A Data-Flow Approach for Compiling the Sequentially Constructive Language (SCL)

Steven Smyth, Christian Motika, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
www.informatik.uni-kiel.de/rtsys/
{ssm,cmot,rvh}@informatik.uni-kiel.de

Abstract. The Sequentially Constructive Language (SCL) is a minimal synchronous language that captures the essence of the Sequentially Constructive Model of Computation (SCMoC), a recently proposed extension of the classical synchronous model of computation. The SCMoC uses sequential scheduling information to increase the class of constructive (legal) synchronous programs. This facilitates the adoption of synchronous programming by users familiar with sequential programming in C or Java, thus simplifying the design of concurrent reactive/embedded systems with deterministic behavior. The theoretical foundations of the SCMoC are fairly well covered by now, and also the upstream compilation from SCCharts (a Statechart dialect) and SCEst (a variant of Esterel) to SCL. In this paper, we focus on how to compile SCL down to data-flow equations, which ultimately can be synthesized to hardware or executed in software.

1 Motivation & Related Work

Reactive systems are characterized by their regular interaction with the environment, typically under real-time constraints. Physical time is conceptually divided into a sequence of discrete *ticks*, and during each tick, the reactive system reads inputs from the environment, processes them according to some internal system state, and then updates the system state and produces outputs to the environment. Reactive systems may implement safety-critical applications where determinate behavior is essential. However, they often entail concurrent threads of control and interact through shared memory, which makes determinacy challenging. This has motivated the development of *synchronous languages* [3], which have been used successfully in the industry since the 1990s, e. g., for the development of avionics software or power plant control. Edwards [4] and Potop-Butucaru et al. [11] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages.

Synchronous languages achieve determinacy for concurrent systems by the *synchrony hypothesis*, which abstracts from computation time and thus assumes outputs to occur synchronously with the inputs they react to. This is achieved by demanding unique, stable values for all shared variables throughout each tick.

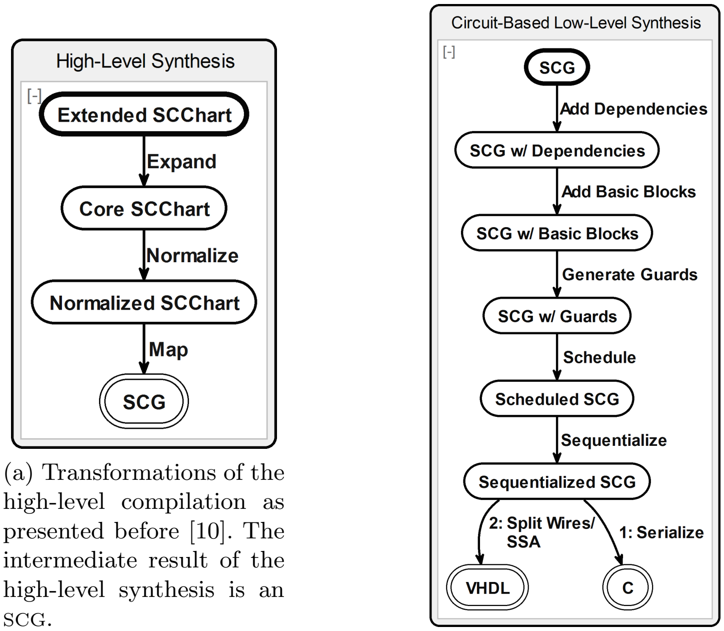
This simplifies formal reasoning, and has a natural, physical analogy of well-defined, stable voltages on all wires on a hardware circuit. We thus say that a synchronous program is *causal*, or *constructive*, if and only if it corresponds to a circuit where all wires have well-defined voltages for all possible inputs and all possible internal states, independent of variations in signal propagation delays in an actual hardware realization.

This Synchronous Model of Computation (SMoC) for concurrent programming contrasts with, e. g., the programming model offered by Java or Posix threads. There the outcome of a program with concurrent threads that share variables may depend on run-time scheduling decisions out of control of the programmer. In Java, achieving determinacy often requires additional, brittle constructs such as semaphores, monitors, barrier synchronizations etc. [9].

However, the classic realization of the synchrony hypothesis comes with restrictions that may be difficult to realize, in particular for novice programmers used to imperative languages such as C or Java. Specifically, the requirement that shared variables cannot change within a tick may come as a surprise. For example, a construct such as "if (x) { x = false }" would be forbidden in the classical SMoC, because x could be true and false within a tick. Instead, one could write for example "if (pre(x)) { x = false }" which states that if x was true in the *previous* tick, then set it to false in the current tick. This, however, would introduce an artificial tick boundary in a computation that conceptually has nothing to do with the passage of physical time. Conversely, the computation in question is purely sequential, with an obvious order of computation: first the test whether x is true, second the possible assignment to false. Thus there is no race condition between the read and the write of x, and a compiler should have no difficulty to produce determinate code.

