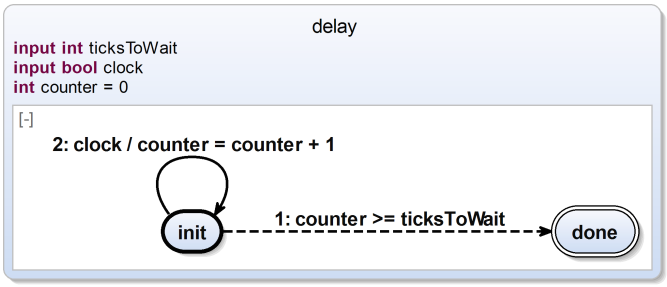
```
extern railway_system;
```

Even though not much is known about the railway_sytem in the example, the compiler can still use it (e. g., in other hostcode calls), run analyses on it (e. g., dependency analysis), and serialize it in the final code generation step.

2.3. Referenced SCCharts

Arguably the addition with the biggest impact is the *referenced SCCharts* feature, the newly introduced modularity concept of SCCharts. Organize large models, divide projects in smaller sub-projects, share work between a team are all examples that make a module functionality indispensable.

Referenced SCCharts are similar to the referencing mechanism provided by SyncCharts [And96] or Esterel [Ber00]. The following clock example will explain the different aspects of this feature. The first model delay (Figure 2.4) will wait a certain amount of clock ticks before it proceeds. The number of ticks to wait in ticksToWait and the clock signal clock are both coming from the environment. Hence, we can adjust the waiting time and the frequency of the tick interval from outside the model.



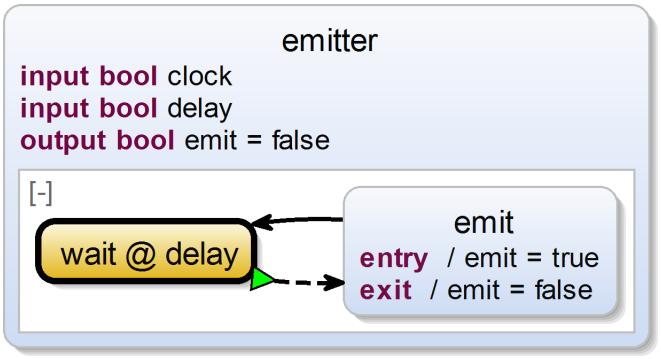

```
1  scchart delay {
2    input int ticksToWait;
3    input bool clock;
4    int counter = 0;
5
6    initial state init
7    --> done immediate with
8      counter >= ticksToWait
9    --> init with
10     clock / counter = counter + 1;
11
12   final state done;
13 }
```

Figure 2.4.: Clock example: delay

In the beginning counter is initialized with zero and the model starts at the init state. If the counter value reaches ticksToWait, the done states becomes active and the SCChart terminates. Otherwise, the init state stays active and if a clock signal occurs, the counter is incremented.

A second model emitter (Figure 2.5) should wait a specific amount of clock ticks and emit a signal for one tick if the waiting time has passed. Therefore, emitter can make use of delay. As usual we can declare a new state in line 6 but let it point to another SCChart with the keyword references in line 7. After a reference target SCChart has been determined, one *binds* the interface declaration of that SCChart to variables in the actual scope. In line 8 emitter binds the clock of delay to the local clock variable and in line 9 delay's ticksToWait is bound to the local object metrum. Apart from the link to the referenced SCChart the state behaves like a normal *superstate*. Hence, once the delay state finishes, the emit state becomes active for one tick and emits the emit vairable. Eventually, the cycle starts over again.

Although the clock to clock binding is done explicitly and recommended for clarity, a binding of objects with the same name is not mandatory. The reference transformation



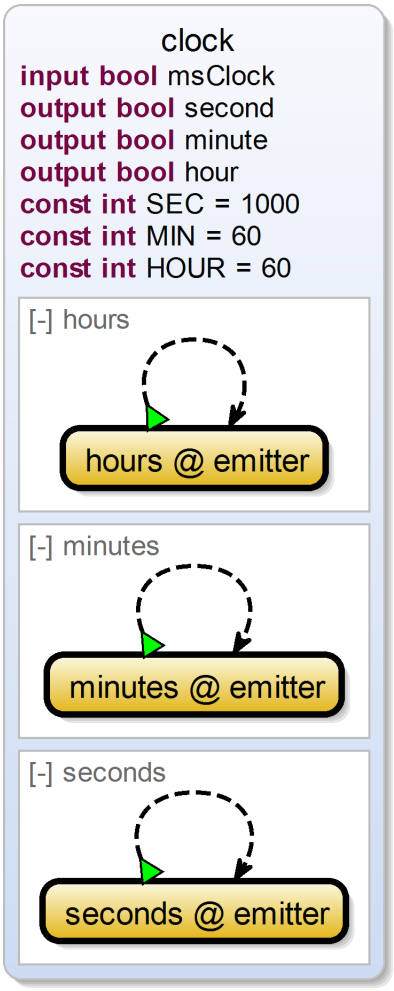

```
1  scchart emitter {
2    input bool clock;
3    input bool delay;
4    output bool emit = false;
5
6    initial state wait
7    references delay
8    bind clock to clock,
9         ticksToWait to delay
10   >-> emit;
11
12   state emit {
13     entry / emit = true;
14     exit / emit = false;
15   }
16   --> wait;
17 }
```

Figure 2.5.: Clock example: emitter

will search for valid objects and generates an implicit binding if the explicit binding is omitted.

Finally, a third model clock (Figure 2.6) simulates a clock. It receives a millisecond signal in msClock as input and sends signals for seconds, minutes, and hours itself. Therefore it comprises three concurrent regions and uses the emitter model to accomplish the task. In the region seconds the clock of the emitter is bound to the millisecond clock, the metrum is bound to the necessary interval for one second stored in the const SEC, and the emit variable should be second. In a similar manner the region minutes handles the signal emission minute. Here, the clock is linked to the generated seconds impulses and the metrum will be constant MIN. The same is true for region hours w.r.t. minutes using minute and constant HOUR to bind the emitter.

Hence, the SCChart clock uses the emitter model three times, which itself uses the delay model. The fully expanded version of clock is depicted in Figure 2.7. After the expansion the transformation order proceeds as usual.




```
1  scchart clock {
2    input bool msClock;
3    output bool second;
4    output bool minute;
5    output bool hour;
6    const int SEC = 1000;
7    const int MIN = 60;
8    const int HOUR = 60;
9
10   region seconds:
11
12   initial state seconds
13   references emitter
14   bind clock to msClock,
15        metrum to SEC,
16        emit to second
17   >-> seconds;
18
19
20   region minutes:
21
22   initial state minutes
23   references emitter
24   bind clock to second,
25        metrum to MIN,
26        emit to minute
27   >-> minutes;
28
29
30   region hours:
31
32   initial state hours
33   references emitter
34   bind clock to minute,
35        metrum to HOUR,
36        emit to hour
37   >-> hours;
38 }
```

Figure 2.6.: Clock example: clock

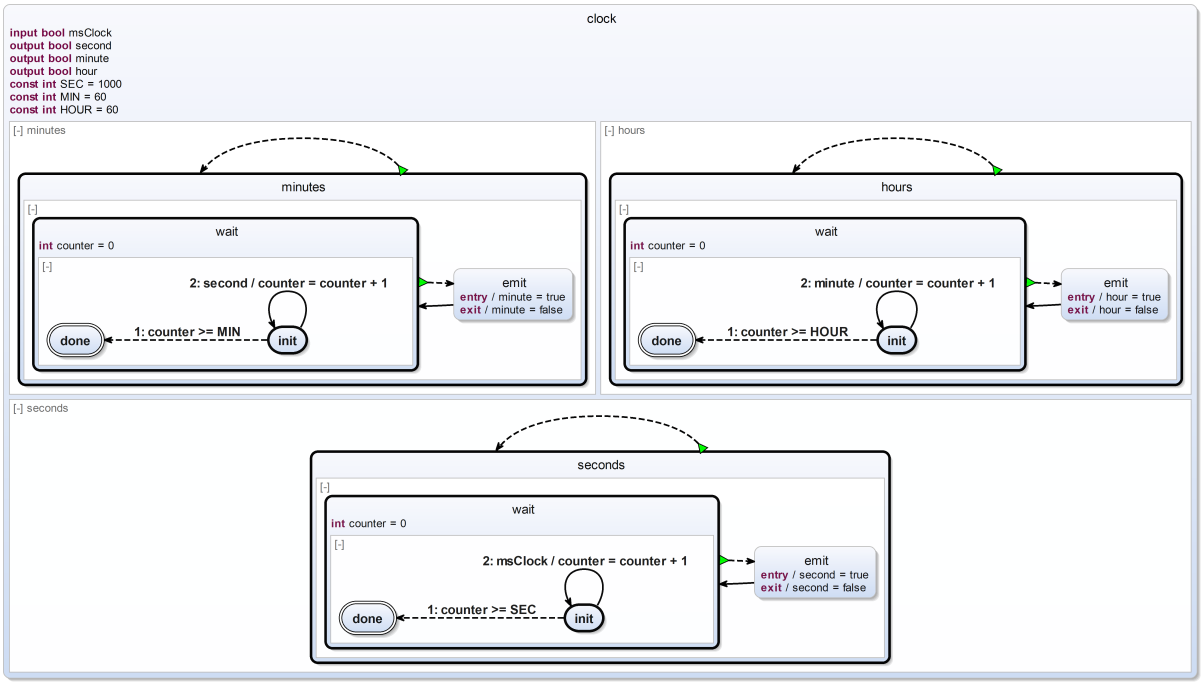


Figure 2.7.: Clock example: fully expanded version of clock

Since a link to a referenced SCChart results in the expansion of the target SCChart into the actual model and therefore in an inclusion of arbitrary (extended) SCCharts features, it is mandatory that the referenced SCCharts transformation is executed before any other transformation is invoked. This procedure follows the SLIC approach mentioned in Section 1.

2.3.1. Implementation Details

As can be seen in the listings of the clock example models emitter in Figure 2.5 and clock in Figure 2.6, a state may reference another SCChart model with the keyword references. With help of the Xtext[3] framework the content assist will pop up suggestions for valid SCCharts if asked. If any interface variables of the referenced model should be bound to local VOs, the keyword bind provides the possibility to add a list of *bindings*.

Listing 2.8: SCT grammar binding rule

```
1 Binding returns sccharts::Binding:
2     (annotations+=Annotation)*
3     formal = [kexpressions::ValuedObject|ID] 'to'
4     actual = [kexpressions::ValuedObject|ID];
```

In the grammar (Listing 2.8) a binding is simply a connection from one VO to another VO. The *formal* VO of the interface of the referenced model will be bound to the *actual* VO in the model. Since the XText grammar cannot distinguish between the VOs of the

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

two different models, the *SCT scope provider*[4] is responsible for resolving the different scopes accordingly. One may notice that the rule also allows specific annotations for each binding. With the binding rule formulated it is possible to extend the original state rule of the SCCharts grammar (Listing 2.9).

Listing 2.9: SCT grammar state rule

```
1  State returns sccharts::State:
2      (annotations += Annotation)*
3      (initial?='initial') (final?='final')
4      (type=StateType)? ('state') (id=ID) (label=STRING)?
5      (
6          ('references' referencedScope = [sccharts::State|ID]
7          ('bind' bindings += Binding (',' bindings += Binding)*)?
8      ) | ('{'
9              (declarations+=Declaration | localActions+=LocalAction)*
10             ((regions+=SingleRegion)(regions+=Region)*)?
11         '}')
12     )?
13     (outgoingTransitions+=Transition)* ';';
```

Line 6 and 7 define the reference extension. In the actual implementation the referencedScope may be another state identified by its ID. However, it might be possible to extend this to regions as well in future versions. This referenced link is mandatory for referenced SCCharts. The optional binding list follows. Different bindings are separated by comma.

The transformation rule is depicted in Listing 2.10. In lines 2 - 10 the function invokes the copyState of the *SCChartsExtension* to generate a copy of the referenced state. All nested references are resolved first to ensure that the chain of VO bindings stays intact. Then, it retains the local id and label and adds the state to the parent region. The copy is called newState.

In lines 13 - 19 each object in the new state that is bindable will be checked. If it is bound to a local VO it will be replaced. The remaining VO declarations will be removed in lines 21 - 26. However, this will not guarantee that all interface declarations are dealt with. Therefore, the next code block (lines 28 - 36) will search for any remaining declarations in the copied state that are marked as input or output. Then, line 32, the method tries to find a suitable VO in the local model. This resembles the implicit binding mentioned earlier in Section 2.3. Each VO of the interface declaration that was not bound explicitly may be bound implicitly if a corresponding VO can be found. Suitable VOs are determined by ID. Hence, we recommend an explicit binding if possible because IDs in different models may be equal and an implicit binding may be unintended. The interface declaration will be removed after if has been processed.

Subsequently, the incoming and outgoing transitions to the original state will be redirected to the new state in lines 38 - 39 and newState retrieves the initial and final flags of the referencing state. Eventually, the original state that referenced the external model is deleted. At this point the referenced state is fully embedded in the local model and replaced the referencing state.

[4] in de.cau.cs.kieler.sccharts.text in KIELER

Listing 2.10: Referenced SCCharts transformation

```
def void transformReference(State state, State targetRootState) {
  val newState = (state.referencedScope as State).copyState => [
    allContainedStates.filter[ referencedState ].immutableCopy.forEach[
      transformReference(it)
    ]

    state.parentRegion.states += it
    id = state.id
    label = state.label
  ]

  // This has to be done for each bindable object.
  for(eObject : newState.eAllContents.toList) {
    if (eObject.isBindable) {
      for(binding : state.bindings) {
        eObject.valuedObjects.bindTo(binding)
      }
    }
  }

  state.bindings.forEach[ binding |
    newState.declarations.immutableCopy.forEach[
      val objects = valuedObjects.filter[ name == binding.formal.name ].toList
      objects.immutableCopy.forEach[ delete ]
    ]
  ]

  newState.declarations.immutableCopy.forEach[ declaration |
    if (declaration.isInput || declaration.isOutput) {
      declaration.valuedObjects.forEach [
        val newObject = (newState.eContainer as Scope).findValuedObjectByName(name)
        newState.replaceAllOccurrences(it, newObject)
      ]
      declaration.delete
    }
  ]

  state.incomingTransitions.immutableCopy.forEach[ targetState = newState ]
  state.outgoingTransitions.immutableCopy.forEach[ sourceState = newState ]

  newState => [
    initial = state.initial
    ^final = state.^final
  ]

  state.remove
}
```

2.4. Hostcode

SCCharts possesses a hostcode mechanism that allows a modeler to include hostcode directly into the model. Text written in single quotes will be interpreted as hostcode. Although this is not new to SCCharts several improvements have been made.

Basically, the code generation creates the *tick function*, the *reset function*, and the declaration of used, not external variables. Additionally, in the C code generation, a header with matching name will automatically be included to provide a comfortable way to inject external objects or API.

2.4.1. Function Calls

As stated in Section 2.4 it is already possible to inject function invocations in models (e. g., in triggers or effects). However, regarding the performed analyses in the compiler this injected hostcode is a black box and the transformations do not know anything about it. Therefore, the hostcode mechanism got refined with respect to *function calls*.

Listing 2.11: Function call grammar rule

```
1 FunctionCall returns FunctionCall:
2     '<' functionName=ID
3     ('(' parameters += Parameter (',' parameters += Parameter)* ')')?
4     '>';
5
6 Parameter returns Parameter:
7     (callByReference ?= '&')?
8     expression = Expression;
```

Listing 2.11 depicts the new rules for such a function call. A call is introduced with a left angle bracket and followed by the name of the function. Afterwards a optional *parameter* list may follow. The call concludes with a right angle bracket. A parameter holds any expression (meaning a literal, a VO, or again hostcode) and may be preceded by an ampersand to indicate a *call-by-reference*.

With this calling convention in place the compiler is able to detect variable accesses in hostcode function calls and can create dependencies accordingly. The function call rule may appear anywhere where hostcode is allowed. An example is depicted in Listing 2.12.

Listing 2.12: Function Call Example

```
1 scchart FCMOVE {
2   int retVal, speed;
3
4   initial state s1
5   --> s2 immediate with / retVal = <move(100, &speed)>;
6
7   final state s2;
8 }
```

3. The Railway Project

As stated in the introduction in Section 1, the railway project was a bachelor and master project course held in the summer term of 2014 at the Chair for Realtime and Embedded Systems. Seven students attended the class accompanied by three supervisors. The goal was to design and implement a railway controller that is able to control the railway installation of the Department of Computer Science. In the initial tutorial phase of the project the participants got to know SCCharts and the actual state of the compiler. Afterwards they should estimate what was possible to implement in the remaining time before the actual implementation phase began.

Since SCCharts and especially the SCCharts compiler are a rather new development, this project was conducted in a fairly different manner than usual because the developers of the compiler were also the supervisors. The information feedback of the students

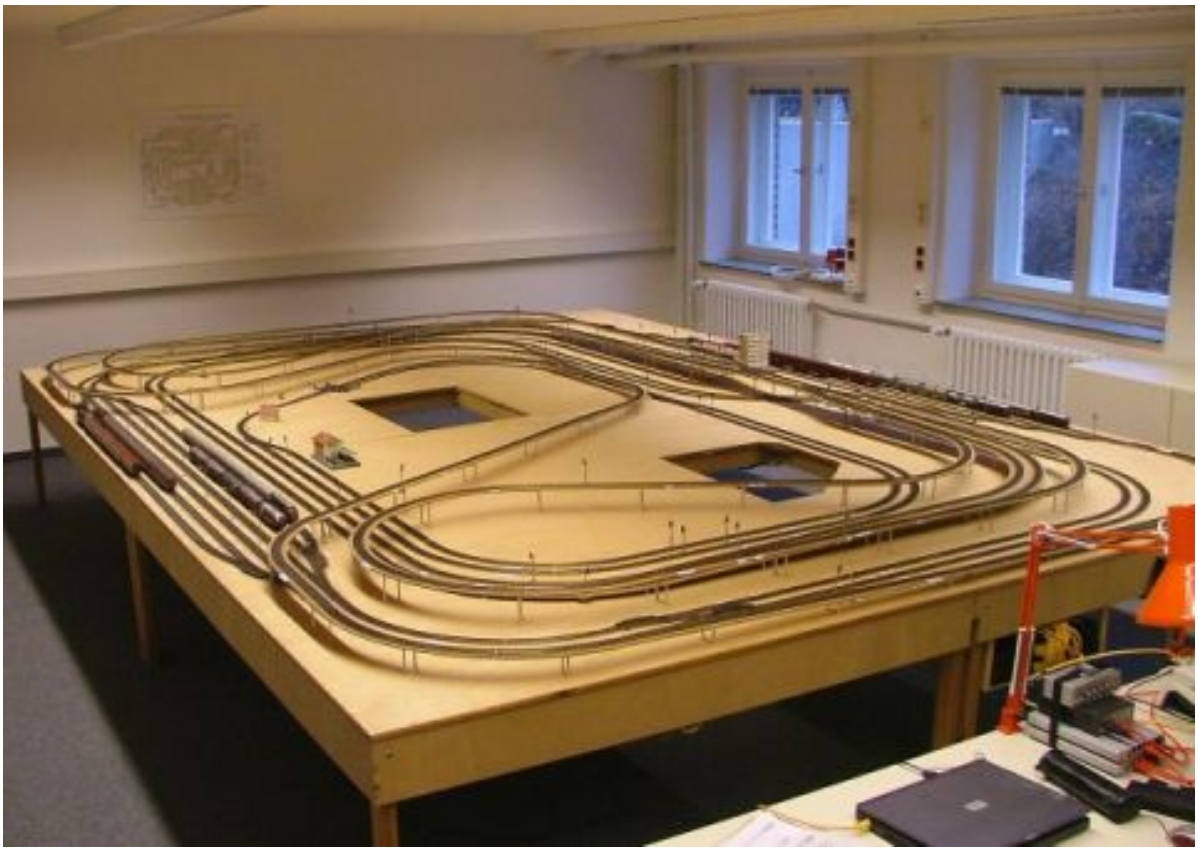


Figure 3.1.: Model railway installation

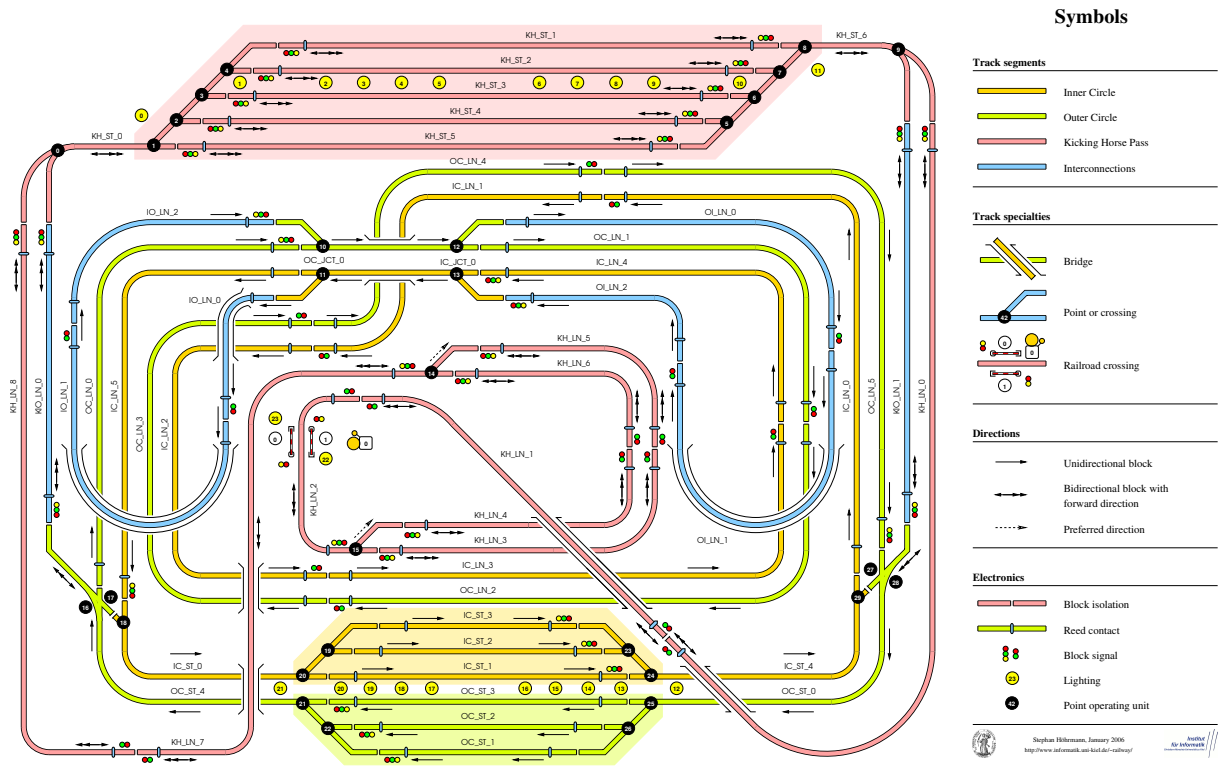


Figure 3.2.: Model railway scheme

w.r.t. the compiler, bugs, and large models were most valuable. On the other side, improvements and bugfixes were implemented in short cycles.