The desire to combine the foundations and nice properties of synchronous languages with instantaneous memory updates has motivated the development of the *Sequentially Constructive* model of computation (SCMoC) [17,1]. The basic idea is to use any sequential scheduling information in the program to schedule computations in a determinate fashion, and to use a particular scheduling protocol to order concurrent variable accesses. The SCMoC may still reject certain programs as not being sequentially constructive, such as "fork x=y par y=x join", where the SCMoC does not know how to order the concurrent accesses to x and y and which is therefore not determinate. However, the SCMoC accepts many more programs than the SMoC.

The SCMoC is formally defined with the *Sequentially Constructive Graph* (SCG), which is textually represented as the *Sequentially Constructive Language* (SCL). The SCL is rather low-level and very simple yet rich enough to be used as an intermediate language for compiling higher-level languages that want to build on the SCMoC. So far, two such higher-level languages have been proposed. The first language is *Sequentially Constructive Statecharts* (*SCCharts*) [15], a dialect of Harel's statecharts [7]. The second language is *Sequentially Constructive Esterel* (SCEst) [12], an extension of Esterel [11].



(a) Transformations of the high-level compilation as presented before [10]. The intermediate result of the high-level synthesis is an SCG.

(b) Transformations of the low-level data-flow synthesis.

Fig. 1. Single-Pass Language-Driven Incremental Compilation (SLIC) approach transforming SCCharts to code.

Outline and Contributions

The original paper on SCCharts [15] briefly sketched two approaches to compile SCL further into software or hardware, namely the *data-flow approach* and the *priority-based approach*. We here present the data-flow approach in detail.

In Sec. 2 we explain the SCL and recapitulate definitions that are important for the remainder of the paper. The previously presented compilation chain (cf. Fig. 1) shows that the data-flow approach transforms from the generated SCG to a sequentialized variant and then eventually into code. This transformation is executed in several steps depicted in Fig. 1b. Each step is described in Sec. 3, which is the technical core of this paper. We here follow the Single-Pass Language-Driven Incremental Compilation (SLIC) approach [10] where every step is executed as a Model-to-Model (M2M) transformation, which facilitates a modular compilation chain.

We have validated our compilation chain with a range of test cases. This includes fairly extensive use in the class room. In Sec. 4 we report on a medium-sized railway project that uses the data-flow approach to synthesize a railway controller. We conclude in Sec. 5.

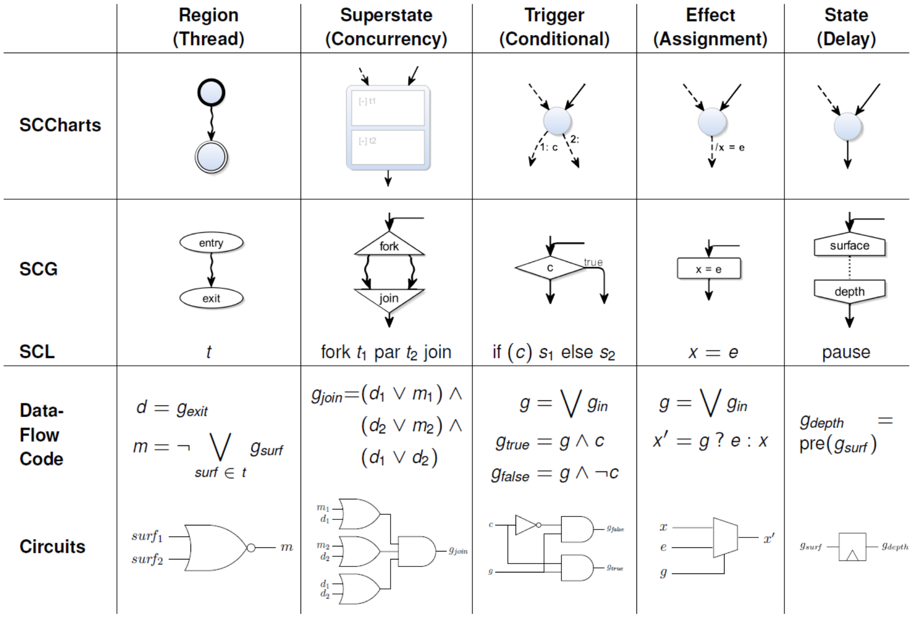


Fig. 2. Matrix showing the entire mapping throughout the transformation process from SCCharts to circuits.

2 The Sequentially Constructive Language (SCL)

This section gives an overview of SCL. It only gives the necessary explanation to understand the data-flow transformations. We refer to the sequentially constructive foundations [17] for a more formal and wider introduction.

The minimal SCL is adopted from C and Esterel. The concurrent and sequential control-flow of an SCL program is given by an SCG, which acts as an internal representation for elaboration, analysis and code generation. Rows two and three of Fig. 2 present an overview of SCL and SCG elements and the mapping between them. SCL is a concurrent imperative language with shared variable communication. Variables can be both *written* and *read* by concurrent threads. Reads and writes are collectively referred to as variable *accesses*. SCL programs consist of one or more sequentially ordered statements with the following *abstract syntax* of *statements*

$$s ::= x = e \mid s \,;\, s \mid \text{if } (e)\ s \text{ else } s \mid l\!: s \mid \text{goto } l \mid \text{fork } s \text{ par } s \text{ join} \mid \text{pause}$$

where x is a *variable*, e is an *expression* and $l \in L$ is a *program label*. The statements s comprise the standard operations assignment, the sequence operator, conditional statements, labelled commands and jumps.

The sublanguage of *expressions* e used in assignments and conditionals is not restricted. However, we rule out side effects when evaluating e. Our notion of sequential constructiveness is based on the idea that the compiler guarantees a strict "initialize-update-read" (*iur*) execution schedule during each macro tick. The *initialize* phase is given by the execution of a class of writes which we call *absolute* writes (e. g., "x = 1"), while the *update* phase consists of executing *relative* writes where scheduling order does not matter (e. g., "x += 2" and "x +=3" can be scheduled in any order, with the same result). All the *read* accesses, in particular the conditional statements which influence the control-flow, are done last.

2.1 SCG Representation

An SCG is a labelled graph $G = (N, E)$ whose *statement nodes* N correspond to the statements of the program, and whose edges E reflect the sequential execution ordering and data dependencies between the statements. Nodes and edges are further described by various attributes. A node n is labelled by the *statement type*. Nodes labelled with $x = e$ are referred to as *assignment nodes*, those with if (e) as *condition nodes*, all other nodes are referred by their statement type (*entry nodes*, *exit nodes*, etc.). Fig. 2 sketches how SCG elements correspond to an SCL program. A technical report [16] describes this mapping in detail.

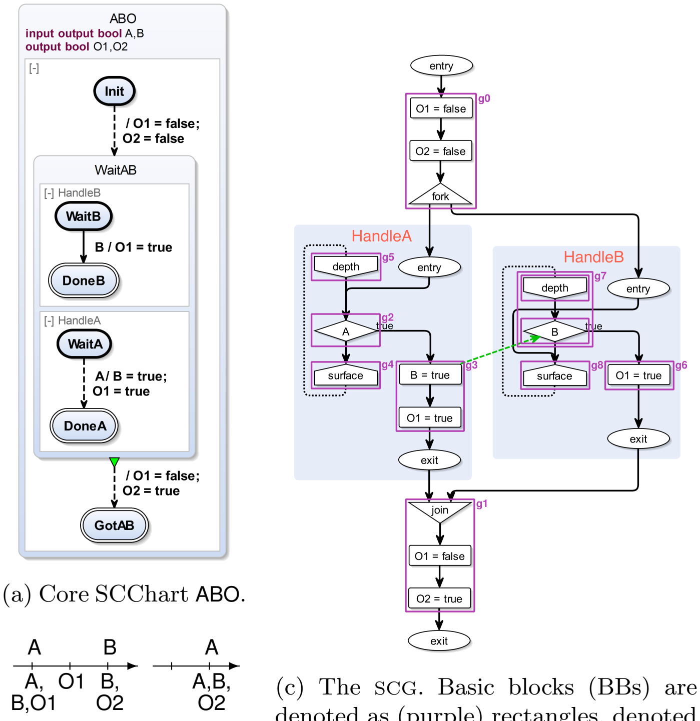
Every edge e has a type $e.type$ that specifies the nature of the particular ordering constraint expressed by e. Edges that follow the initialize-update-read schedule are labeled *iur*-edges. *iur*-edges combined with the sequential control-flow edges are termed *instantaneous* edges. An *SC-schedule* is a subset of instantaneous edges of an SCG. A *structural SC-schedule* is an *SC-schedule* that is solely derived by analysis of the program structure. A program for which the structural SC-schedule is acyclic is *structurally acyclic SC*, abbreviated *SASC*. The data-flow approach presented here requires that the SCG is SASC; this, for example, forbids any loops that are *instantaneous*, i. e., where the loop body is not interrupted by a tick boundary.

2.2 The ABO Example

The ABO example shown in Fig. 3a illustrates the concepts of core SCCharts, namely synchronous ticks, concurrency, deterministic scheduling of concurrent shared variable accesses, and sequential overwriting of variables.

The execution of an SCChart is divided into a sequence of logical ticks. The *interface declaration* of ABO states that A and B are boolean *inputs*, which are initialized by the environment at the beginning of each tick, as well as *outputs*, which are fed back to the environment at the end of each tick. O1 and O2 are boolean outputs, which here are initialized to false, and which are persistent from one tick to the next.

Initially, the system is in state WaitAB, which consists of regions (threads) HandleA and HandleB. HandleA stays in the initial state WaitA until the boolean input A becomes true. Then it sets B and O1 to true and transitions to state DoneA, which is final and hence terminates HandleA. Similarly, WaitB waits for



(a) Core SCChart ABO.