Section 3.1 introduces the model railway installation. Then, the overall approach of the class and the modeled controller is presented in Section 3.2. The chapter continues with the implementation details of the surrounding C code in Section 3.3 and concludes with remarks about the project workflow in Section 3.4.

3.1. The Model Railway Installation

To present the model railway installation at first the hardware is detailed in Section 3.1.1. The programming interface which controls the hardware is detailed in Section 3.1.2. These sections only provide a brief introduction to the system. for more detailed information, see the thesis by Höhrmann [Höh06].

3.1.1. Railway Hardware

Figure 3.1 shows the third generation of the model railway installation[1] of the Department of Computer Science at Kiel University. The hardware consists of 127 meters

[1] `http://www.informatik.uni-kiel.de/~railway/`

of tracks, 28 switch points, 58 signals, 24 lights and a railroad crossing [Höh06]. All these parts are normal Commercial of-the-shelf (COTS) railway components which can be found on numerous railway installations worldwide. The majority of the installation is built with products of the company *Fleischmann* with some additions of third party parts.

The track net is divided in 48 track segments. The complete track layout is depicted in Figure 3.2. The tracks form three separate circles, called Inner Circle (IC), Outer Circle (OC) and Kicking Horse Pass (KH). The name and design of the Kicking Horse Pass are inspired by the original pass across the Canadian Rockies, an equally famous and infamous pass built in the 1880s. Each of these circles has a station with several parallel tracks. On the Inner and Outer Circle, trains are only allowed to travel in one direction, namely clockwise on the Outer Circle and counter-clockwise on the Inner Circle. The Kicking Horse Pass can be used in both directions and has a turnout track in the middle of the circle. The three circles are interconnected to allow trains from one circle to change to another circle.

Most of the track segments are equipped with redundant reed contacts at each end of the segment. These contacts allow the controller to register passing trains, which are equipped with small magnets at the front and at the back. One crucial feature is the usage of non-magnetic tracks throughout the installation to avoid interferences between the tracks and the reed contacts. Due to the redundancy, the sensor can also provide informations about the travel direction of the train and a rough estimation of the speed.

The overall 245 sensors and actors are controlled by 24 computing nodes, mounted under the intallation. Each node consists of a custom-build power electronics circuit and a connected PC104 386 computer. As an alternative to the PC104 system, Time-Triggered Protocol (TTP)[2] power nodes can be used with the power electronics. The PC104 nodes are interlinked via Controller Area Network (CAN) bus and Ethernet while the power nodes can access a separate Ethernet, CAN bus and TTP bus. To control the PC104 nodes of the installation, an Application Programming Interface (API) written in C is available to the developer.

3.1.2. Programming Interface

The main programming interface is provided by the so-called *Höhrmann-API* [Höh06]. Using this API a separate controlling computer connects to the nodes via CAN bus or User Datagram Protocol (UDP) and has the ability to retrieve status information from the railway system or to change the status of the railway hardware. To initialize the controller a `railway_system` datastructure has to be initialized and a link to all nodes has to be established as shown in Listing 3.1. When these connections have been created the controller can claim the hardware and define boundaries for the length of a cycle.

To determine whether the railway is still properly connected and working as expected, the API provides the function `railway_alive` which is also shown in Listing 3.1.

[2] `http://www.tttech.com/technologies/ttp/`

Listing 3.1: Railway controller initialization

```
struct railway_system *railway_initsystem(struct railway_hardware *hardware);
int railway_openlinks_udp(struct railway_system *railway, char *hostformat, char *device);
int railway_startcontrol(struct railway_system, unsigned mincycle, unsigned maxcycle);

int railway_alive(struct railway_system *railway);
```

When the controller has established a connection it can control the individual parts of the hardware with several functions shown in Listing 3.2. Each of these functions needs the `railway_system` as the first parameter, followed by one or more parameters to select the individual hardware part. The next parameters determine the desired state of the component. Most of the parameters can be filled with predefined constants like GREEN, YELLOW or RED for signal states, BRANCH or STRAIGHT for point states or OFF, FWD, REV or BRAKE for the track settings. To select the track segments, precompiler constants like IC_ST_2 are available for each individual track.

Listing 3.2: Railway control functions

```
void setsignal(struct railway_system *railway, int block, int signal, int lights);
void settrack(struct railway_system *railway, int track, unsigned mode, unsigned target);
void setpoint(struct railway_system *railway, int point, int state);
void setlight(struct railway_system *railway, int light, int state);
void setgate(struct railway_system *railway, int gate, int state);
```

To track the movement of the trains the primary sensors are the reed sensors in the tracks. Sensors can individually be interrogated and return their current status. The primary function to read the status of these contacts is shown in Listing 3.3.

Listing 3.3: Railway contact query

```
unsigned getcontact(struct railway_system *railway, int block, int contact, int clear);
```

This function returns one of the following values:

NONE No new event has been detected.

FWD The last passing magnet has passed the sensor in forward direction

REV The last passing magnet has passed the sensor in reverse direction

UNI An event has been detected but no direction could be determined. This might either happen with very fast trains[3] or due to a defective reed sensor. The firmware of the power electronics can differentiate between these faults and presents an error message if one reed sensor seems to be faulty.

The API provides several additional fuctions like testing the existence or probing the status of railway parts.

[3]Faster than 3 meter per second.

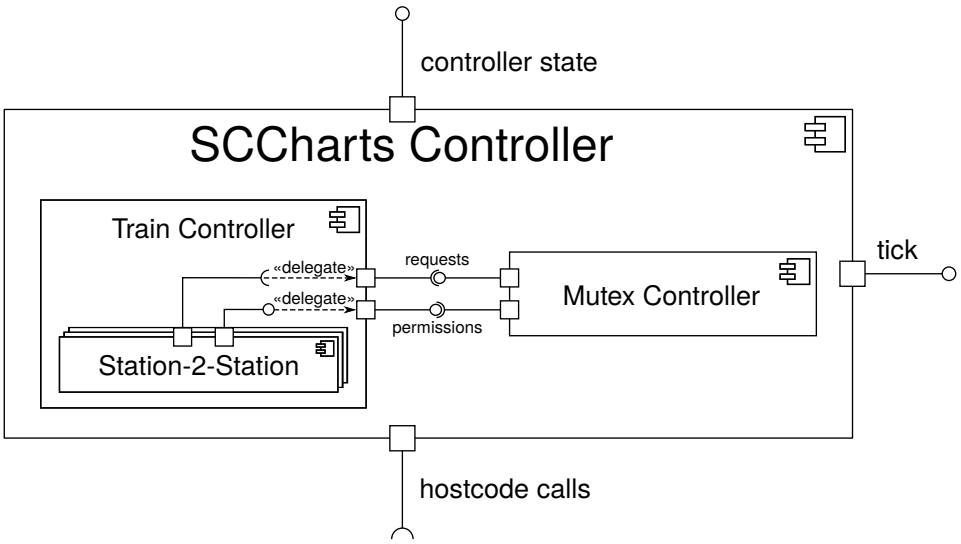


Figure 3.3.: SCChart railway controller hierarchy

3.2. The Overall Approach

The general approach to create a complete controller was a bottom-up design, starting with smaller components which could be distributed among the group and combining these components on higher layers. In general three layers of abstraction were used to create a controller for the model railway. The lowest layer consists of so-called *Station-2-Station* controllers. These models can be used to travel with any train from one specific station to another adjacent station. These models were then used to create a general train controller which is able to dynamically receive the next destination and select the appropriate Station-2-Station controller. In the end 11 of these train controllers were combined to create a complete railway controller. To manage the permissions on the track segments, and to prevent train collisions, this railway controller needed additional mutex controllers. Each of these mutex controllers is able to manage the permissions of one track segment, resulting in 48 mutex controllers in the final controller. A schematic visualization of this controller hierarchy can be seen in Figure 3.3.

3.2.1. Station-2-Station Controllers

Due to the track layout, which is shown in a simplified version in Figure 3.4, some routes were not feasible. For instance a train in the Inner Circle Station traveling to the Kicking Horse Station will always arrive there in reverse direction. If it should travel to the station facing forward it must pass through the Outer Circle first. These indirect connections were not modeled explicitly but built on the next layer.

The remaining connections were:

- One loop on the Inner Circle
- From Inner Circle to Outer Circle

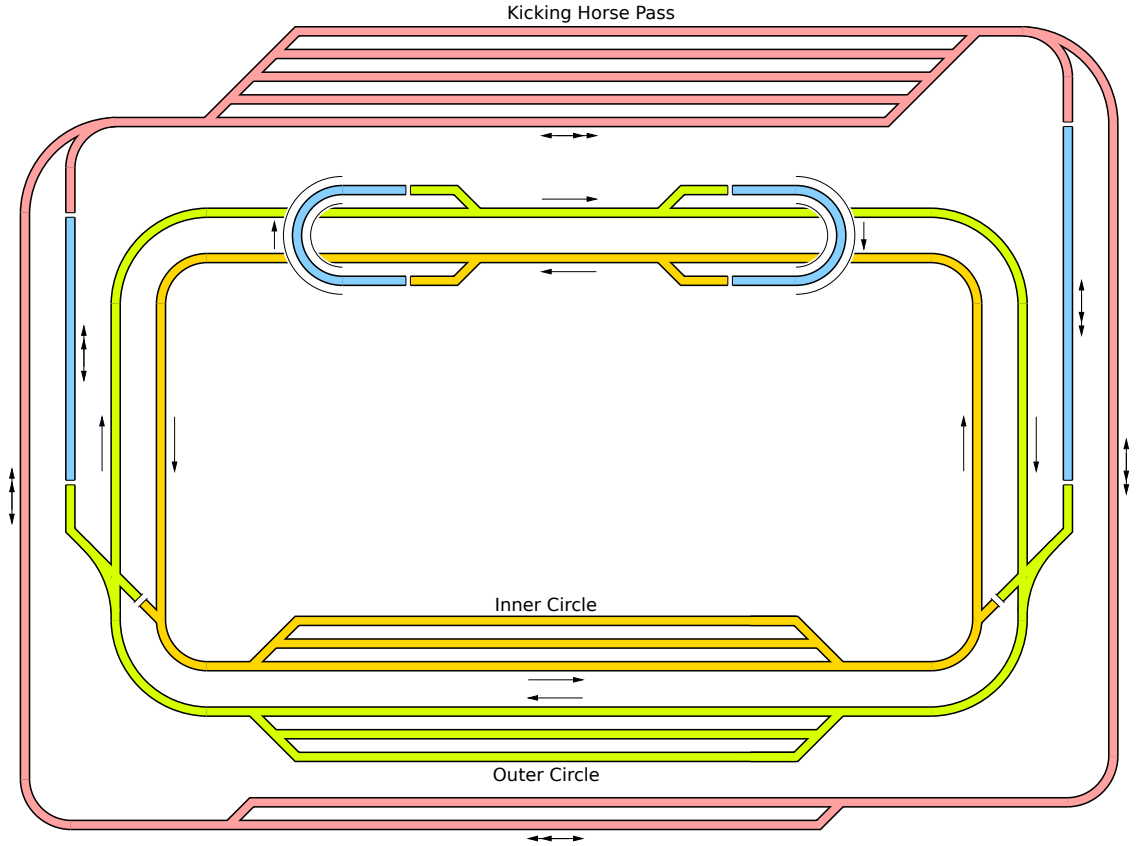


Figure 3.4.: Railway stations and circles

- From Inner Circle to Kicking Horse Station counter-clockwise
- One loop on the Outer Circle
- From Outer Circle to Inner Circle
- From Outer Circle to Kicking Horse Station clockwise
- One loop on the Kicking Horse Pass clockwise
- From Kicking Horse Station clockwise to Outer Circle
- One loop on the Kicking Horse Pass counter-clockwise
- From Kicking Horse Station counter-clockwise to Inner Circle

Each of these scenarios was modeled in a separate SCChart and had a similar interface like in Listing 3.4.

Listing 3.4: Station-2-Station interface example

```
1 input int IC_JCT_0_perm, IC_LN_0_perm, IC_LN_1_perm, IC_LN_2_perm;
2 input int IC_LN_3_perm, IC_LN_4_perm, IC_LN_5_perm, IC_ST_0_perm;
```

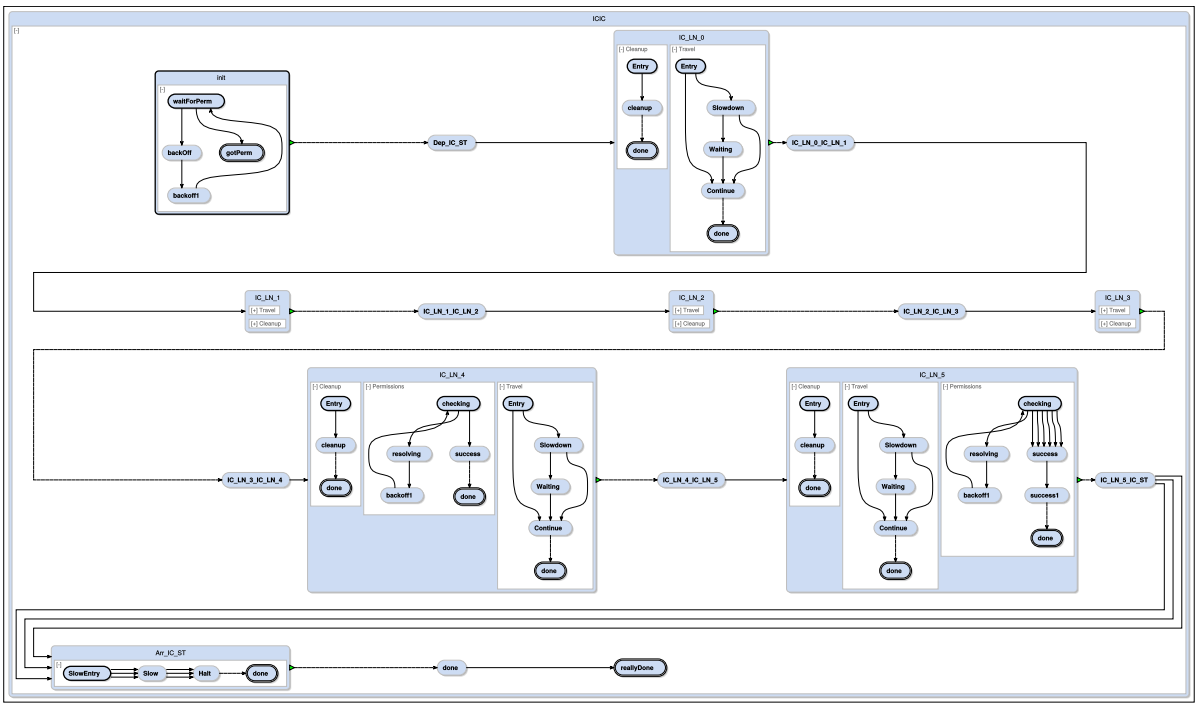


Figure 3.5.: Station-2-Station controller (without declarations and transition labels)

```
3  input int IC_ST_1_perm, IC_ST_2_perm, IC_ST_3_perm, IC_ST_4_perm;
4  output bool IC_JCT_0_req[11], IC_LN_0_req[11], IC_LN_1_req[11], IC_LN_2_req[11];
5  output bool IC_LN_3_req[11], IC_LN_4_req[11], IC_LN_5_req[11], IC_ST_0_req[11];
6  output bool IC_ST_1_req[11], IC_ST_2_req[11], IC_ST_3_req[11], IC_ST_4_req[11];
7
8  input int trainNum;
9  input int depTrack;
10 input int destTrack;
11 input bool cleanup;
12 input bool debug;
13 output int arrTrack;
14 output bool trainTravelling;
```

Each train is assigned an individual number, which is passed to the controller as the constant trainNum. The departure track of the train is passed via depTrack and the desired arrival track is defined in destTrack. The controller has a form of load balancing on the station tracks, meaning that a train won't insist on the given track but changes to another free track in the station if needed. The real arrival track is returned to the controller via arrTrack. To assert mutual exclusion on the track segments, each segment has a pair of request and permission variables. Each train can request the permission to enter a track segment by setting the various request variables to true. These variables always take the form XX_YY_Z_req[trainNum] where XX stands for the current circle, YY choses the general area (Station, Lane or Junction) and Z denotes the exact block.

The permission is granted by the mutex controller which is described in Section 3.2.3 as a higher level controller. When the permission has been granted the variable XX_YY_Z_perm is set to trainNum.

A sample of a Station-2-Station controller is shown in Figure 3.5. The leftmost state

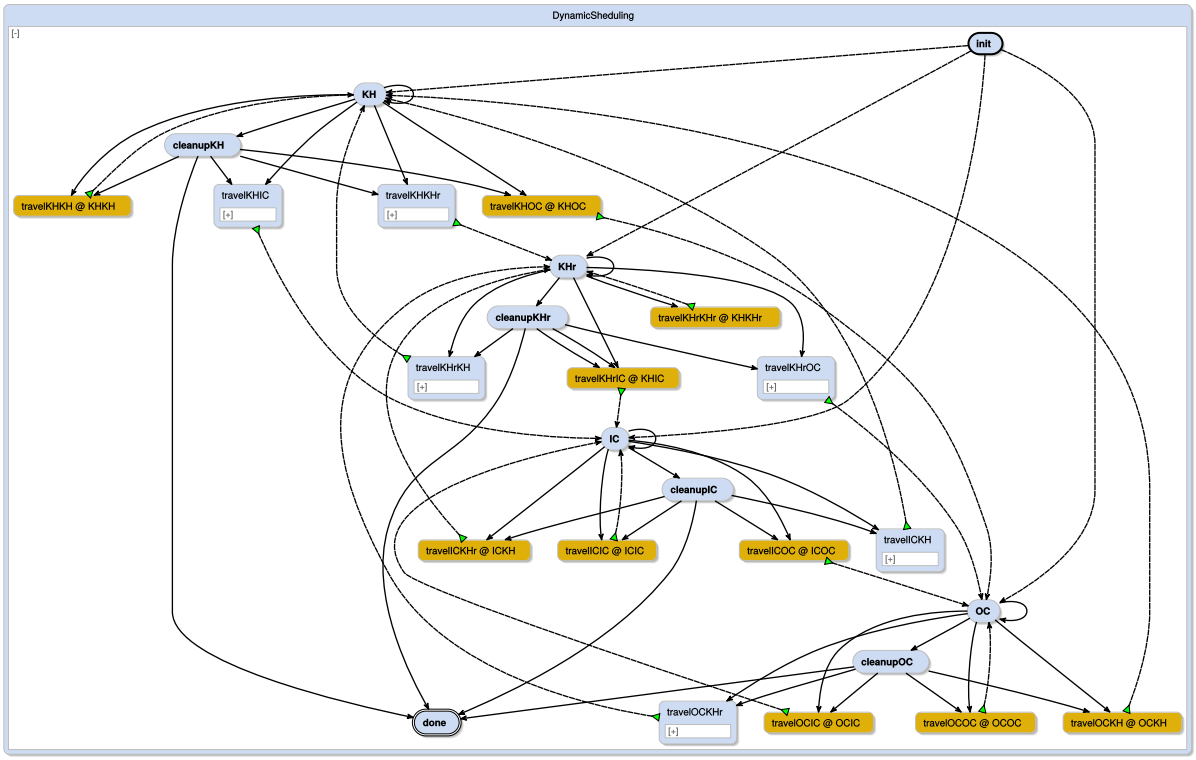


Figure 3.6.: Dynamic train controller (without declarations and transition labels)

takes care of leaving the station while the successive states manage the passage through one track segment each. The Station-2-Station controller takes care of handling all signals, points and tracks and fetches all relevant contact events. It requests the needed track permissions and slows the train down (or stops it completely) if the next segment is still occupied by another train. The states at the right end manage the load balancing, the arrival at the station and the waiting time at the platform. When requesting the station track it tries to grab the primary destination track, but will switch to any free track if the destination is blocked. When entering the station it will first slow down and then stop at the end of the station track. It then signals the arrival to the C controller part, which causes the train to wait some time before terminating this Station-2-Station controller.