(b) Possible execution traces, with inputs above the tick time line and outputs below.

(c) The SCG. Basic blocks (BBs) are denoted as (purple) rectangles, denoted with their guards; gi guards BB i. The data dependence on B (dashed arrow) splits BB 7 into two scheduling blocks.

```
module ABO
input output bool A, B;
output bool O1, O2;
{
  O1 = false;
  O2 = false;

  fork
    HandleA:
    if (!A) {
      pause;
      goto HandleA;
    }
    B = true;
    O1 = true;

  par
    HandleB:
    pause;
    if (!B) {
      goto HandleB;
    }
    O1 = true;

  join;

  O1 = false;
  O2 = true;
}
```

(d) SCL code.

Fig. 3. The ABO example, illustrating the Core SCChart features [15].

B to become true, sets O1 to true, and transitions to final state DoneB. Once both HandleA and HandleB have terminated, WaitAB is left, O1 is set to false, O2 to true, and state GotAB is entered. The dashed edge denotes the transition to DoneA to be *immediate*, meaning that HandleA does not pause for a tick before it is ready to detect the transition trigger. In contrast, the transition to DoneB in HandleB is *delayed* and thus does not get triggered in any tick in which WaitB is entered.

Two possible execution traces are shown in Fig. 3b. The first trace begins with A set to true by the environment in the initial tick. This triggers the transition to DoneA and sets both B and O1 to true. As this is the initial tick, the non-immediate transition from WaitB to DoneB does not get triggered by the B. In the next tick, all inputs are false, no transitions are triggered, and O1 stays at true. In the third and last tick, B then triggers the transition to DoneB, which sets O1 to true, but sequentially afterwards, O1 is set to false again as part of the transition to GotAB, which is triggered by the termination of HandleA and

HandleB. Hence, at the end of this tick, only O2 will be true because the SCMoC allows O1 to be overwritten sequentially. The second trace illustrates how A in the second tick triggers the transitions to DoneA as well as to DoneB, hence emission of B and O2 and the termination of the automaton.

3 Data-Flow M2M Transformation

As depicted in Fig. 1 the transformation of the SCG mapped from a normalized SCChart to a sequentialized SCG is done in several distinct steps. Following the SLIC approach [10] every step is executed as an M2M transformation. This section explains each of these steps, namely dependency analysis, basic block arrangement, guard creation, scheduling, and sequentialization. For the subsequent analyses we assume that superfluous fork-join-constructs, i. e., containing only one thread, and *dead code* were removed.

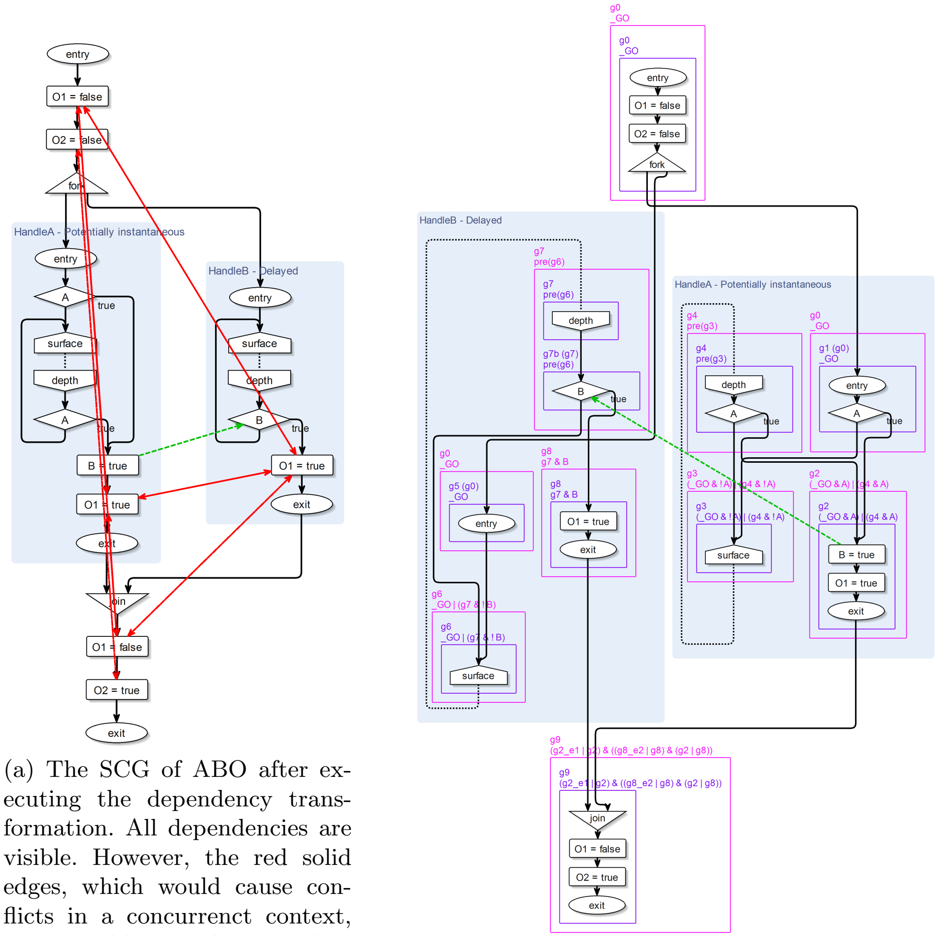
3.1 Dependency Analysis

The analysis of dependencies between different expressions is done straightforwardly. Every assignment and conditional in the program is checked for variable accesses. The type of the access is stored during the compilation. For each pair of accesses it is determined if the access is *concurrent* and/or *confluent*. Confluence means that the scheduling order does not matter, as is the case for example for the aforementioned relative writes.

All non-concurrent (confluent or non-confluent) dependencies are handled according to the sequential control-flow, whereas concurrent accesses are scheduled following the “initialize-update-read” protocol. According to Sec. 2.1, SCL programs that contain immediate dependency cycles are not *constructive* and are rejected.

In the ABO example applying the dependency analysis reveals one concurrent dependency as depicted in Fig. 4a. The concurrent dependency is shown as green, dashed line between the two threads connecting the B = true assignment in HandleA with the B conditional of HandleB. Furthermore, there are several other dependencies indicated by the red solid edges. These dependencies would cause conflicts in a purely concurrent context if they were not confluent. However, in ABO they can be executed sequentially. For example, the assignment to O1 at the top before the fork gets always executed before the assignment to O1 after the join.[1]

[1] Non-conflicting non-concurrent dependencies are normally not shown in our tool chain. Here, we activated the visualization to demonstrate the dependency analysis. However, concurrent non-confluent dependencies, which represent conflicts, are always shown.



(a) The SCG of ABO after executing the dependency transformation. All dependencies are visible. However, the red solid edges, which would cause conflicts in a concurrent context, are unproblematic because they either can be solved sequentially or are confluent.

(b) The SCG of ABO after proceeding with the BB and guard creation transformations. The nodes are encapsulated in their SB and, hence, BB. Each BB has its *guard expression* attached.

Fig. 4. Transformation of the SCG following the SLIC approach executing the dependency, the basic block, and the guard creation transformations.

3.2 Basic Block Arrangement

The data-flow compilation approach converts all control flow, be it sequential or concurrent, into a flat sequence of *guarded commands*. As a consequence, we cannot handle instantaneous loops with this code generation approach, as mentioned before. A guarded command is a statement that gets executed in the current tick if and only if a specific guard evaluates to true in the current tick; guards have a unique value throughout each tick.

To economize on the number of guards, we make use of the standard concept of Basic Blocks (BBs). In our setting BBs denote sets of statements that are executed together in a tick, i. e., either all or none of them are executed in a tick. Thus, all statements within a BB may share the same guard. The following rules, defined in a more formal way elsewhere [13], define BBs:

- A BB begins if the SCG representation of that statement has two or more incoming control-flow edges.
- A BB ends with a statement that forks the SCG control-flow and hence, has two or more outgoing control-flow edges. The last instruction of a thread may also be the closure of a BB.
- BBs are split at pause statements.
- SCG fork nodes close a BB, whereas join nodes start a new one.
- Any statement of a given program can only be included in one BB at any time.

The statements in a BB are not necessarily executed atomically, in the sense that BBs of concurrent threads may be interspersed with each other in order to satisfy dependencies induced by shared variables. Therefore, we may further divide each BB in Scheduling Blocks (SBs). With the BBs defining **what** set of instructions becomes active in the current tick, the SBs take the dependencies into account and define **when** a particular instruction set is executed, thus, defining the execution order. Therefore, a SB subdivides a BB if an incoming dependency edge targets an instruction inside the BB because the scheduler might want to reschedule here.

The arrangement of the blocks for the ABO example is depicted in Fig 4b. Building the blocks according to the rules imposed earlier in this section, each instruction is included in a SB directly surrounding it. The SB itself is included in a BB. In ABO most BBs only comprise one SB. A BB containing two SBs can be seen at the top of HandleB marked with the guard g7. This block gets divided due to the dependency found in Sec. 3.1.

3.3 Guard Creation

In every tick instance a BB may be *active* or *inactive*. The activity state of a BB depends on previous BBs and is called the *guard* of the BB. The BB determines the guard expression, whereas the SB defines the order of execution as described in Sec. 3.2.

Simple Guards directly depend on their predecessors. The guard of the first block in an SCL program depends on the *GO* start signal of the environment usually emitted at the *initialization* or *reset* of the program. Outgoing control-flows of a BB may serve as *guard expressions*, or *activators*, for succeeding BBs. Thus, a guard is a disjunction of all preceding activators and evaluates to true as long as one incoming activator is true. Generally speaking, a BB is active, if and only if at least one of its guard expressions is active in the same tick.