3.2.2. Train Controller

The train controller, visualized in Figure 3.6, takes care of the complete dynamic management of one train. In this context dynamic management means that the controller is not only using a predefined train schedule but is able to receive the next destination during runtime and react to changes in the schedule. This schedule can be set by the user using the Transmission Control Protocol (TCP) interface described in Section 3.3.2. Each schedule consists of a list of destinations and a pointer to the active entry in this list. If the train reaches the end of this list it restarts the schedule and takes the first

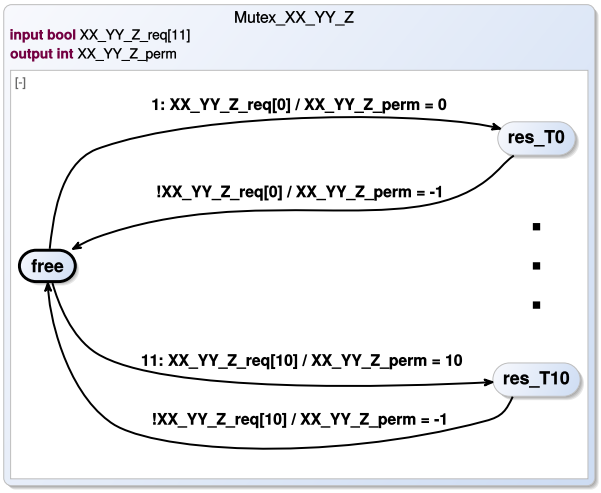


Figure 3.7.: Mutual exclusion controller (simplified)

element of the list as the next target. It the schedule is empty the train stays at the current station until a new schedule is set.

To accomplish this, each train controller tracks the location of its individual train. The primary control function can be found in the four states IC, OC, KH and KH which are active when the train is standing in the corresponding station. In theses states the controller polls the environment for the next destination. When the new destination has been selected, the controller selects the shortest path and starts the corresponding Station-2-Station controllers. In simple cases a single Station-2-Station controller can be used directly and the termination points to the next station state. If no direct connection between the current station and the destination exist, the controller combines multiple Station-2-Station controllers and returns to the final station state after all intermediate steps have been passed.

3.2.3. Mutual Exclusion Controllers

To manage the permissions on one track segment one separate region, as shown in Figure 3.7, is modeled. This region contains one free state and one state for each train. Each train can request the right to enter the segment by setting its request variable to true. If the segment is free the mutex controller selects the train with the highest priority which wants to enter and gives the permission to it. This permission is then kept until the train releases the segment by setting the request to false again.

3.2.4. Complete Railway Controller

To build a complete railway controller the separate components have to be combined. Each train is controlled by a separate train controller and each segment has to be managed by a mutual exclusion controller. In addition one state for initialization of the whole system is needed which reserves the initial tracks for the trains. Due to the design

this initial configuration is not determined from the railway system itself but explicitly modeled.

3.3. Controller Implementation

The main part of the controller implementation process was done by modeling the SCCharts components according to the approach given in Section 3.2. When the modeled controller should actively control the railway, it needs to be executed. The compilation of SCCharts produces a sequentialized synchronous *tick function*. In this case C code was synthesized. Consequently an additional environment, including a *main function*, is needed to compile and execute this code.

In general a main function calling the tick function repeatedly would be sufficient to simply run the modeled controller. But there are more aspects which must be considered to create a working and usable controller.

The railway system is an event driven system. The synchronous model of time must be preserved by consuming events only on tick-borders. Thus it is necessary to add a separation layer between the hostcode calls in the synchronous tick function and the railway API function calls. This layer of abstraction is also suitable for further abstraction from the given railway API to offer a simplified and specialized interface for the hostcode used in the SCChart model.

Another important aspect of the controller is user interactivity which cannot be handled easily in the SCCharts model. A simple controller which drives all trains in an predefined way might be sufficient for testing. But already at testing it is handy to allow the testers to pause and continue the controller in a secure way, instead of canceling its execution.

In addition to these main extensions the controller also offers further features such as measurement of consumed time during a tick and sanity checks of the railway behavior with appropriate error handling.

Extending the generated controller by implementing these additional features and aspects leads to a more complex controller.

3.3.1. Components

Figure 3.8 shows the different components of the railway controller and their communication interfaces. The diagram only illustrates the architecture of the controller implemented in C. The structure of the SCCharts controller is explained in Section 3.2 and only appears as the SCCharts Controller component.

The following paragraphs describe each component presenting its purpose, functionality and interaction with other components.

SCCharts Controller This component represents the compiled controller modeled in SCCharts. The generated code defines a sequential synchronous tick and reset function to

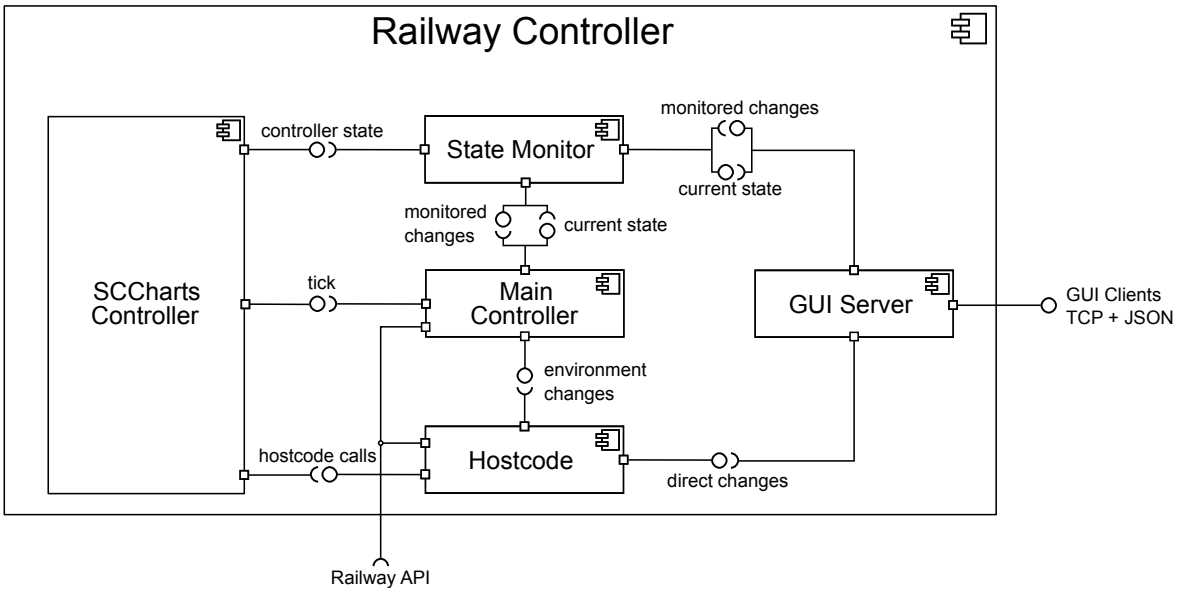


Figure 3.8.: Component diagram of the railway controller in C

drive the modeled behavior. These functions are invoked by the Main Controller component. The tick function itself uses functions defined by the intermediate hostcode API allowing interaction with the railway and additional controller features. To gain more detailed information about the inner state of the SCCharts Controller the State Monitor accesses some internal variables.

Hostcode The Hostcode component defines a extended interface wrapping the railway API by wrapping its functions and providing additional functionality. All needed API functions are redefined with a smaller signature and parameterized functionality is handled internally. This reduces and simplifies the hostcode calls in the SCChart model. All functions changing the environment of the tick function are handled by the Main Controller component to ensure a correct behavior according to the synchronous constructive Model of Computation (MoC). Other functions which have no temporal context are directly forwarded to the railway API.

In addition to these basic railway IO functions the Hostcode component also provides an interface to more advanced features such as dynamic scheduling, network communication aware message logging, abstract train speed and other functions regarding the trains behavior protocol.

Main Controller The core of the overall Railway Controller is the Main Controller component. It contains the main function and consequently handles the complete start up of all components and the correct shutdown sequences of the controller. The main purposes are the correct management of the system and invoking the tick function provided by the SCChart Controller. Additionally, it maintains a synchronous environment for the

tick function. All actions which change the environment are delayed until the current tick ends and processed afterwards. Contact events and other events are polled via the railway API at the tick border. Thereby event-based changes, such as the gates on the railway, are transformed into state-based data. Changes received from a Graphical User Interface (GUI) client are only applied between two ticks. Additionally, the Main Controller component creates information about the current state for the State Monitor component.

In addition, the tick loop can be paused. The complete state of the railway and the controller is saved and restored on continuing. This facilitates debugging.

To allow some evaluation of the efficiency of the system, the time consumption inside and outside the tick function is measured. Furthermore, the controller checks internal permissions of the SCChart Controller against detected trains on the railway to increase robustness of the system.

GUI Server The server component allows multiple GUI clients to connect to the controller. Each client connection is handled in its own thread, while disconnecting and establishing new connections is always enabled during the controller execution. The detailed communication is explained in Section 3.3.2.

Most commands and requests are performed delayed and thread-safe via the State Monitor. Some uncritical commands are directly performed via the Hostcode component. Different clients sending the same commands at the same time are processed with mutually excluded first-come-first-serve-principle, which may cause lost update writes but does not affect the integrity of the Main Controller component. Broadcasting log messages is handled independently from client requests.

State Monitor The State Monitor component allows a thread-safe data exchange between GUIs connected to the GUI Server and the running Main Controller. It provides the state information of the last completed tick and buffers changes ordered by GUI clients. Consequently this communication is tick-safe because changes are only applied at tick-borders. In addition to that, the data access is mutually excluded without blocking the controller thread and thus the controller remains a reactive system.

3.3.2. TCP Communication

Early versions of the controller provided only a restricted set of controls via keyboard interrupts (namely *pause*, *cleanup* and *shutdown*). It quickly turned out that further control features would enhance the usability. To keep the C controller small, understandable and maintainable, the user interface has been swapped out to a separate program. The first version of a GUI communicated with the controller via pipes. The next step was to further generalize the communication between them. Network-based communication provided the extra feature of separating controller and GUI onto different computers.

To control the railway system and its trains, clients can connect to the controller via TCP. The communication is line-based and uses JSON-formatted strings to send

commands and status messages. Possible commands are ones like `status`, which calls for the current status, or `schedule`, which sets a new schedule for a specific train. A list of available commands can be found in the listing below. All commands are asynchronous calls, meaning there is no response to a command. By using TCP as the underlying protocol, no responses are needed as one can rely on the successful transmission of TCP.

The only (implicit) response is possible via polling updated status information and comparing the previously set property with the one from the status message. The status message sent by the controller contains general information about the railway system such as debug and pause state, number of traveling trains and statistics about tick times as well as train specific information such as if the train is currently traveling, it's next station, set speeds for different situations (i. e., entering stations, waiting for free track segments) and the current schedule. Other messages are error and log messages which are quite self-explanatory.

Messages sent by the controller

log Contains the log messages as an array of strings.

error Contains a single string describing the error.

status Contains debug, cleanup and pause state, the number of traveling trains, locked track segments, tick time, overall loop time and for each train the state of traveling, the remaining waiting time in stations, several speed settings, waiting times for different stations, the schedule and current destination.

Messages sent by clients

status Poll for the current status, should trigger a status reply message.

shutdown Terminate the controller and close all sockets.

logout Close the connection, leaves the controller running.

pause Payload: `state`. Suspend or resume the controller according to `state`.

cleanup Start the cleanup procedure which sends the trains back to their home tacks.

debug Payload: `state`. Enable or disable verbose output according to `state`.

echo Payload: `message`. Print `message` as log message on the controller and broadcast it to all connected clients.

light Payload: `state`. Activate or deactivate the lights on the railway according to `state`.

wait Payload: `train`. Force `train` to wait in the next station.

start Payload: `train`. Force `train` to immediately abort the waiting timer.

schedule Payload: `train`, `currentIndex`, `tracks`. Set the new schedule for `train` where `currentIndex` is the current integer position in the schedule and `tracks` is an array of integer values, representing the station tracks.

speed Payload: `train`, `speeds`. Change the speed settings of `train` where `speeds` contains the new speeds for slow, caution and normal.

time Payload: `train`, `times`. Change the waiting times for `train` where `times` contains minimum waiting time and maximum random offset for each station.

3.3.3. GUIs

The use of TCP and JavaScript Object Notation (JSON) based communication makes it possible to connect to the controller from different platforms. To control the railway in a convenient way, there is a GUI running on the common (desktop) Java Virtual Machine (JVM). Another GUI runs on Android devices which allows monitoring and controlling the railway from all around the room. This turned out helpful while testing the modeled controllers as one can easily pause the controller while supervising the trains on the tracks.

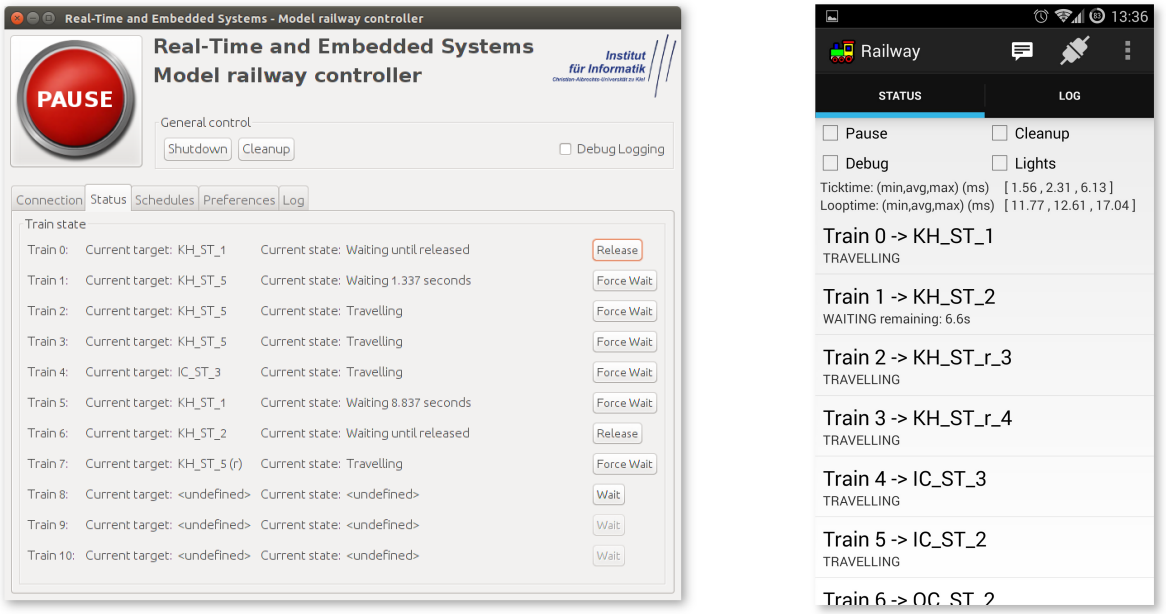
Both GUIs have a similar structure. There are different panels for monitoring the status, editing train preferences and viewing the log. The information viewable and editable are described in detail in the previous Section 3.3.2. Figure 3.9 shows the status view from both applications. Here, one can see the state of each train, its current destination, if it's waiting and the remaining waiting time. The different panels for editing the trains or viewing the log are accessible via tabs and on android via tap on each train.

3.4. Project Workflow

The creation of the controller was carried out in several phases as depicted in Section 3.2 and Section 3.3. The different phases of the implementation and the resulting increase in Lines of Code (LoC) are shown in Figure 3.10a.

As described before the complete controller includes the Station-2-Station and the Dynamic Controllers. These result in approximately 14.000 LoC. Additional features such as network communication added another 11.000 Lines of Code, hence, the overall model comprises roughly 25.000 LoC. A detailed evaluation of the code composition of automatic generated and manually written code will follow in Chapter 4.

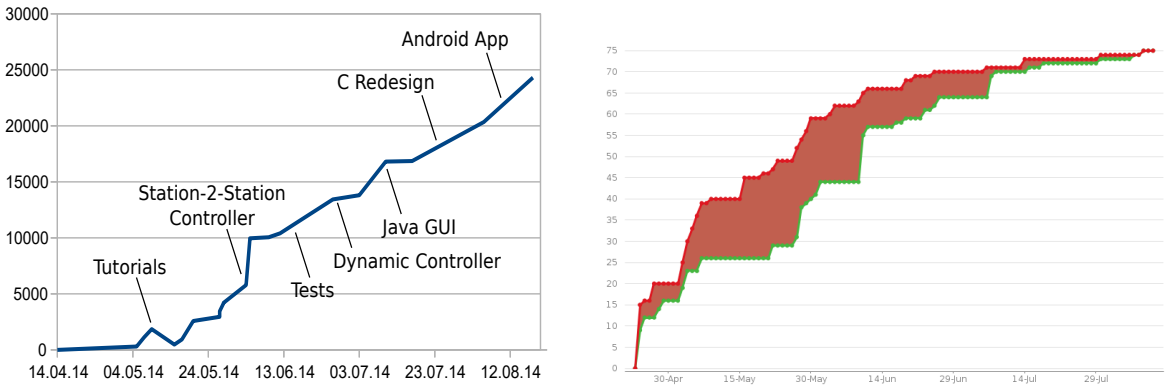
As this academic project was part of the educational training of the participating students, a close communication loop with the SCCharts developers was possible. The feedback of the students and the prompt handling of issues were valuable for both sides. Figure 3.10b shows the generated tickets in the group's issue tracker in red and the resolved tickets in green.



(a) Desktop status view

(b) Android status view

Figure 3.9.: Example views from the GUIs



(a) Controller lines of code

(b) Project tickets

Figure 3.10.: Project workflow

4. Validation and Experimental Results

The final version of the controller presented in Chapter 3 successfully managed the schedule and controlling of all 11 trains on all tracks. A recorded video of the controller in action can be found on the homepage of the group[1].

A big bonus of this success results in the immediate availability of large-scale and practically tested SCCharts models. Section 4.1 depicts statistics about the scale of produced models and code in the project. Furthermore it will give a detailed analysis about the increase of the code size during compilation. Subsequently, Section 4.2 presents the results of the improvements on the SCCharts compiler and finally, the actual execution time performance of the generated code is examined in Section 4.3.

4.1. Source-Code Size

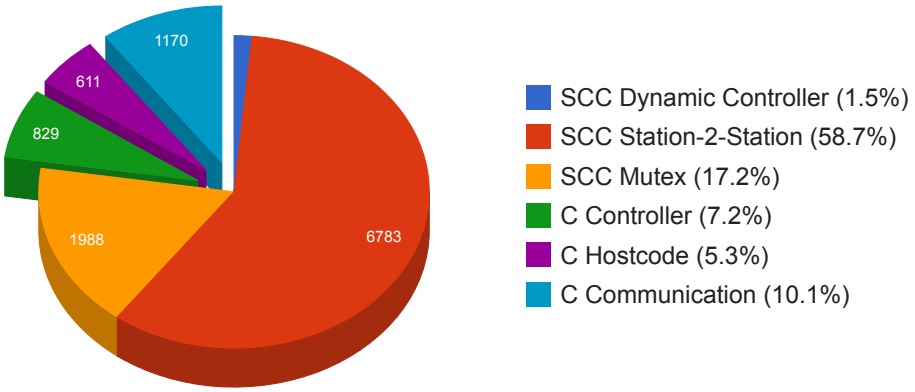


Figure 4.1.: Distribution of self-written code in LoC

Regarding the modeling approach illustrated in Section 3.2 and the architecture mentioned in Section 3.3.1, the controller consists of independent components written in different languages. To evaluate the distribution between textual modeling and coding in C, the amount of self-written code is compared. Figure 4.1 shows the code size of the different components in the final controller. The amount is measured in LoC but without a standardized formating and indention style for both languages. Nevertheless, the results are comparable since the team had general formating rules, producing similar structured code. Another issue is the SCChart Mutex component which is not entirely

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

self-written because some parts are generated by a script to save monotonic repetitive work. The ability to use scripting was a benefit of having a textual input format.

Considering these results, the modeling part took three quarters of the work, which is appropriate for a project focusing on modeling. Furthermore, the slices highlighted in the pie chart of Figure 4.1 represent the written C code. Regarding the purpose of the components, some components are not strictly necessary to run the modeled controller. It turned out that the amount of C code needed to correctly run a SCChart in a real world environment is relatively small.

4.1.1. Compilation

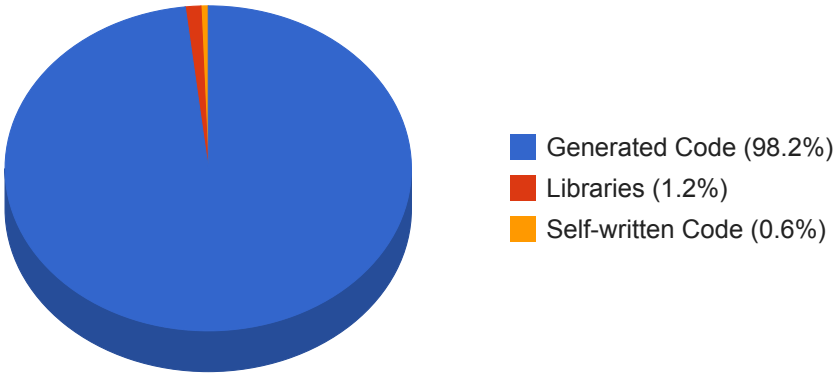


Figure 4.2.: Distribution of the complete C code before compilation

In the previous section, only self-written code was compared and also textual model code and C code was mixed. Figure 4.2 shows the size of C code source files used in the final compilation with the GNU Compiler Collection (GCC). The amount is again measured in LoC.

The files are combined to groups presented in this chart. The Self-written Code refers to the self-written C code files which are highlighted in Figure 4.1. In addition to the standard libraries the Libraries slice represents the railway API and a JSON parser. The most significant group is the Generated SCChart Code. This group consists of a single file generated by the *KIELER Compiler* when compiling the source models into C code. Since this code is generated, the indent style does not follow the style used in self-written code and libraries. The LoC in this file are measured with standard GNU indent style. Any differences in the amount of line resulted by this style are insignificant due to its relative size to the other files.

Considering the results presented in Figure 4.2, an enormous increase of the SCCharts code can be observed. The nearly 12,000 lines of textual model code resulted in about 450,000 lines of C code. The reason for this significant increase of code can be found in the SCCharts compilation.

4.1.2. SCCharts Compilation

The compilation of SCCharts is a process of stepwise model-transformations [MSvH14]. Each step changes the model according to the purpose of the transformation [vHDM+14]. Since most transformations replace extended features by simpler features with a more complex structure or add information to the model, the model size grows.

Experiment

To acquire detailed information about the growth of the model and the distribution of model elements, the *KIELER Compiler* was extended by an analyzer module. The KIELER version[2] chosen in this experiment was the state used to compile the final railway projects. The added module analyzed every intermediate result of the compile chain applied on the complete railway controller.

In each intermediate model the number of model elements relevant for its structure was measured. For SCCharts these elements are states, transitions, regions and variables. Consequently, in the Sequentially Constructive Graphs (SCGs) the nodes, control-flow-edges, threads represented by entry and exit node pairs and variables were counted.

In the previous sections, code size is measured in LoC which could not be acquired in this experiment. The models could either be saved in a XML Metadata Interchange (XMI) format which is not comparable to human written code or serialized into a proper text representation. Due to the size of the railway models and the resulting intermediate models, the synthesis was not able to terminate in finite time. In addition to that, the analysis of the model elements allows a more detailed view on the inner structure of the SCCharts than LoC.

Results

Figure 4.3 shows the combined and processed results of the experiment. The complete raw data can be found in Appendix B.

The chart lists the transformation and presents the number of corresponding model elements after the given transformation is performed. The lines represent the combined model element categories for both SCCharts and SCGs. The Initial transformation refers to the modeled source files before any transformation is performed. Transformations without any effect on the model are omitted in the diagram. In case of the SCCharts part, this implies that the corresponding *extended features* were explicitly not used in the modeling process or not produced by other transformations.

The first applied transformation is the Reference transformation, performing a macro expansion. This transformation has great importance for modeling in the railway project, since it allows modularization and re-using of components. As a consequence, resolving the re-usage by creating copies of the referenced modules causes the increase of states, transitions and regions noticeable in the diagram. The number of variables is slightly affected because most variables in the railway model are globally shared variables.

[2]Version: 0.10 ; Commit: c577524c72d9a94362e5ed165a1645ceab365ce3

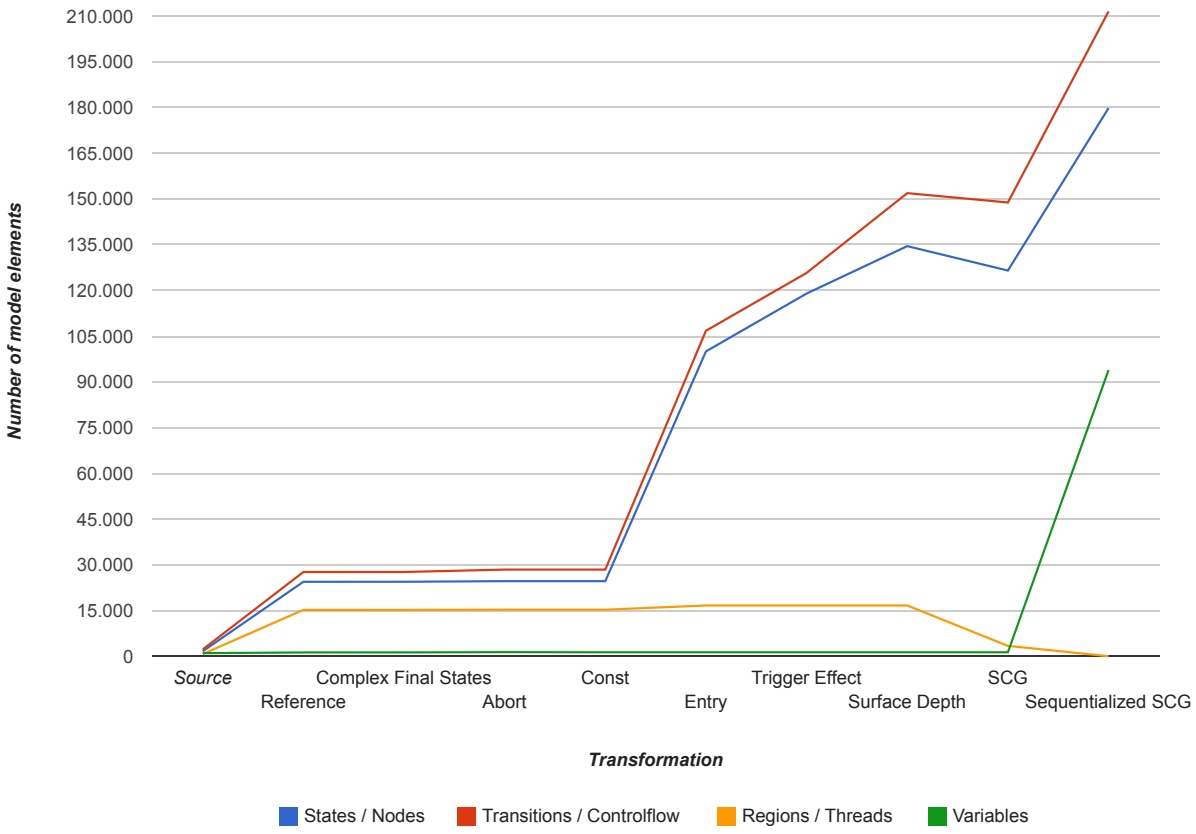


Figure 4.3.: Number of model elements during compilation

The succeeding transformations have insignificant effect on the model size because the corresponding extended features are rarely present in the model. In contrast, the *entry action* feature is heavily used in the controller model. It proved to be the easiest and visually most compact way to store textual hostcode calls in SCCharts. Since the hostcode contains all interaction with the railway, the controller contains many entry actions. The Entry Transformation translates these entry actions into sequences of states and immediate transitions holding the same effect as the entry actions. Consequently this transformation produces the huge amount of new structural model elements visible in Figure 4.3.

The Trigger Effect and Surface Depth transformation restructure the model into a more simple and compiler friendly but also larger structure. The translation of SCCharts into SCG benefits from this structure. Additionally the number of nodes in the SCG is smaller than the number of states and transitions analogously. The reason for this decrease is the elimination of hierarchy. This effect is also noticeable in the number of threads.

After a static scheduling, the SCG is sequentialized into a single-threaded sequence of actions. Again a significant increase of nodes, control-flow and especially variables is noticeable. This is caused by introducing synchronous guards to control the concurrency.

Analysis

In the end, the sequentialized SCG has about 180,000 nodes and 90,000 variables. Analyzing the translation of the SCG into C code explains the resulting number of LoC of the C file.

At first each variable requires a declaration, producing about 90,000 lines. Additionally a proper initialization of most of the variables is needed. Each node represents an assignment or conditional branch. A conditional is translated into an `if`-statements with the corresponding `if`-block consuming 3 lines of code. The difference between control-flow and nodes leads to the conclusion that there are about 30,000 conditional nodes, producing about 90,000 lines. Consequently, the remaining 150,000 nodes are assignments which can be translated into a single statement. Following the given indent style most of these statements are split in to multiple lines regarding the complexity of the expression and the length of the statement.

Summing up the estimated number of lines justifies the 450,000 LoC mentioned in Section 4.1.1. In addition to that, the increase of code size during the SCCharts compilation can be explained and traced to the different transformations and used modeling features.

4.2. Compiler Performance

We tested the performance of the SCCharts compiler w.r.t. three different versions of its development

1. June 2014,
2. August 2014, and
3. January 2015.

The June and August compiler versions were influenced directly from the running railway project. A lot of improvements could be made and incorporated into the August version. The January version shows that these improvements are still retained in the most current version of the SCCharts compiler.

We measured the compile time in seconds, the beautified target code size in lines of code and a normalized compile time per line of generated target code. We performed our experiments on an Intel Core 2 Duo T9800 @ 2.93GHz system with 8GB RAM. For presentation purposes we ordered the models in ascending order of their (model and target) size measured in lines of code where the smaller models are on the left and the larger models are on the right side of each diagram. For our experiments we used valid test models from the railway project, which served as regression tests during the project. Namely, these are Test4.sct, TestKHKH.sct, Test2.sct, Test3.sct, Test2b.sct, TestOCOC.sct, TestAll.sct, TestICIC.sct, Test1.sct, TestOCKH.sct, TestICKH.sct, and TestICOC.sct. Additionally, as an example of really large models, we also measured our compiler using two running versions from the final project presentation of the full dynamic railway

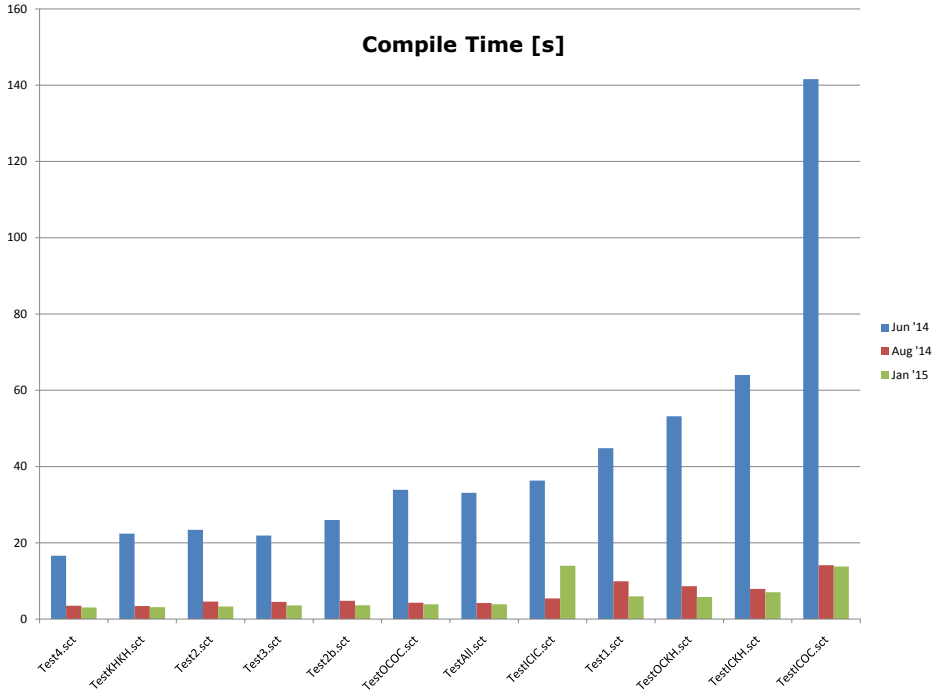


Figure 4.4.: Compile time for rail regression test models

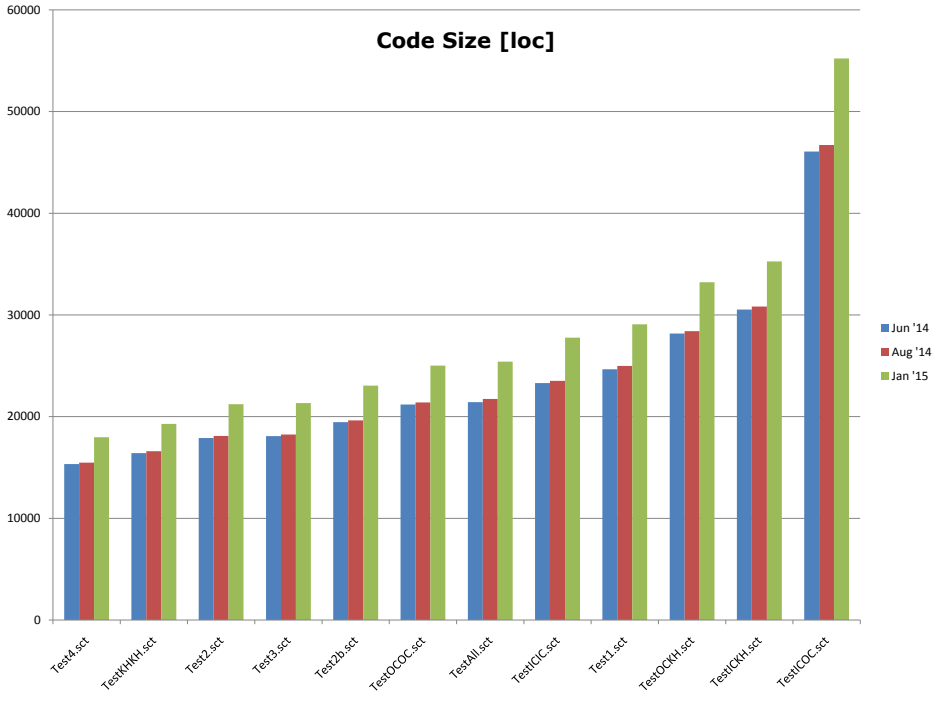


Figure 4.5.: Code size for rail regression test models

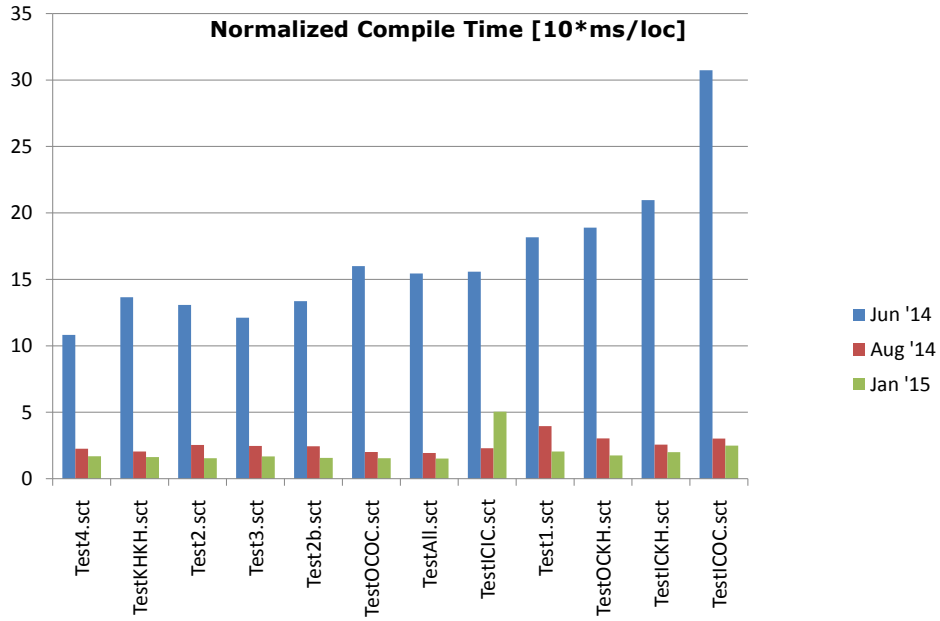


Figure 4.6.: Normalized compile time for rail regression test models

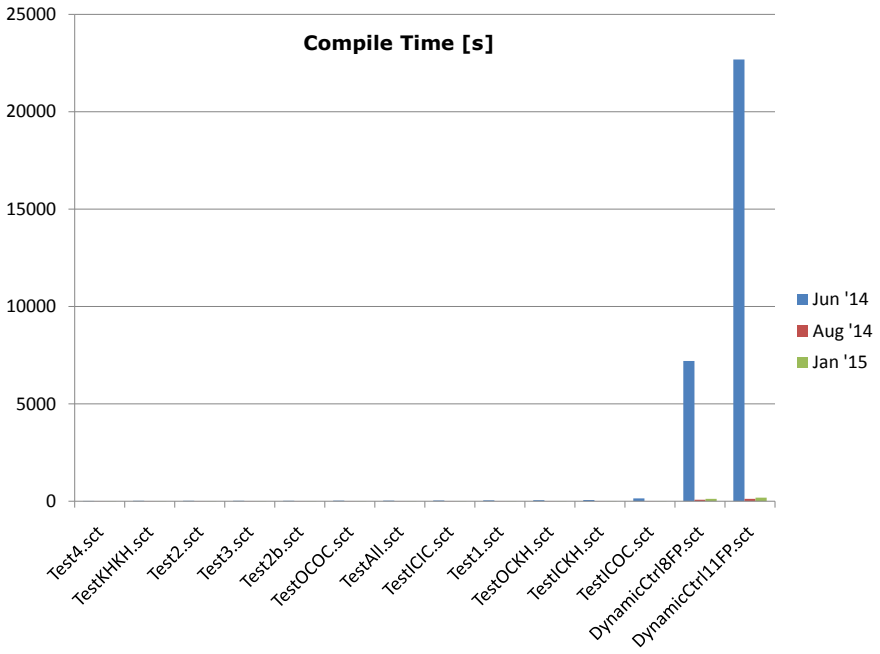


Figure 4.7.: Compile time for rail regression test models and final dynamic SCCharts controller

controller models, one for 8 trains, DynamicCtrl8FP.sct, and the other for 11 trains, DynamicCtrl11FP.sct. The full dynamic controller for 11 trains is the final outcome of this project which is also discussed earlier. The 8 train variant is a nearly equivalent controller that just runs 8 trains and hence is a lot smaller in model size.

Figure 4.4 shows the compile times for only the regression test models for all compiler versions. It can be seen that the June compiler version has major problems when in comes to larger models. Figure 4.7 also attests this where the large dynamic controller models for 8 or even 11 trains take an enormous amount of compile time in the June compiler version. Figure 4.4 also shows that the January version of the compiler is still slightly faster in most cases than the August compiler version.

The target C code produced by each compiler version is shown in Figure 4.5. It reveals that the June version produced the most compact code but the compactness is not incommensurate with the longer compile time (cf. Figure 4.4). In fact the January compiler version behaves a bit worse w.r.t. compactness than the August compiler version. The main reason for slightly more C code are additional guards and variables for conditionals that were introduced to aid the debugging process. These extra variables do not alter the semantics of the generated code. Further versions of the compiler should clean up redundant artifacts using standard compiler techniques, i. e., *copy propagation*.