Not all BBs need an own guard if their order of activation is determined statically at compile-time. Therefore, we use standard compiler techniques such as *copy propagation* [2] to reduce the amount of needed unique guards.

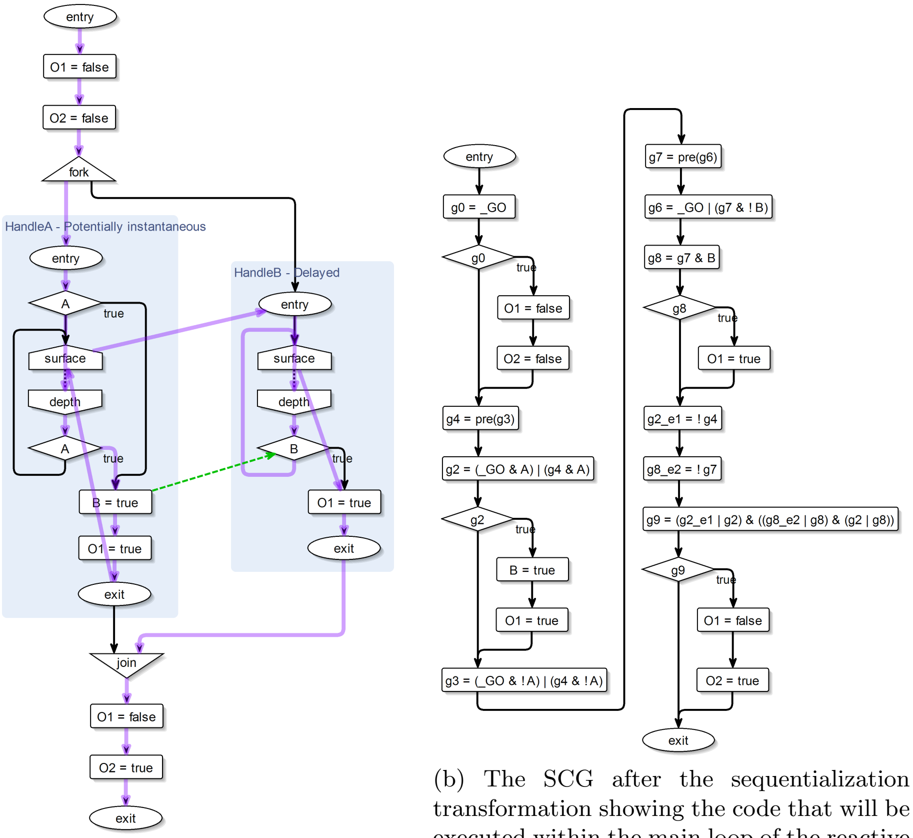
The guards generated for the ABO example are shown in Fig. 4b. Each BB has caption lines at the top. The first line shows the guard of the block and the guard expression generated for this block is displayed in the second line. For instance, the first block of ABO is named g0. Since it is the first block of the program, it gets activated once the program starts (and at every reset). Therefore, the guard expression is equal to the _GO signal of the environment. Another example is g8 in HandleB. Here, the guard depends on its predecessor g7. Furthermore, the BB of g8 is a successor of the *true branch* of the conditional in the block of g7 which evaluates to true if B is true. Therefore, the guard expression of g8 is g7 & B.

Notice that the BB of g7 includes two SBs due to the incoming dependency edge. SBs are named like their parent BB and suffixed alphabetically after the first one. As explained in Sec. 3.2, the scheduler may reschedule between any two SBs. However, as both SBs live inside the same BB, the guard expression of both SB are identical as explained earlier. The guard expression of g7 is pre(g6) meaning that it is set to the activation status of g6 in the tick before. This implements a *tick boundary*, and we also say that a program resumes the execution at the *depth* here if it was paused at the *surface* in the preceding tick.

Also, the effects of the copy propagation can be seen in the figure. SB g1 in HandleA will immediately become active after program activation because it solely depends on g0. Hence, the _GO signal also triggers this block.

Complex Guards cannot simply depend on their direct predecessors. In SCL programs, forked threads must be joined at some point in time. The join instruction will not proceed unless each thread has finished. Hence, a BB including a join node only activates if all preceding threads are terminated and at least one of them exited in the actual tick instance. Each thread status is signaled by an *empty flag* which describes whether or not a thread is inactive. All empty flags are combined in a conjunction together with a combination of exit codes that signal whether at least one thread terminated in this tick instance. The empty flag is combined with the _GO signal of the preceding circuit to detect active instantaneous threads. BBs that are responsible for joining threads are also called *synchronizer*. The construction of the synchronizer is done similar to the synchronizer circuit described in Compiling Esterel [11].