Figure 4.6 shows the normalized compile time for all compiler versions. It reveals an interesting point. That is, the relative compile time per line of code of the August and January compiler versions are effectively (bounded by a) constant where for the June compiler version it still increases. It may be inferred that the compile time of the June compiler version increases more than linearly with the model size.

This hypothesis was confirmed by code inspection where code parts in the June compiler version which had exponential complexity where discovered and could be removed during the project.

Figure 4.8 compares the code size of all models, the regression test models and the railway controller models. It can be deduced from that figure that the railway controller models are quite a lot larger than the regression test models, nearly larger by one power. The figure also reveals that both controller models are of comparable size although the 11 train controller is larger in the end.

Still Figure 4.9 shows that for the 11 train controller the June version of the compiler took almost the double time per line of code compared to the 8 train controller. Again this is due to exponential complexity of the compilation algorithm of the June compiler version.

Figure 4.10 and Figure 4.11 exclude the June compiler version and just show the results for the latest August and January compiler versions to emphasize and compare the latest compiler improvements. It can be seen as a zoomed-in variant of Figure 4.7 and Figure 4.9. These figures reveal that the January compiler version takes a little longer for the large models but the normalized performance values are still quite dense and comparable.

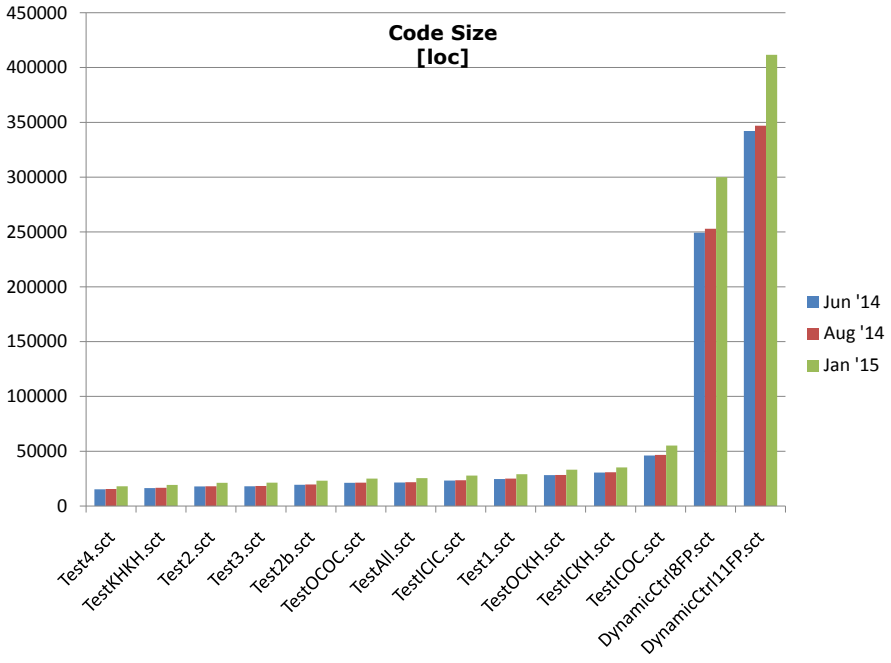


Figure 4.8.: Code size for rail regression test models and final dynamic SCCharts controller

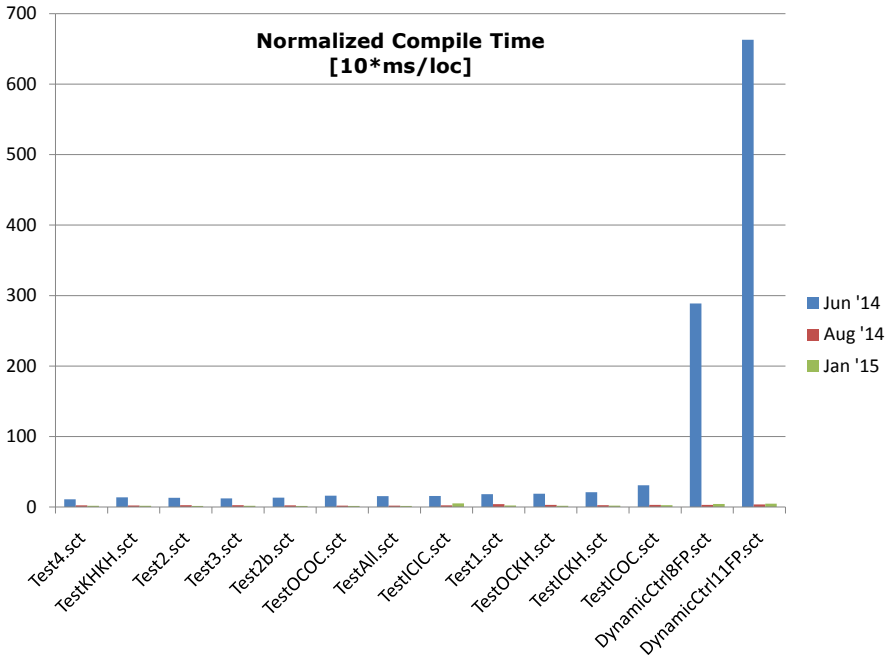


Figure 4.9.: Normalized compile time for rail regression test models and final dynamic SCCharts controller

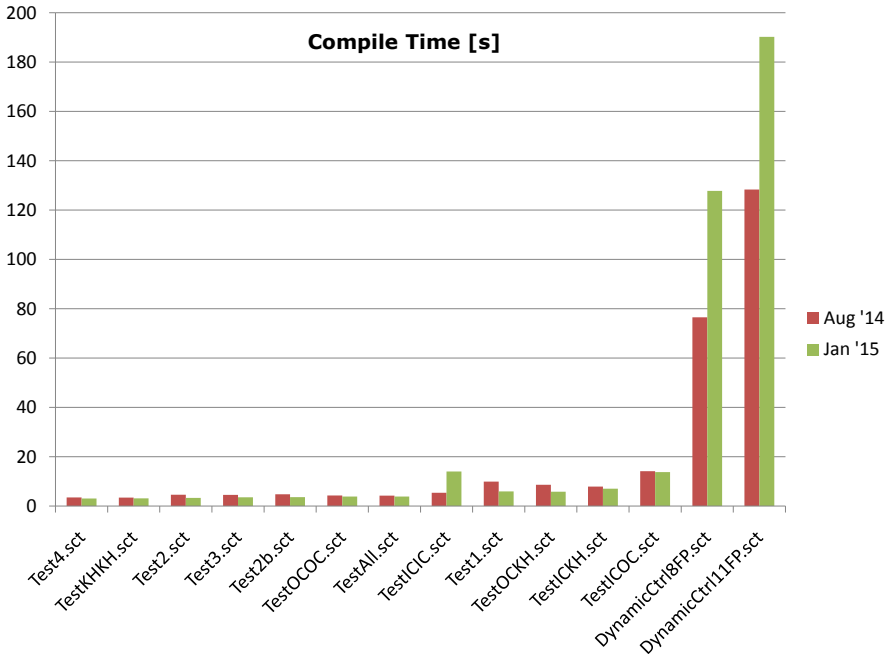


Figure 4.10.: Compile time for rail regression test models and final dynamic SCCharts controller (except June compiler version)

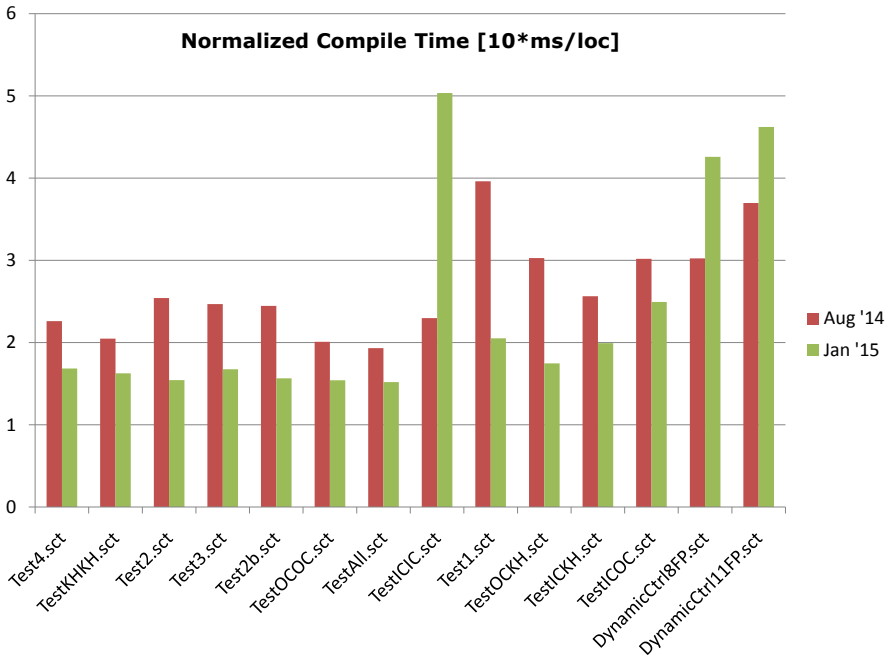


Figure 4.11.: Normalized compile time for rail regression test models and final dynamic SCCharts controller (except June compiler version)

4.2.1. Remarks on Compiler Performance

It is understood in computer science that execution times of linear complexity are desirable. Nevertheless, while working on the compiler part of the rather large academic project KIELER [vHFS11], it became apparent that worse complexity might not always be as obvious as one might suppose. Moreover, the main focus of several implementations was on correctness and not particularly on execution time efficiency. However, this oversight became a serious disadvantage when working with models of larger sizes, i. e., the railway model. Therefore, we recommend to be wary of the following circumstances in large, team-driven, and eclipse-based projects.

Obfuscated Bad Code

KIELER comprises many components and plugins which were written by a whole team of students and assistants. Many have left the university since the beginning of the project and typically program code gets old or obsolete over time. However, even with rather large code changes one wants to reuse as much code as possible. One way of doing this in a neat way are Xtend[3] *extensions*. An extension enhances the functionality of a given object and looks like a simple method call. As a consequence the code stays readable and other programmers can easily understand the meaning of a particular extension providing a meaningful naming scheme is present. For example Listing 4.1 shows an old code snippet from the abort transformation rule in the SCCharts compiler. Most programmer will understand this line instantly. The extension createRegion will create a new region with the name GENERATED_PREFIX + "Ctrl" in the state state. Then second extension uniqueName makes sure that the name is unique and does not create a name collision. As the XTend framework is already used in almost every SCCharts transformation, exploiting the extension feature came naturally.

Listing 4.1: XTend Extension Example

```
1 val ctrlRegion = state.createRegion(GENERATED_PREFIX + "Ctrl").uniqueName
```

Although this facilitates the readability in large projects immensely it involves the danger to worsen the performance unexpectedly because the used code is actually hidden and may be more complex than thought or even badly written. In addition to that the program code may be obfuscated even more as the extensions themselves may also use other extensions. Thus, even though the intention of the programmer is well understandable, the comprehension of the actual implementation gets worse.

Furthermore, Xtend comprises another in most cases useful feature *dispatch*, which allows the definition of methods for specific parameter lists. This introduces another danger: Although the depicted code in Listing 4.1 is without problems in this example, the uniqueName functions comes in three dispatch variants for regions, states, and VOs each. In two out of three cases the method performs well. However, in the VO version the complexity rises extremely because a search over the whole model takes place. This

[3] http://www.eclipse.org/xtend/

made the compilation of large models unbearable and even though only fairly simple techniques were used, the obfuscation of the code made the debugging rather hard.

Combined, we do not condemn the usage of those extensions as they provide a more than useful way to retain readability in such large project with a fluctuating team, but we strongly advise to use them with care and perform code reviews on extensions that are widely used even more thoughtfully than usual.

Use of Ecore Implementations

As KIELER depends heavily on the Eclipse Modeling Framework (EMF)[4] and its model structure, it naturally uses the data structures of EMF. Hence, one might conclude that an *EList* is just a specialized version of a standard Java *List* and that it will perform similarly with respect to execution time. This is not always the case. Depending on the actual *ecore* settings of involved classes, different implementations of EList are used. Thus, although similar by name, the performance of the data structures may differ a lot. Of course the usage of EMF data structures is mandatory in the models but should be avoided in large list operations. Accumulating large amounts of data in standard lists and copying them in one step to the EList structures increases the performance a lot in our transformations. Furthermore generic ecore object calls such as -empheAllContent are a great aid to navigate through models. However, one should evaluate carefully when to use generic content calls that traverse through the whole model instead of implementing a specialized version that only processes the parts of the model that are of particular interested. As mentioned in the uniqueName example in the section before, unnecessary traversal through a model of large size is not desirable. To shorten such model searches even more, we implemented a lot of caches inside the transformations to save execution time. If possible we advise to do as few searches on the model as possible without compromising the SLIC approach.

Visualization of Intermediate Results

While modeling with KIELER we generally follow a textual modeling approach with transient graphical automatic diagram synthesis as proposed elsewhere [SSvH13]. Usually this gives the modeler an overview of the actual model as well as the result of intermediate compilation steps if desired. However, in case of large models such as the railway controller this way of working seems to be unproductive as the automatic layout and redrawing of the model cost vast amounts of time. Thus, the visualization of the model was deactivated completely most of the time and only the final generated C code was processed further. It should be discussed further how to improve the textual modeling approach for large models.

[4] `http://www.eclipse.org/modeling/emf/`

4.3. Tick Function Performance

Despite the 450,000 lines of synthesized C code, the tick function performs quite well. The tick function alone takes about 2.3 ms to execute the compiler logic on an Intel i5 with 2.4 GHz (also visible in Figure 3.9b in Section 3.3.3). The overall loop—executing commands issued by the SCCharts controller, polling the new railway state and persisting the updated information—is done in less than 20 ms.

As described by Höhrmann the railway hardware polls new information on reed contact states only every 10 ms [Höh06, p. 45]. Due to hardware error tolerance handling this results in a minimum time difference between any two events on the same contact of at least 660 ms. The tolerance handling in form of a hysteresis function needs 20 ms (two cycles of 10 ms) to detect a closed contact and afterwards 640 ms to detect an open contact again (64 cycles). Thus the loop time of 12 ms average is more than sufficient to ensure that every railway event (regarding a single contact) is consumed in a separate SCChart tick.

5. Survey Results

As final task the participants of the railway project were asked to fill out a survey (Appendix A). The survey is divided into three parts:

1. **Language General Aspects:** In this part the survey asks general questions about SCCharts and asks to compare SCCharts to other languages. The results are presented and discussed in Section 5.2.

2. **Language Feature Aspects:** In this part the survey asks questions about SCCharts features and there relevance. The results are presented and discussed in Section 5.3.

3. **Tooling Aspects:** In this part the survey asks about feedback for the SCCharts implementation in the KIELER framework. The results are presented and discussed in Section 5.4.

5.1. Survey Setup

The survey was given to the participants at the end of the project. All seven participants filled out the survey anonymously and independently. All seven participants of the project returned their survey.

In general in all questions that compare different languages the synchronous and model languages taught at the department were chosen, namely SCADE, Esterel, SyncCharts, SCCharts, and Ptolemy. Additionally, two mainstream imperative languages, C and Java, and one functional language, Haskell, were selected.

5.1.1. Future Work

It is noteworthy that seven is not yet a significant survey sample size. However, we plan to improve and use SCCharts further in upcoming projects and are going to perform similar surveys which will be compared to the results of this project's surveys.

5.1.2. Participants

All participants were first or second year master students and had advanced knowledge of modeling, classical, and synchronous languages. In particular Ptolemy, SyncCharts, Esterel, and SCCharts as well as C, Java and Haskell were well understood. However, SCADE was mostly unknown to the audience and hence not listed in this evaluation.

5.2. Language Aspects

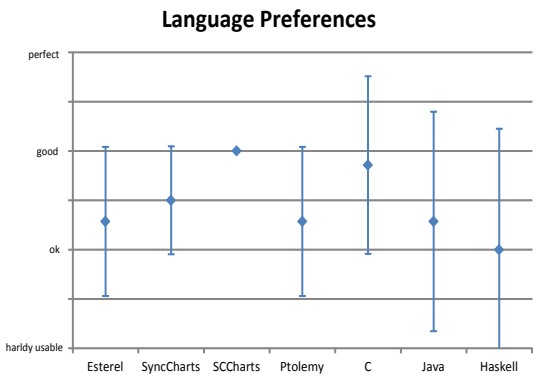


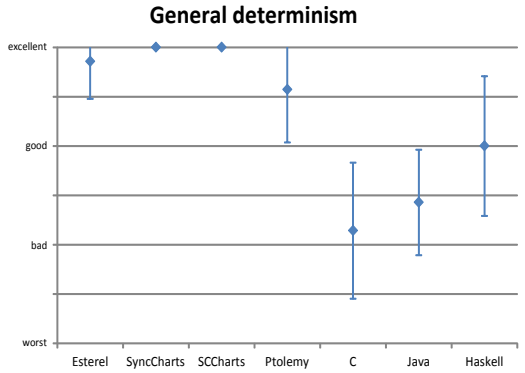
Figure 5.1.: Language preferences

As a general question the team was asked to specify which language they deem well suited to build a railway controller for the model railway installation. Figure 5.1 depicts the results.

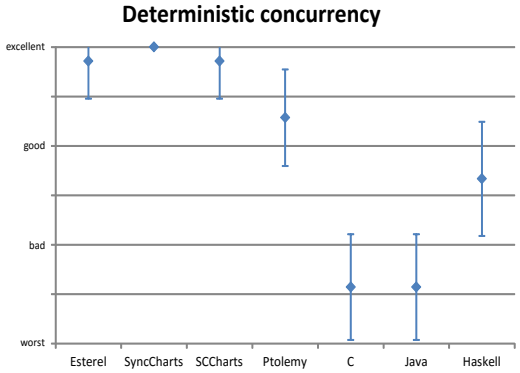
SCCharts were rated better than the other synchronous languages and is comparable with C, which is rated higher than Java and Haskell. Obviously, C is a natural pick as the railway API of the third version of the railway installation is also written in C. Hence, it is a good sign that SCCharts was chosen by the students to be comparable with C.

5.2.1. Deterministic Behavior

Figure 5.2 shows the survey results of the question about how easy or hard it is to archive deterministic behavior with each of the given languages. More specifically, Figure 5.2a depicts results for the question about the overall capability of each language to gain deterministic programs. The question is, how much effort do modeler or programmer need to invest to reach strict determinism in his or her program and are these ways well understood. Therefore, systems that react as *black box* or heavily depend on I/O interaction may compromise the understanding of the semantics and ultimately the determinism. Figure 5.2b narrows the broader determinism question specifically w.r.t.



(a) General determinism



(b) Deterministic concurrency

Figure 5.2.: Deterministic behavior

concurrency parts of the system. The subsequent question here is how much effort is necessary to avoid *race conditions*.

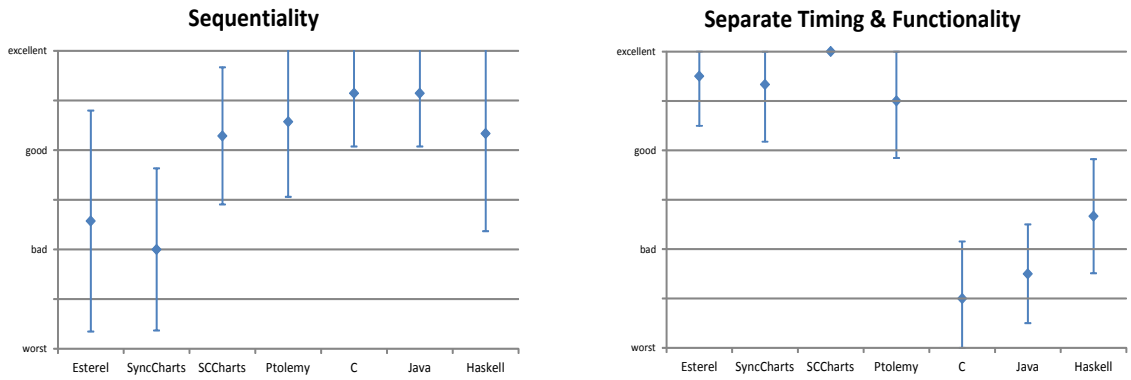
Naturally, the results of the answers show that achieving determinism was voted to be easier implemented using synchronous languages. The reason is that synchronous programs behave deterministically by design. Especially at avoiding race conditions in concurrent threads they were rated a lot better than classical imperative languages. On the one hand, archiving deterministic behavior is the main benefit of synchronous programs. The rules imposed by the MoCs are easy to comprehend and help the modeler to create *constructive* programs. However, on the other hand, as a direct result the set of valid programs in synchronous languages is more restricted compared to traditional languages which is the main drawback of synchronous programs. But for safety-critical systems it is reasonable to vote against an unrestricted set of programs because it is essential to be able to rely on deterministic behavior.