In the ABO example the BB with guard g9 is activated by a complex guard expression. The conjunction consists of three parts. The first two parts (g2_e1 | g2) & (g8_e2 | g8) indicate if the threads HandleA and HandleB are inactive or terminated in this tick. g2 and g8 are the SBs that are active if their thread is exited in this tick because they include the corresponding exit node. The empty flags, suffixed with _*ex*, are set to true if the registers, i. e., pause instructions, are inactive. To activate the block with the join node, at least one thread must have been exited in this tick. Therefore the third part (g2 | g8) is checked.



(a) The SCG after the scheduling transformation depicting the route the scheduler has chosen. Here, only one context switch is necessary.

(b) The SCG after the sequentialization transformation showing the code that will be executed within the main loop of the reactive tick cycle.

Fig. 5. Transformation of the SCG following the SLIC approach executing the scheduling and sequentialize transformations.

3.4 Scheduling

In the scheduling step (cf. Fig. 1) the transformation returns a valid schedule for the given program if one exists. Therefore, the blocks, and hence their included instructions, are ordered topologically according to their guard expressions and with respect to any dependencies between the blocks. This may result in arbitrarily many context switches between threads. However, to support any low-level analysis on threads that may be executed later on, it is desirable to perform as few switches between threads as possible. One example for a low-level analysis is a worst-case execution time (WCET) analysis. Hints about superfluous context switches can be found in our work towards interactive timing analysis [5]. Here, any context switch results in the insertion of an so called *timing program points*. As these points may influence measurements, one should not include more than necessary.

The schedule chosen for ABO can be seen in Fig. 5a. The bold purple arrow depicts the path chosen as schedule. As one would expect the program starts at the first entry node. Then, at the fork, the scheduler chooses to proceed with HandleA. It switches to HandleB not until HandleA has finished and proceeds to the join after both threads completed.

This is not the only possible schedule. Nevertheless, every valid schedule will produce the same deterministic output [17]. It would have been valid to start with HandleB and then switch to HandleA. However, the scheduler could not have finished HandleB completely as it has done with HandleA because HandleB depends on HandleA due to the conditional node testing B. Therefore, beginning the other way around would lead to more context switches.

3.5 Sequentializing

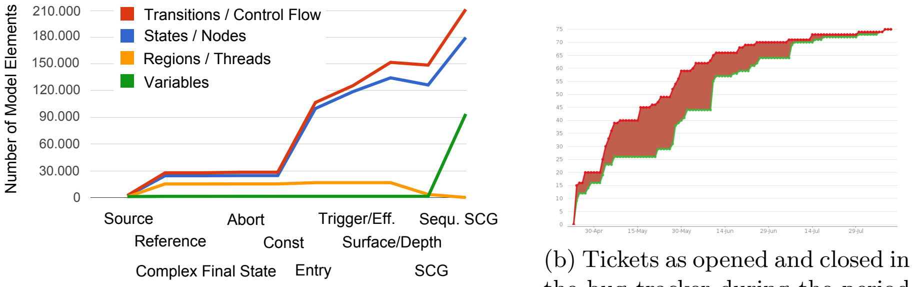
Finally, from the schedule we can derive the sequentialized program. Therefore, the guards created before are written in the order defined by the scheduler. If an SB contains assignments they must be executed if the BB is active. Hence, these are added to the sequentialized program guarded by the guard of their block.

The fully sequentialized program for the ABO program is shown in Fig. 5b. One will recognize the guard expressions from Fig. 4b. Here, they are ordered according to the schedule depicted in Fig. 5a. Whenever a guard is responsible for any assignments, a conditional is added which holds the guard as condition. For instance, at the beginning of the program g0 is set to the _GO signal of the environment. Hence, it will be true in the first tick and therefore O1 and O2 will be initialized with false. If A is also true in the first tick, B and O1 are set to true. However, as described in Sec. 2.2 the join g9 is not activated because HandleB cannot reach its exit node in g8 in the first tick. g8's expression is g7 & B and g7 depends on pre(g6). Thus, g6 must have been active in the tick before so that HandleB may terminate in the actual tick. This is not possible in the first tick.

Fig. 5b also depicts the empty flags g2_e1 and g8_e2 needed for the synchronizer. As discussed in Sec. 3.3 these indicate whether a thread is inactive in the actual tick. Therefore, they negate the status of the concurrent depth nodes which abstractly resemble registers. The complex guard expression is then constructed as explained earlier in Sec. 3.3. The sequentialized program is now ready to be deployed. It can be translated to various languages such as C, Java or VHDL. Deployment to hardware requires the final deployment step to apply an SSA transformation to the variables used in the program if they are written more than once [8].

4 Experimental Results

In summer term 2014 we launched a student project [14] where the task was to build an SCCharts based controller for a rather large model railway system.



(a) Hiding complexity by using Extended SCCharts features: Expansion of SCCharts features down to sequentialized SCG elements gives an idea of the complexity of this model [14].