Note that Figure 5.2a is quite comparable to Figure 5.2b which reveals that achieving deterministic concurrency is the key enabler for achieving deterministic programs in the end.

5.2.2. Programming Paradigms

In the question about *sequentiality* (see Figure 5.3a) the participants were asked to rate how easy it is in each language to express sequential behavior in a program. Naturally this is a strength of sequential programming languages. Its also a common drawback for synchronous programs where programmers often tend to run into causality issues when trying to express sequential variable value changes within one reaction computation. To close this gap, one advantage of SCCharts over classical synchronous languages is the combination of the synchronous MoC with the imperative sequential programming paradigm to overcome this common drawback.

The results of the survey show that the participants also rated SCCharts en pair



(a) Sequentiality

(b) Separate timing & functionality

Figure 5.3.: Programming paradigms

with traditional sequential programming languages and distinctively better than Esterel and SyncCharts which represent other synchronous languages. It is noteworthy that at the time of the project Sequentially Constructive Model of Computation (SC MoC) capabilities such as instantaneous loops in the priority-based compiler and joins for schizophrenic behavior were not fully implemented in the KIELER SCCharts compiler and that the rating is above good nevertheless. A more complete implementation of the proposed SC MoC [vHMA+13] should improve this aspect even more.

As already mentioned in Section 1 real-time embedded systems are often *safety-critical*. In such a system it is mandatory that the correct function of a needed task is independent of the actual timing of the system. It is only necessary to prove that the hardware in the end can compute a reaction (tick) timely which can be asserted using worst case reaction time analysis [BTvH08]. As synchronous languages are also designed to separate timing and functionality, they handle this task naturally well. On the contrary, classical programming languages often heavily depend on the implementation with respect to timing and are restricted to specific systems.

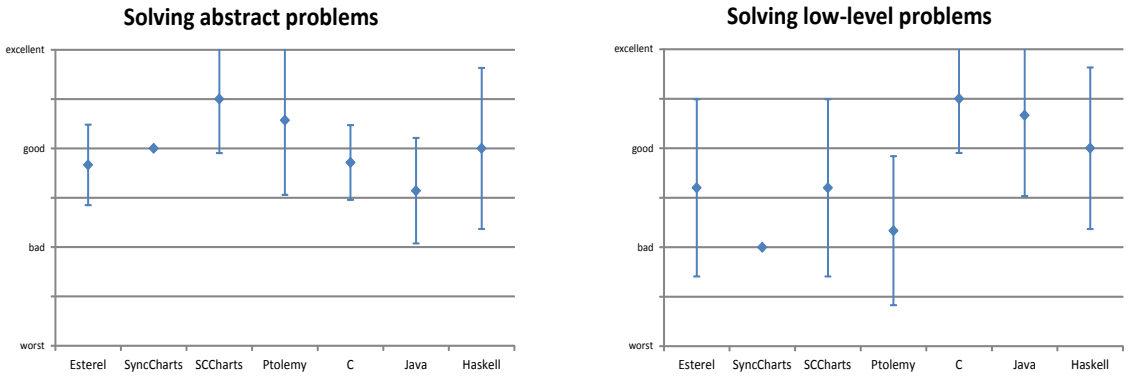
This is also reflected in the answers of the students (cf. Figure 5.3b) which also had to choose for several parts if they want to implement these on the SCCharts or the C code level.

Overall it is worth to mention that SCCharts is the only language from the set of given languages that is voted to be well usable for expressing sequentiality and at the same time still to separate timing and functionality. This combination can be seen as a main benefit for SCCharts.

5.2.3. Problem Solving

Figure 5.4 shows the results of the question which languages perform better in solving *abstract* (Figure 5.4a) and *low-level* (Figure 5.4b) problems.

In the case of abstract problems SCCharts and Ptolemy were voted to perform best



(a) Solving abstract problems

(b) Solving low-level problems

Figure 5.4.: Problem solving

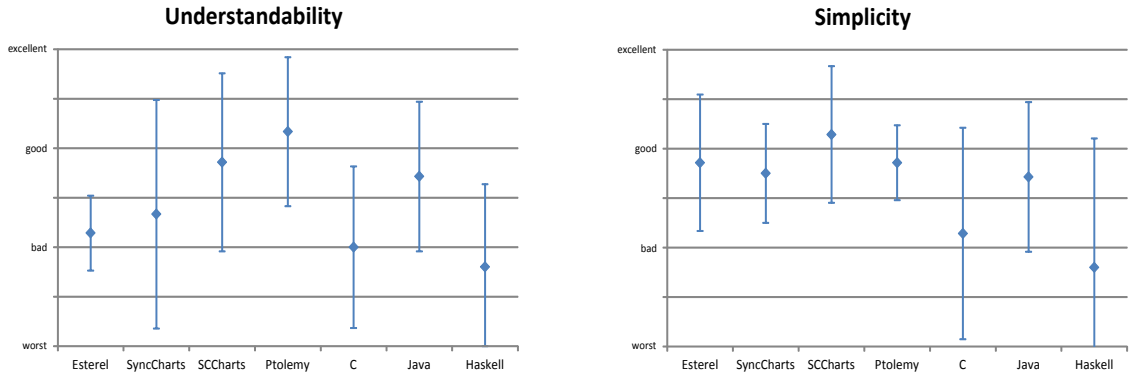
(cf. Figure 5.4a). By design these languages emphasize the modeling aspect of a problem without the need of the modeler to get deeply involved in technical details. Although this is also true for other languages, handling this aspect seems to be easier in SCCharts and Ptolemy.

Contrarily, solving low-level problems was voted to be more difficult in the languages that performed well with abstract problems (cf. Figure 5.4b). Naturally, classical programming languages are better suitable for low-level tasks. Nevertheless, as explained in Section 2.4.1 SCCharts is capable to handle low-level tasks with the implemented hostcode and function call methods. As described in Section 3.3 the SCCharts hostcode system is sophisticated enough to control the model railway installation and should suffice for smaller problems. However, we advise a revision of host code tasks are considered in SCCharts in order to ease solving low-level problems tightly integrated in SCCharts for the future.

5.2.4. Language Difficulty

Figure 5.5 depicts the student's ratings for difficulty to comprehend and work with SCCharts models. More specifically *Understandability* questions how easy it is to read and understand a program written in a certain language where the program may be written by the reader itself or by some other person. *Simplicity* asks how easy it is to learn a certain language in order to be able to fully utilize its features to build comprehensive programs or models. Understanding models and the simplicity of a language play crucial rules when working in teams, discussing processes with non-technical staff, and certification authorities. It is mandatory to keep models comprehensible and the functionality explainable to others. It is also preferable to maintain good understandability with growing models.

Figure 5.5a shows the results to the question how easy it is to, e. g., get an overview of large statecharts or projects. Here, SCCharts performs better than classical synchronous



(a) Understandability

(b) Simplicity

Figure 5.5.: Language difficulty

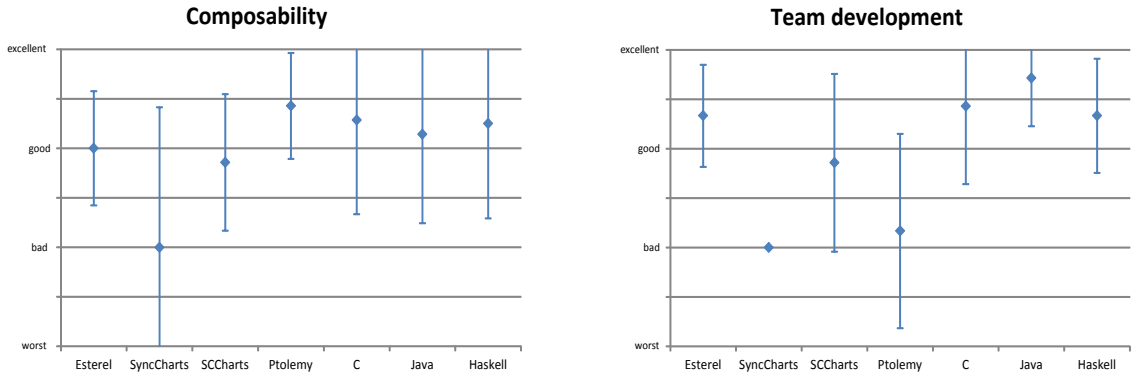
and classical imperative languages. Only Ptolemy got a better rating than SCCharts. Ptolemy has a very clear and lean graphical syntax which is very intuitive because of its data-flow nature. Also interfaces are mostly visible in Ptolemy which emphasizes the communication between model components. It is a quite good indication that SCCharts was rated comparable to Ptolemy w.r.t. the understandability aspect.

Simplicity again, determines how easy it is, e. g., to learn a language and understand the semantics of new constructs. Figure 5.5b tells that in this field, SCCharts was rated better than the other synchronous languages and distinctively better than classical languages. Even though SCCharts comprises a lot of extended features, each feature is based on a very small set of core constructs. It follows that its quite easy to first learn only about these small set of core constructs and then widen the view and subsequently learn about extended features that build upon each other. Additionally the stepwise compilation tool chain of the KIELER compiler aids the modeler to reconstruct complex features and also inspect language features in detail for certain usage of these features in models. This is even more supported by being able to simulate intermediate models in order to inspect also their dynamic behavior w.r.t. certain language features.

Understanding the SC MoC is sufficient to work with SCCharts. However, as shown in Section 3 only a limited subset of the extended features of SCCharts were used by the team which also may contribute to the simplicity rating. This also shows that it is possible to limit the set of learned and used features. Hence, the learning curve for SCCharts can be claimed not to be steep.

5.2.5. Modularity

Figure 5.6a shows the answers for the question of composability. Composability asks if a sub system implementation in this language that has certain meaning in one context will keep its meaning in any other contexts (e.g. when referenced). This a major drawback of Esterel, SyncCharts and also SCCharts compared to other programming languages



(a) Composability

(b) Team Development

Figure 5.6.: Modularity

because it is easier to run into causality problems when composing sub systems. More specifically a sub system cannot be seen as a black box for these languages because it may introduce dependencies between its inputs and outputs which in the worst case may lead to dependency cycles for the overall system. Of course for all languages including the general purpose languages one needs to take special attention to global signals or global variables as these often destroy composability.

The answers support these general observations. And because composability is one key enabler for team development, these results also correlate with each other.

As explained in Section 2.3 we extended SCCharts during the railway project to incorporate its own module concept. That is, within one SCChart a state can be of type reference and during SLIC compilation this state is expanded to the referenced SCChart considering a valid signature mapping. This improvement enables team development especially for large projects that can now be split into separate parts where parts of the team can work on different SCCharts and referenced SCCharts at the same time.

Although the ratings of Figure 5.6b testify a good team development capability of SCCharts, compared to other languages the module concept still seems to lack behind. A sophisticated browseable library might help the team modeler organizing sub components of a large project. Additionally we advise to enhance the interface bindings which today must be edited manually. A third enhancement could be an instantiation concept instead of static expansion. Also, the bad rating of SyncCharts results in the incomplete realization of the KIELER SyncCharts implementation which did only include an experimental reference expansion option in its implementation. However, the enhanced standard for SyncCharts includes referenced charts that are quite similar to Esterel's module concept. Another explanation for the worse rating for SyncCharts might be that the XMI persistence representation hampers the usage of version control management. As SCCharts uses the textual description format SCT it overcomes these difficulties. Ptolemy did also get only bad ratings for team development although it has a library concept and it allows for a static referencing and instantiation modularization. A main disadvantage of Ptolemy that may have influenced the ratings in this category may have been a more tooling aspect of browsing limitations for larger team models. Another disadvantage of Ptolemy is also an XMI persistence representation and the above described resulting problems with a version control management.

5.2.6. Project Revisions

Figure 5.7a shows the results for the *maintainability* rating. The maintainability aspect determines how easy it is to make changes to an existing program. This correlates with readability but focuses on older programs that have to be maintained for a longer period. This is a crucial aspect especially for embedded systems which have a quite long life-cycle of often 10-20 years. Subsequently, software for these systems have to be maintained also for a long time period.

Overall SCCharts were voted nearly as good as other languages. The direct comparison with SyncCharts indicates that is easier to maintain the textual editing language of SCC̃harts than the graphical syntax of SyncCharts. The comparison with Esterel indi-

cates that the graphical visualization of SCC̃harts helps w.r.t. to the readability aspect in order to get an overview and improve navigation of/in an existing model.

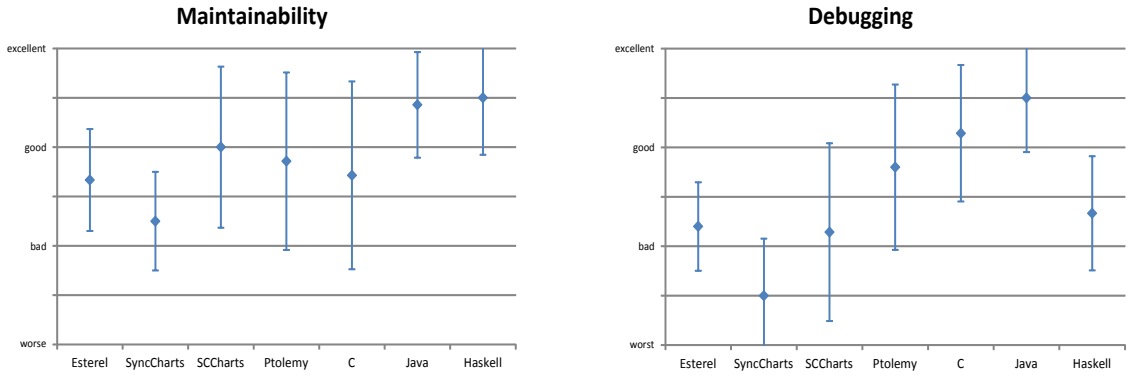
With respect to *debugging* possibilities the survey showed (Figure 5.7b) that all synchronous languages perform worse than their classical relatives. The participants of the survey explained the bad rating of SCCharts in the free text answers of the survey with cyclic dependency and mutual exclusion problems, broken extended features, and missing overview in large SCCharts if programs were not schedulable.

To address these issues several improvements are planned. Besides new pragmatics features such as lazy loading of SCCharts references and adaptive zooming depending on the hierarchy level, backward information propagation of (intermediate) compilation results should provide the modeler with important information about dependencies on modeling level.

5.3. SCCharts features

Figure 5.8 shows the importance of specific extended SCCharts features w.r.t. the railway project rated by participants. Figure 5.8a shows essential core SCCharts elements such as different transitions, triggers and effects. Especially the huge gap between terminations and strong and weak aborts is interesting. Termination transitions belong to the core SCCharts language and are easy to transform. A join is executed when the corresponding superstate terminates. Contrary to terminations aborts are extended features and need to be transformed using terminations. The generic abort transformation rules increase the complexity of the statechart. At the time of the project the abort transformation introduced some errors in conjunction with other features. Therefore, only a few aborts were used in the final controller when this feature transformation was fixed.

Figure 5.8b lists additional transition features and actions. The railway team made intensive use of entry actions because they result in more compact diagrams. The large



(a) Maintainability (b) Debugging

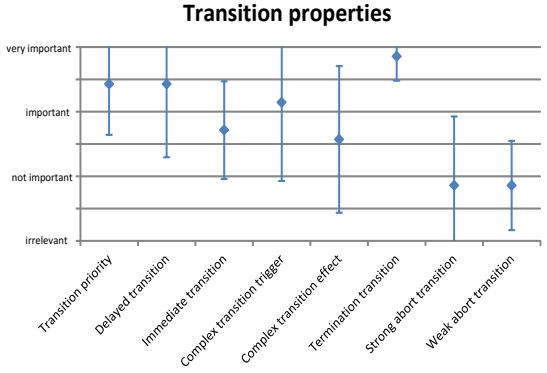
Figure 5.7.: Project revisions

amount of hostcode and train combination effects create very large transitions if used in transition effects. Placing them in entry actions is semantically identical but improves the readability of the SCChart. Providing the same compactness in transitions might be a future improvement. History transitions were not used in this project and claimed to be irrelevant by the participants. The same holds for deferred transitions and suspensions.

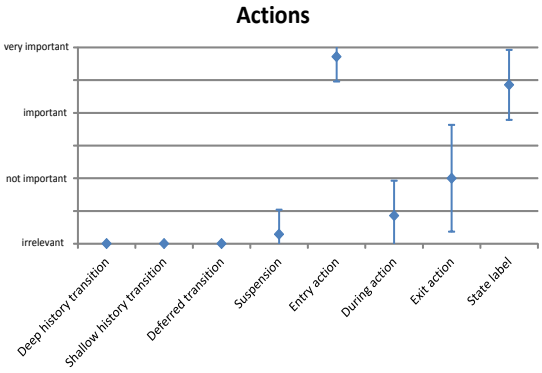
Concurrency and declaration features are combined in Figure 5.8c with concurrent regions rated very important. Hence, local declarations are also used frequently. Naturally, boolean and integer data types scored higher than their relatives because they are used by the railway API.

Figure 5.8d shows complex state, signal and the new features introduced in Chapter 2. According to the team complex final states would have been important for convenience but resulted in bad compile times. However, semantically they were not mandatory. Here, further improvements are strongly advised.

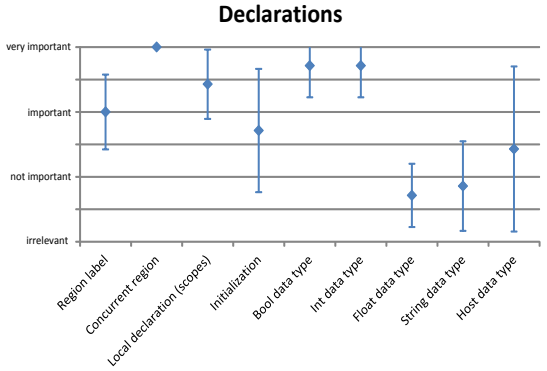
As already stated in Chapter 2 the two new main features, namely references SCCharts and arrays, were mandatory and hence naturally were rated very important.



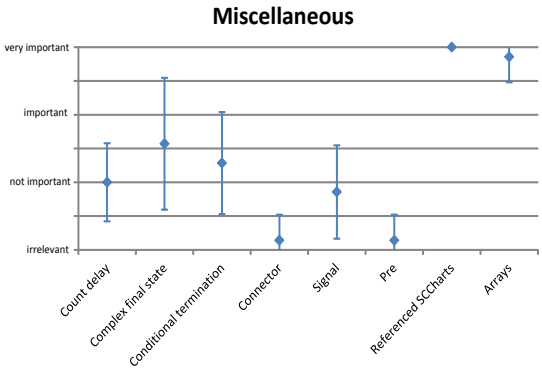
(a) Basic core and extended features



(b) History and local action features



(c) Concurrency, declaration and data types



(d) Additional extended features

Figure 5.8.: SCCharts feature importance poll

5.4. Modeling Aspects

In the next set of questions in the survey (Appendix A) the participants should rate the overall quality of the SCCharts tool chain in the actual KIELER implementation.

Figure 5.9 depicts the score of the modeling tools at the beginning and in the end of the project. According to the participants the tools were hardly sufficient to build a sophisticated railway controller at the start of the project. However, we were able to improve the performance of the overall tool chain drastically which resulted in the big improvement in the final rating. This corresponds to the tight team feedback mentioned in Section 3.4 and problem solving depicted in Figure 3.10b.

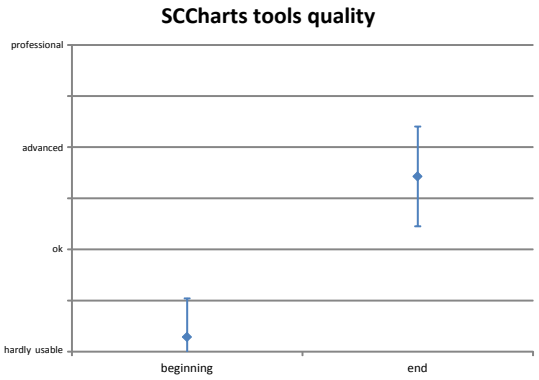
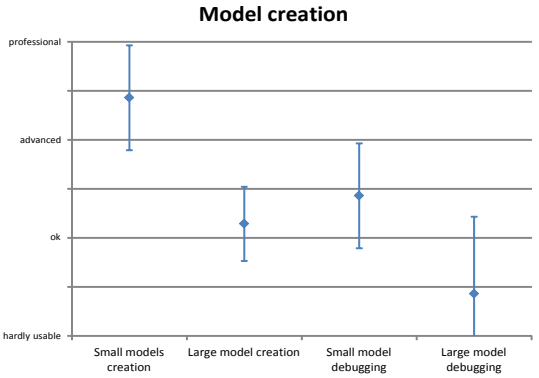


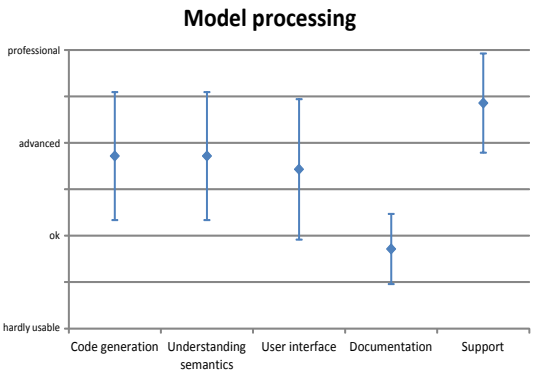
Figure 5.9.: Tools quality

Figure 5.10a illustrates that the participants rated the possibility to create models positively with smaller models almost professional and larger models average. Nevertheless, debugging these models seems to be a bigger challenge. While debugging smaller models is still possible with the actual tool chain, the figure implies that this is not the case for larger ones.

One of the main reasons for this were hard to find concurrent dependency cycles but also extended layer compiler problems. These may still be relatively easy to find and fixed in small programs. However, without sophisticated tools to help the modeler to identify these low-level compiler issues on modeling level, finding these errors becomes educated guess work. This was also confirmed in the free text answers of the survey.



(a) Model creation & debugging



(b) Tooling aspects

Figure 5.10.: Modeling aspects

Additionally, as described in Section 4.2 working with large models such as the railway controller (compare sizes in Section 3.3) introduces a new level of statechart complexity for the KIELER tool chain. Both the transformation rules as well as the transient visualization and automatic layout mechanisms reached their limits during the project. Further profiling, evaluation of potentials problems, and improvements of the tool chain is advised.

Figure 5.10b depicts additional main aspects of the current KIELER tooling w.r.t. SCCharts. The code generation, understandability of the semantics, and the user interface were all rated sufficient for the given task. The results presented in Chapter 3 and Chapter 4 showed that all set goals, especially controlling up to eleven trains concurrently, are met and furthermore are remotely controllable. However, the documentation part of the tooling still needs to be improved. Finally, the figure shows the high support rating which also underlines the remarks of Section 3.4 that the close feedback loop between the project participants and the supervisors was appreciated and frequently of use.

As the evaluation of the language aspects in Section 5.2.4 showed, understandability of SCCharts scored already quite well. However, since overview of a model does not solely depend of the understandability of the language itself, we believe that we can also improve the understandability of particular models further through tooling aspects. Therefore we plan to implement pragmatics features such as different semantic zoom levels and lazy loading for referenced SCCharts in the KIELER SCCharts editor[1] version 0.11. Further consulting with the pragmatics team of the group is advised.

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

6. Related Work

6.1. Model Railway Installation

The current version of the model railway installation, which was used in the projcet and described in Section 1.2, was built by Stephan Höhrmann in 2005. The design and implementation of the power electronics as well as details of the communication protocols are documented in his diploma thesis [Höh06]. He also briefly describes the evolution of the model railway installation and the problems with previous generations.

Modeling the interaction of trains on the tracks was previously mostly done using petri net models. Jürgen Koberstein and Oliver Schmitz designed petri net models for several parts of the installation [Kob01].

Figure 6.1 shows the petri net model of the Kicking Horse Pass with the turnout track in the middle part. In combination with several other models for switching points and straight lanes a complete controller was constructed.

The initial version of the model railway installation featured a slightly different track layout. In particular it was missing the turnout track in the middle of the Kicking Horse Pass. This first generation was also modeled with petri net models by Wolfgang Hielscher et. al. [HURK98]. The model shown in Figure 6.2 controls the original Kicking

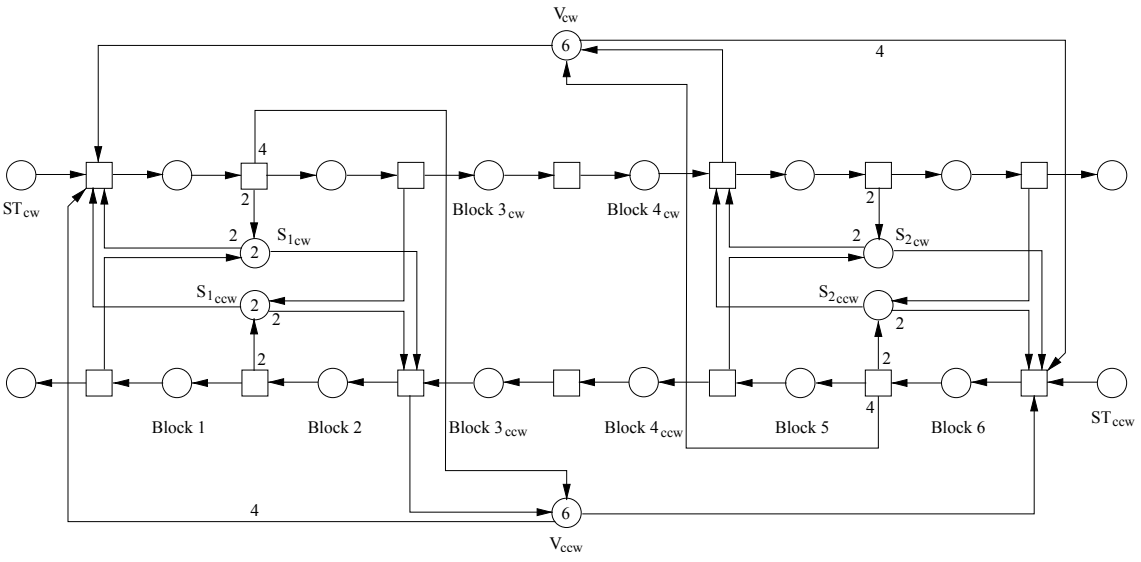


Figure 6.1.: Deadlock-free model of pass lane with turnout track [Kob01]

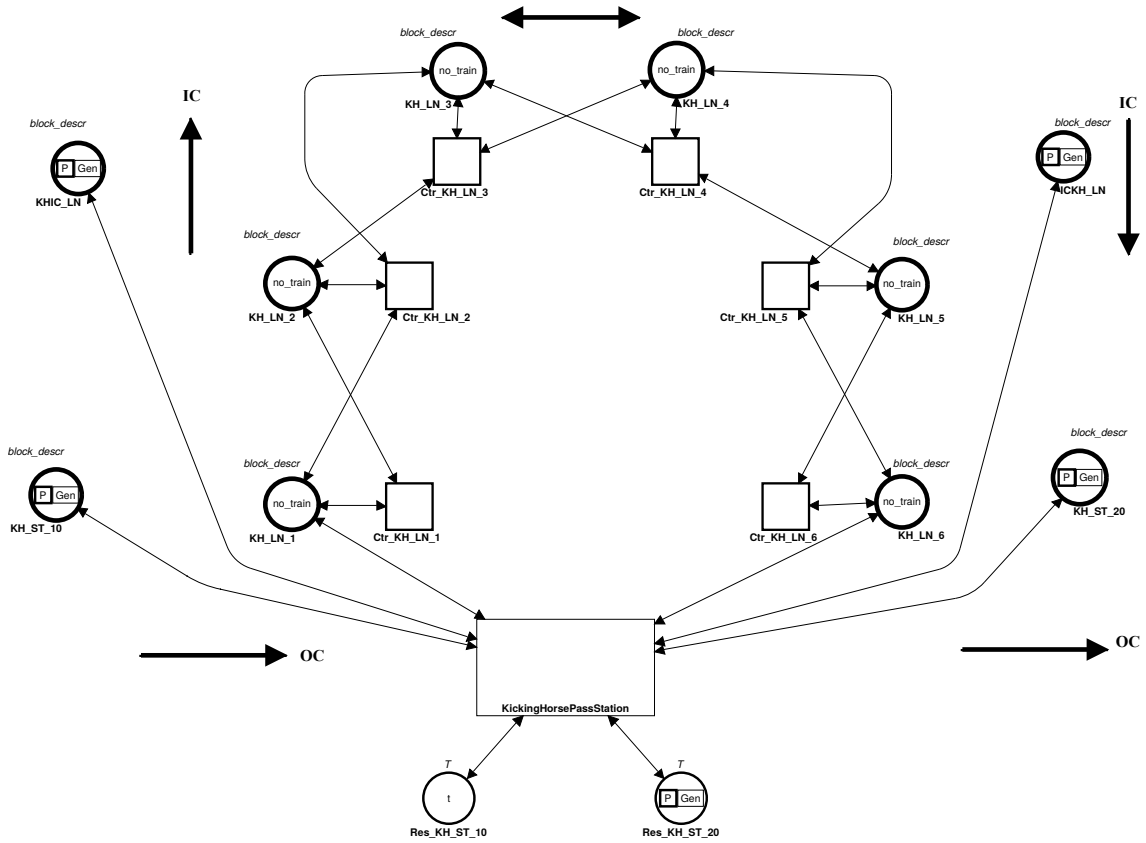


Figure 6.2.: Net model of the kicking horse pass track [HURK98]

Horse Pass without the turnout track. The model was developed using Design/CPN[1] and was used to generate a working controller for the first hardware generation

6.2. Model Railway Project 2007

In this railway project[2] the task given was to control the trains using a distributed controller that runs on TTP nodes. These TTP nodes where connected to each other using the synchronous TTP network. Each TTP node was connected to a power electronics which was responsible for a certain amount of low level peripherals like track segments, switch points or contacts. This is illustrated in Figure 6.3.

Figure 6.4 gives an overview of the software tool chain. In this project the target code running on the TTP nodes was also generated. As a high-level specification and implementation language the visual synchronous data-flow language Safety Critical Application Development Environment (SCADE) was chosen. For properly getting the generated C code onto to the TTP nodes, a special additional TTP tool chain was used (SCADElink). As a Human-Machine-Interface a general purpose LCD display was used.

[1] http://daimi.au.dk/designCPN/

[2] http://www.informatik.uni-kiel.de/rtsys/teaching/ss07/p-railway/

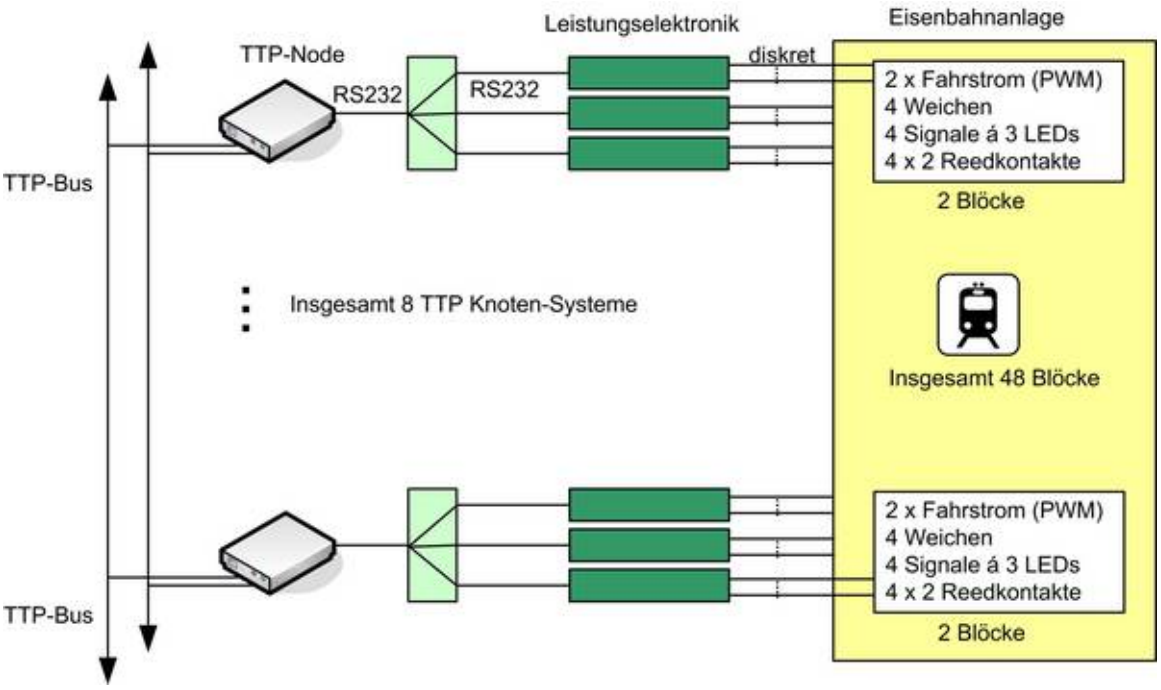


Figure 6.3.: Railway project 2007 hardware schematics [from project website]

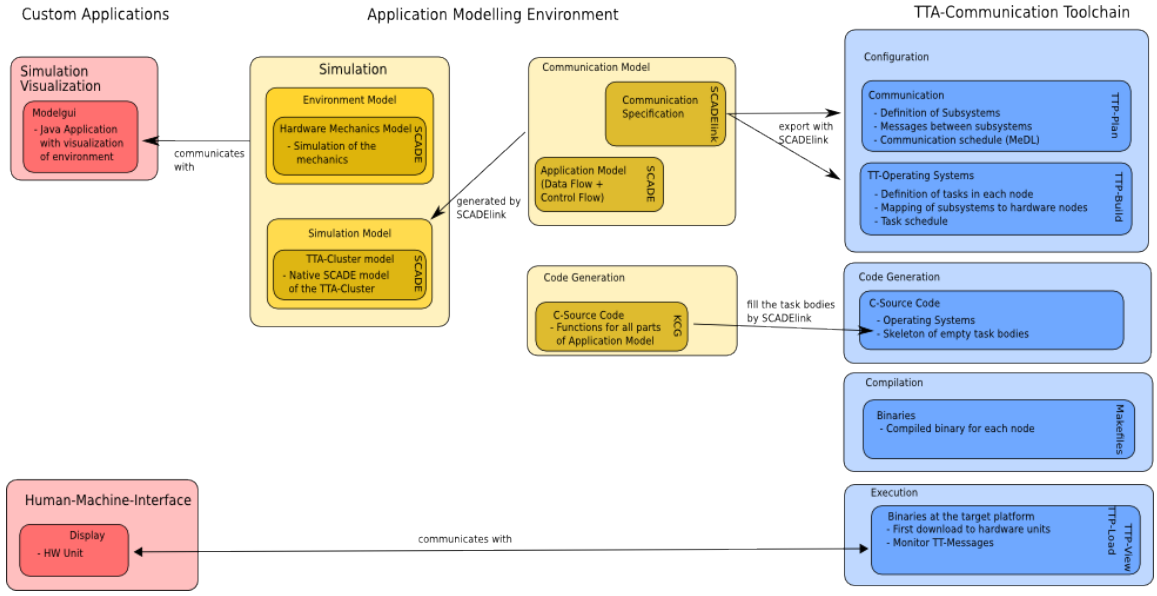


Figure 6.4.: Railway project 2007 tool chain [from project website]

Figure 6.5 and Figure 6.6 show some parts of the SCADE model from this project. The SCADE language allows to combine data-flow and control-flow (statemachines) as can be seen in these figures where data-flow is defined for each state. The statemachine control-flow parts of the railway controller in this project turned out to be quite essential to

reflect the sophisticated control logic. The data-flow parts mostly where used to collect, distribute or convert command or sensor data.

Compared to the SCCharts project the project from 2007 also validated that using a synchronous language is a good choice for modeling and implementing safety-critical systems. As the tight data-flow integration of SCADE turned out to be quite usable, SCCharts could also benefit from such a concept. Actually this is already work in progress and might be available for further SCCharts projects in the near future.

Another point learned from SCADE in this project was about the documentation. Comments could be attached to the model itself and the documentation could be auto-generated later. This helped a lot in maintaining consistency between the implementation and its documentation. At the moment SCCharts does not have such features so the current railway project didn't have any tool support for documentation. This may be a reasonable extension in the future.

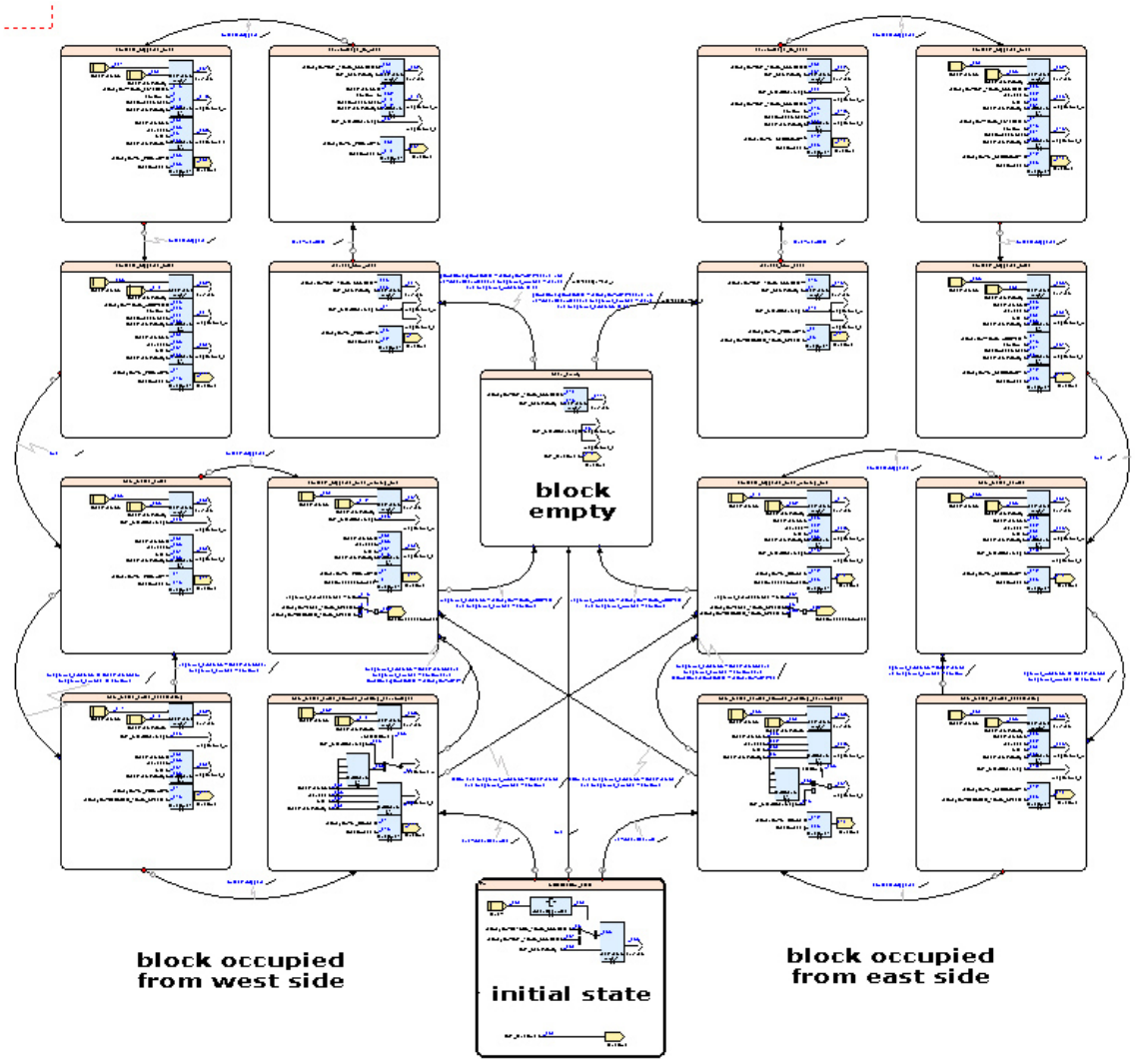


Figure 6.5.: Railway project 2007 SCADE model part [from project website]

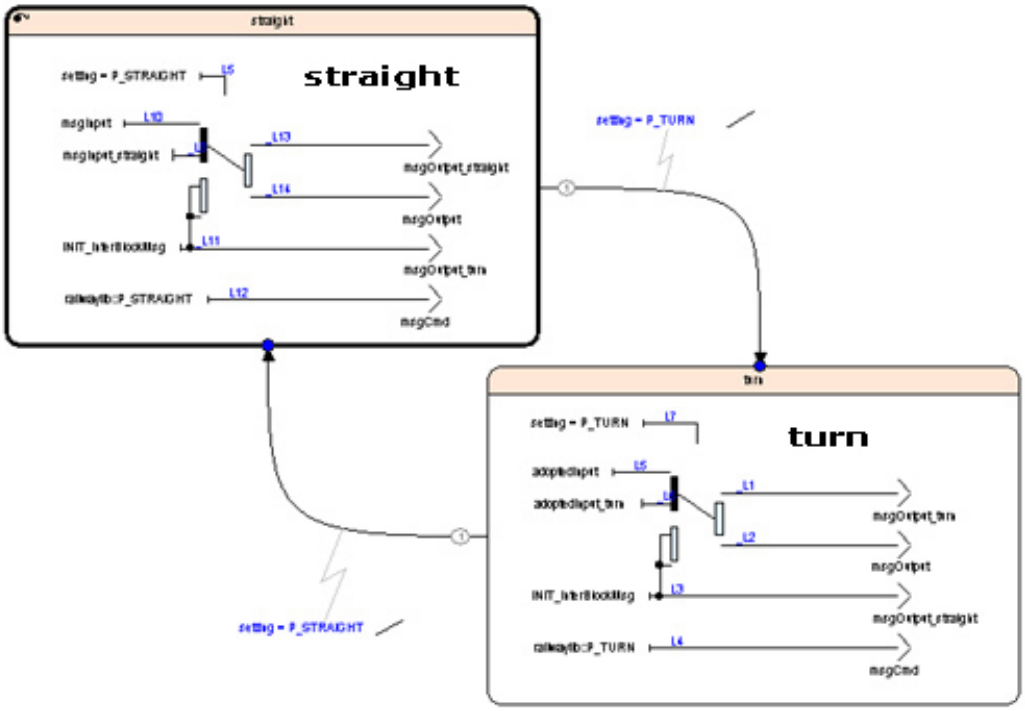


Figure 6.6.: Railway project 2007 SCADE model part [from project website]

The SCADE version that was used 2007 was first an alpha (SCADE 6.0 P3) and later a beta (SCADE 6.0 Beta1) release. According to former students that participated in this project the team had to struggle with the tool quality a lot. This was quite comparable to the SCCharts tool stability mentioned in Section 5.4.