(b) Tickets as opened and closed in the bug tracker during the period of this project validating maintainability and extendability of the model based SLIC compiler approach [14].

Fig. 6. The SCCharts SLIC based compilation approach turns out to be practically usable even for complex models and to be maintainable and extendable.

The model railway system consists of a total track length of 127 meters split into 48 individually controllable block segments with 28 switch points. The final controller is able to drive 11 trains simultaneously with integrated dead-lock and live-lock avoidance. The controller fully expands to 135,000 states, 152,000 transitions and 17,000 concurrent regions after eliminating all reference states by a reference state compiler transformation. 1,628 states were modeled manually together with 2,219 transition and 183 concurrent regions. Compared to David Harel's Wristwatch [7], which was considered a complex statechart back in 1986, we would also call the SCCharts model railway controller at least a medium-size real-world complex system. It compiles using the presented tool chain in 2-3 minutes and generates about 650,000 lines of C code. Still the response time of the running controller was measured to be smaller than 2ms on a standard PC.

We measured the number of model elements for the SCCharts model railway controller example at every intermediate stage of the SLIC compile chain (cf. Fig. 1). Fig. 6a shows the result and suggests how much complexity of the resulting sequentialized SCG model could be hidden by using Extended SCCharts features for modeling the complex behavior of this controller. The students were not only using our SCCharts compiler tool chain but also struggling with teething troubles of our early prototype compiler. That resulted in many bug reports especially in the middle of the project when the students started modeling. As Fig. 6b attests we were able to quickly resolve most of the problems without introducing more new problems. This circumstance validates maintainability of the model based SLIC approach used for the compiler. Additionally new feature requests like reference state expansion arose during the project and could be integrated into the existing compiler validating extendability of our overall approach.

5 Conclusions

As discussed before [10] being able to quickly prototype a modular compiler that is easy to validate and to customize prompted us to follow the SLIC approach. The SLIC approach showed that it is possible to use model-driven aspects to build a compiler that is also fairly compact and efficient. Following this route we here presented SLIC transformation rules for the data-flow approach, one of the two proposed low-level methods for generating code for SCCharts [15]. Similar to the high-level SCCharts transformation rules, the SCL transformations obey the proposed pattern:

- The compilation steps are M2M transformations where the resulting model contains all information. There are no other, hidden data structures.
- The intermediate transformation steps are in the same language. We just apply a sequence of language operations, that added one analyses result at a time.

We see numerous directions for future work. For example, we want to use existing simulation tools to evaluate the activity state of each guard during runtime. Hence, feeding the information back to the simulator and using the M2M transformation information of the SLIC approach, it should be possible to display each active area of the SCCharts on modelling level. Synchronizing threads as explained in Sec. 3.3 becomes difficult when dealing with instantaneous feedback loops. There are possibilities to handle such a feedback in the data-flow approach as long as at least one thread is delayed to prevent instantaneous cycles. We would like to further investigate possibilities to handle feedbacks and implement these in our tool chain. Another active area is that of interactive timing analysis [6], where we investigate how to best preserve timing-information across M2M transformations. The main advantage of our approach is its interactivity. Nonetheless we envision a fully automatic compilation process including the possibility to include our compiler in scripts (e. g., a Makefile) or using it online in the Web. Another important question is how much parallelism should we derive from the initial concurrency modeled by the modeler. Allowing programs to be executed partly in parallel without full sequentialization is ongoing research.

References

1. J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann. Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. *Acta Informatica, Special Issue on Combining Compositionality and Concurrency*, 52(4):393–442, 2015.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.
3. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, Jan. 2003. IEEE.

4. S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.
5. I. Fuhrmann, D. Broman, S. Smyth, and R. von Hanxleden. Towards interactive timing analysis for designing reactive systems. Reconciling Performance and Predictability (RePP'14), satellite event of ETAPS'14, Apr. 2014.
6. I. Fuhrmann, D. Broman, S. Smyth, and R. von Hanxleden. Towards interactive timing analysis for designing reactive systems. Technical Report UCB/EECS-2014-26, EECS Department, University of California, Berkeley, Apr. 2014.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
8. G. Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, Oct. 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf.
9. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
10. C. Motika, S. Smyth, and R. von Hanxleden. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, Oct. 2014.
11. D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.
12. K. Rathlev, S. Smyth, C. Motika, R. von Hanxleden, and M. Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15)*, Austin, TX, USA, Sept. 2015.
13. S. Smyth. Code generation for sequential constructiveness. Diploma thesis, Kiel University, Department of Computer Science, July 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf.
14. S. Smyth, C. Motika, A. Schulz-Rosengarten, N. B. Wechselberg, C. Sprung, and R. von Hanxleden. SCCharts: the railway project report. Technical Report 1510, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015. ISSN 2192-6247.
15. R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. 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.
16. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013. ISSN 2192-6247.
17. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, 13(4s):144:1–144:26, July 2014.