6.3. Model Railway Project 2012

In the 2012 remake of the railway project the task was given to create a railway controller in SCADE running on a central system that was capable to control all trains over the power nodes of the railway installation. Therefore, for each railway element, e. g., switch point or signals, a SCADE node was created that hid the low-level API of the railway system and was accessible at modeling level.

For example, a node setTrack that sets the mode and the speed of a track was wrapped as shown in Listing 6.1. The node has four input ports. The first port expects a dataflow link to the railway system denoted with RS in the listing. The second to fourth ports must be connected to links that specify track, track mode and speed of the track. As a remark, all these values must be integer. However, there was also a node that translated track IDs to track numbers which could be used before the setTrack node. As seen in the listing the first four parameter of the function correspond to the expected inputs. The fifth parameter is a pointer to the output structure of this actor node. If this function gets called, it checks if the railway system got initialized and if true the call

gets redirected to the underlying railway api. Since setTrack does not have any additional output ports, only the address of railway system is copied to the outputs.

Analogously, getTrack (see Listing 6.2)reads out the mode and the speed of a specific track segment and writes them to the outputs of the node.

Listing 6.1: Wrapper code for setTrack

```
void setTrack_RailIntf(
  kcg_int RS,
  kcg_int track,
  kcg_int mode,
  kcg_int speed,
  outC_setTrack_RailIntf *outC)
{
  if (RS) {
    settrack((struct railway_system*)RS, track, (unsigned)mode, (unsigned)speed);
  }

  outC->outRS = (int) RS;
}
```

Listing 6.2: Wrapper code for getTrack

```
void getTrack_RailIntf(
  kcg_int RS,
  kcg_int track,
  outC_gettrack_RailIntf *outC)
{
  unsigned int mode = 0;
  unsigned int speed = 0;

  if (RS) {
    gettrack((struct railway_system*)RS, track, &mode, &speed);
  }

  outC->outRS = RS;
  outC->mode = (int) mode;
  outC->speed = (int) speed;
}
```

Equipped with tools to model the complete railway installation directly in SCADE, the participants of the project in 2012 decided to build smaller controller units for different tasks, namely one Top Level Controller (TLC) and several Second Level Controllers (SLCs). The TLC manages all trains with their train schedules and communicates planned train routes with the SLCs. A SLC is responsible for a specific track segments and grants (or denies) permissions of enter or leave request of the TLC to avoid collisions. The organization of the controller types and the communication routes between them is depicted in Figure 6.7.

It is not necessary that there is a SLC for each single track. Hence, several tracks were combined to a larger logical unit called *segment*. Each segment is handled by its own SLC. The different track segments are shown in Figure 6.8. When planing a rout for a train instead of 48 single tracks only the 12 listed segments must be considered. However, segments with stations are handled differently because they have parallel tracks. The controller decides automatically which station track gets assigned.

The project showed that it is essential for a modeling language to allow for decomposition of the complete complex model into straightforward sub components. Additionally

the model developed in the project used a lot of data flow but also combined with control flow parts. This showed the need for a modeling language that allows to express both, control and data flow. The project participants heavily made use of the integrated simulation possibility SCADE comes equipped with. It was a key to success to be able to excessively validate single components before composing them together. The project was delayed several times when SCADE was unable to generate code because of forbidden immediate cycles. More tool support for resolving these cycles in a more or less optimal way would have been helpful here.

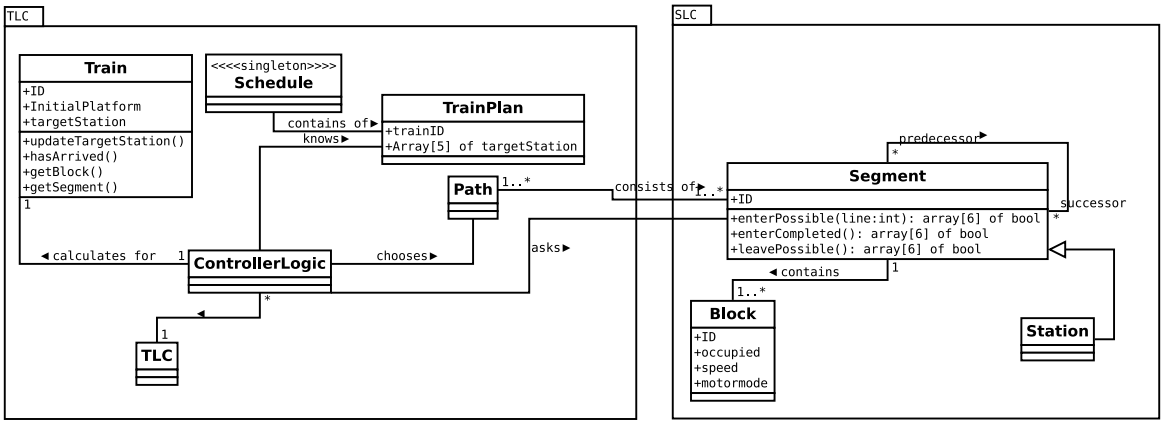


Figure 6.7.: Controller communication from the railway project 2012

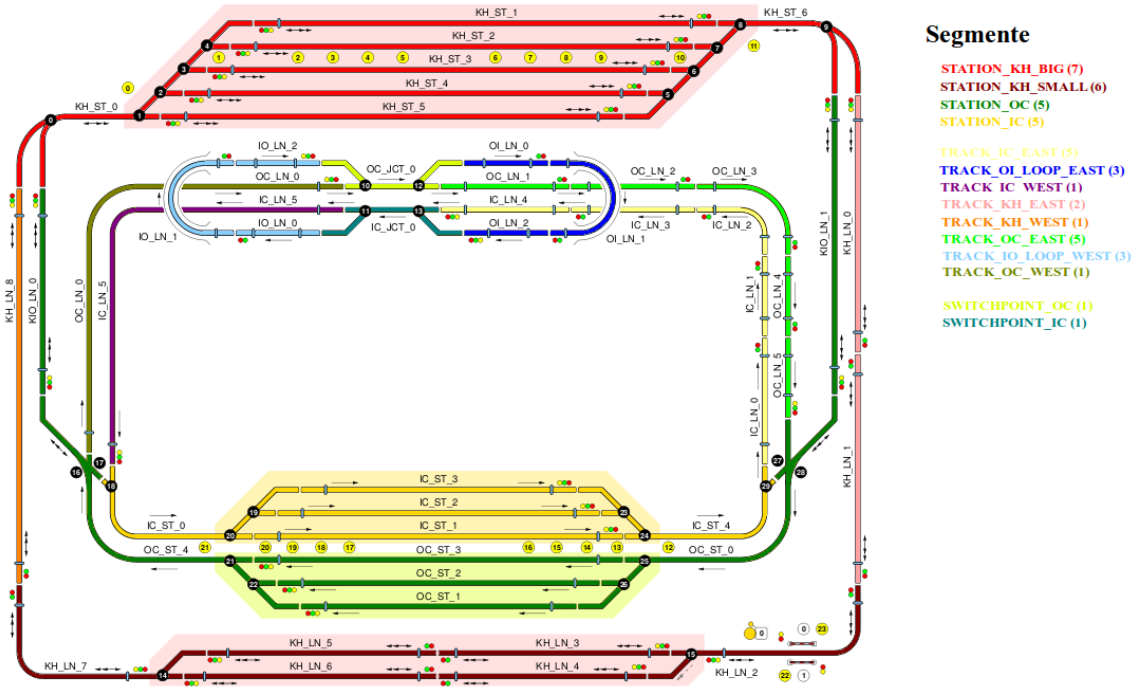


Figure 6.8.: Segment scheme from the railway project 2012

7. Wrap-Up

The railway project succeeded in developing a controller for the given railway installation using SCCharts. A co-design of SCCharts and C code together with a concurrent server-client-based user interface resulted in a safety critical and interactive system. The final controller is capable of controlling 11 individual trains remotely scheduled by multiple GUI-clients on different hardware platforms.

Section 4 evaluated the continuous development of KIELER during the railway project. A detailed compilation experiment showed the process of compilation and explains and justifies the generated results. The performance evaluation of the compiler showed that the influences of the railway project caused significant improvements. Many implementation details which had negative effects on the performance were identified, analyzed and fixed. In addition to that, some limitations of graphical modeling with large models could be stated to influence future developments. Furthermore, performance measurements of the final controller also showed the capability to fulfill realtime requirements.

To conclude the survey section, the project and the resulting final controller show that SCCharts is feasible to create mid-to-large size constructive programs within adequate creation time and efficient execution performance. In many aspects SCCharts performs as good or even better than other synchronous languages and is often en pair with classical programming languages. The participants perceived SCCharts as understandable, simplistic and maintainable. However, they underlined the need for better tooling support for debugging, composability, and team development. Here, debugging becomes increasingly hard with growing statecharts. The development environment must provide the modeler with adequate tools to combat this difficulty. Also the performance issues with respect to large model browsing should be addressed as soon as possible.

Bibliography

[And96] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[Ber00] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91.* Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. `ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf`.

[BTvH08] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems*, 2008:1–21, 2008.

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

[HURK98] Wolfgang Hielscher, Lars Urbszat, Claus Reinke, and Werner Kluge. On modelling train traffic in a model train system. In K. Jensen, editor, *Daimi PB-532: Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 83–102. Department of Computer Science, University of Aarhus, Denmark, 1998.

[Kob01] Jochen Koberstein. Realisierung eines geordneten Mehrzugbetriebs auf einer Modellbahnanlage. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2001.

[MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling scccharts—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)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, October 2014.

[MvH14] Christian Motika and Reinhard von Hanxleden. Light-weight Synchronous Java (SJL) — an approach for programming deterministic reactive systems with java. *Journal of Computing, Special Issue on Software Technologies for Embedded and Ubiquitous Systems*, 97(3):281–307, 2014.

[SSvH13] 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)*, pages 75–82, San Jose, CA, USA, 15–19 September 2013.

[vHDM+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, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013. ISSN 2192-6247.

[vHDM+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, June 2014. ACM.

[vHFS11] Reinhard von Hanxleden, Hauke Fuhrmann, and Miro Spönemann. KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE'11)*, Grenoble, France, March 2011.

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

A. SCCharts Survey

Appendix A contains the survey that was handed out to the participants of the project. To compare the results while working on the SCCharts compiler and tool chain it is recommended to also use this survey in upcoming SCCharts projects. The following list describes the version history of the survey.

Version 1.1s Simplified version, contains only the most important questions.

Version 1.1 Added new features *referenced SCCharts* and *arrays*.

Version 1.0 Initial version

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group

SCCHARTS SURVEY FORM

Rev. 1.1s

Project Name: _____ Size: _____ (# of involved people)

Project Start/End: _____ Date: _____

Start Time: _____ (current time when starting this survey)

I) About yourself

1) In which semester are you (in your current degree program)? ☐

2) What degree do you pursue? ☐ Bachelor ☐ Master

3) Which **modeling tools** have you used before?

4) Which **synchroneous/dataflow languages** have you used before?

II) Project involvement

5a) Estimate distribution of work for **your part** of the project

# of SCCharts **modeled**	
# of SCCharts **reviewed**	
# of SCCharts **documented**	
# of SCCharts **tested**	
# of **additional SCCharts-generating scripts** written	

5b) Of 100% time you spent in the project, you ...

% **modeled** with SCCharts	
% written in **host language**	
% written **additional scripts**	
% elaborated **documentation**	
% did **testing**	
% **discussion time**	
% other:	
% other:	

6) Estimate your SCCharts skills **before** this project

☐	☐	☐	☐
professional know all features and can argue for or against them, could teach others	**advanced** know most features, have some feelings for/against usage	**ok** know main features and can implement requested functionality	**greenhorn** know some features and can read most models, implement running (small) models

7) Estimate your SCCharts skills **after** this project

☐	☐	☐	☐
professional know all features and can argue for or against them, could teach others	**advanced** know most features, have some feelings for/against usage	**ok** know main features and can implement requested functionality	**greenhorn** know some features and can read most models, implement running (small) models

8) To what extent you think you could make use of dataflow (DF) for this project? (If you don't know about DF, you may skip this question)

☐	☐	☐	☐
perfect everything should be modeled with data flow	**most parts** mostly dataflow should be used, only some parts need control flow	**some parts** mostly control flow should be used, only some parts need data flow	**nothing** data flow seems not the right thing to be used at all

9) To which extent would you like to use the following modeling/programming languages **for this project**?

SCADE

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

Esterel

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

SyncCharts

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

SCCharts

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

Ptolemy

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

C

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

Java

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

Haskell

☐ **perfect** use for everything	☐ **good** use for most parts	☐ **Ok** use for up to 50% of the models	☐ things can be **hardly** modeled with this language	☐ **don't know** this language

III) Language aspects

10) Grade the following languages regarding the handling of each aspect. Use ++(excellent), +(good), -(bad), --(worse) as marks. You may skip a grade if you are uncertain. Comments about your choices are appreciated:

Problem/Aspect	SCADE	Esterel	SyncCharts	SCCharts	Ptolemy	C	Java	Haskel
General determinism Achieve deterministic behavior								
Deterministic concurrency Avoid race conditions								
Sequentiality Express sequential parts								
Composability Compose sub-solutions to an overall solution								
Solving abstract problems								
Solving low-level problems								
Understandability Overview of large projects								
Simplicity Learning curve								
Separate Timing & Functionality								

11) Grade the following language regarding the overall suitability for your project. Use ++(excellent), +(good), -(bad), --(worse) as marks. You may skip a grade if you are uncertain. Comments about your choices are appreciated:

Problem/Aspect	SCADE	Esterel	SyncCharts	SCCharts	Ptolemy	C	Java	Haskel
Maintainability Later changing programs								
Debugging Ability to debug errors								
Team development Divisibility of a large project in sub parts								
Version control support Merge & Diff								

12) Your opinion: Rate the following features of SCCharts w.r.t. their significance **for the project**:
(use the grading scheme from 9: ++(very important), +(important), -(not important),-- (irrelevant) or blank)

SCCharts Feature	Grade
Transition priority	
Delayed transition	
Immediate transition	
Complex transition trigger (e.g., (A & B) | (C & D))	
Complex transition effect (e.g., A = B&C; D = E)	
Termination transition & Final States	
Strong abort transition	
Weak abort transition	
Deep History transition	
Shallow History transition	
Deferred transition	
Suspension	
Entry action	
During action	
Exit action	
State label	
Region label	
Concurrent region	
Local declaration (scopes)	
Initialization	
Bool data type	
Int data type	
Float data type	
String data type	
Host data type	
Count delay	
Complex final state (e.g., with outgoing transitions)	
Conditional termination	
Connector	
Signal	
Pre-Operator	
Referenced SCCharts	
Arrays	

13) What were the **most challenging functionalities** to be implemented in the Project? Why?

14) Which extended features did you miss? Describe the features and explain why.

IV) Tooling aspects

15) Your opinion about the **overall quality** of the SCCharts development tools you worked with, compared to other modeling/programming environments:

At the **beginning** of the project:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

At the **end** of the project:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

16) For **specific use cases**, rate the quality of the SCCharts development tools you worked with, compared to other modeling/programming environments (at the end of the project), write down possible enhancements:

Creation/**modeling of small** models:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Creation/**modeling of large** models:

☐ **Professional** as other mature open source/commercial products	☐ **Advanced** as other smaller commercial or open source products	☐ **Ok** As beta versions of commercial software or Freeware	☐ **Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Debugging of small models:

☐ **Professional** as other mature open source/commercial products	☐ **Advanced** as other smaller commercial or open source products	☐ **Ok** As beta versions of commercial software or Freeware	☐ **Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Debugging of large models:

☐ **Professional** as other mature open source/commercial products	☐ **Advanced** as other smaller commercial or open source products	☐ **Ok** As beta versions of commercial software or Freeware	☐ **Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Code generation:

☐ **Professional** as other mature open source/commercial products	☐ **Advanced** as other smaller commercial or open source products	☐ **Ok** As beta versions of commercial software or Freeware	☐ **Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Understanding the language semantics:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

User Interface:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Documentation:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

Support:

☐	☐	☐	☐
Professional as other mature open source/commercial products	**Advanced** as other smaller commercial or open source products	**Ok** As beta versions of commercial software or Freeware	**Hardly usable** like alpha versions or private/obsolete projects

Enhancements:

_____

17) Were any timing-related problems relevant?

☐ Yes, execution time calculation for **whole program**

Was an execution time analysis for the whole program (several ticks) needed? (WCET)

Describe your solution and/or how the tooling could have helped:

_____

☐ Yes, execution time calculation for **one tick**

Was an execution time analysis for the one tick needed? (WCRT)

Describe your solution and/or how the tooling could have helped:

_____

☐ Yes, **synchronization** between ticks

Synchronizing concurrent threads at specific tick boundaries, e.g., “Is thread x at tick c in state s?” or “Who does what in which tick?”

Describe your solution and/or how the tooling could have helped:

_____

☐ Yes, namely**:** _____

Describe your solution and/or how the tooling could have helped:

_____

☐ No

18) Would the automatic display of (partial) execution time information, e.g., for regions, in the graphical diagram have been helpful?

☐ Yes

☐ No

19) Do you have any other remarks about the SCCharts tools/compiler you would like to share?

Pro:

_____

_____

_____

_____

_____

_____

_____

_____

Contra:

_____

_____

_____

_____

_____

_____

_____

_____

20) End Time: _____ (current time when finishing this survey)

Thank you!

B. Raw Controller Model Analysis Results

Transformation ID	States	Transitions	Regions	Variables	Nodes	Control-flow	Threads
SourceFiles	1628	2219	713	1006	0	0	0
REFERENCE	24414	27623	15197	1238	0	0	0
MAP	24414	27623	15197	1238	0	0	0
FOR	24414	27623	15197	1238	0	0	0
COUNTDELAY	24414	27623	15197	1238	0	0	0
WEAKSUSPEND	24414	27623	15197	1238	0	0	0
SUSPEND	24414	27623	15197	1238	0	0	0
DEFERRED	24414	27623	15197	1238	0	0	0
SIGNAL	24414	27623	15197	1238	0	0	0
DURING	24414	27623	15197	1238	0	0	0
COMPLEXFINALSTATE	24426	27631	15201	1250	0	0	0
ABORTALTERNATIVE	24627	28418	15263	1338	0	0	0
EXIT	24627	28418	15263	1338	0	0	0
HISTORY	24627	28418	15263	1338	0	0	0
STATIC	24627	28418	15263	1338	0	0	0
CONST	24627	28418	15263	1293	0	0	0
PRE	24627	28418	15263	1293	0	0	0
INITIALIZATION	24627	28418	15263	1293	0	0	0
ENTRY	99994	106764	16644	1293	0	0	0
CONNECTOR	99994	106764	16644	1293	0	0	0
TRIGGEREFFECT	118966	125736	16644	1293	0	0	0
SURFACEDEPTH	134524	151908	16644	1293	0	0	0
SCG	0	0	0	1293	126511	148842	3395
SCGDEP	0	0	0	1293	126511	148842	3395
SCGBB	0	0	0	88938	126511	148842	3395
SCGSCHEDSIMPLE	0	0	0	88938	126511	148842	3395
SCGSEQSIMPLE	0	0	0	93875	179809	211521	1