SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts

Abstract

We present a new visual language, SCCharts, designed for specifying safety-critical reactive systems. SCCharts use a statechart notation and provides deterministic concurrency based on a synchronous model of computation (MoC), without restrictions common to previous synchronous MoCs. Specifically, we lift earlier limitations on sequential accesses to shared variables, by leveraging the sequentially constructive MoC. The semantics and key features of SCCharts are defined by a very small set of elements, the *Core SCCharts*, consisting of state machines plus fork/join concurrency. We also present a compilation chain that allows efficient synthesis of software and hardware.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.4 [*Processors*]: Compilers

Keywords Statecharts, synchronous languages, visual languages, determinism, concurrency, hw/sw-synthesis, safety-critical systems

1. Introduction

Background. Statecharts, introduced by Harel in the late 1980s [11], have become a popular means for specifying the behavior of embedded, reactive systems. The visual syntax of statecharts is intuitively understandable for application experts from different domains who are not necessarily computer scientists. The statechart concepts of hierarchy and concurrency allow the expression of complex behavior in a much more compact fashion than standard, flat finite state machines. However, defining a suitable semantics for the statechart syntax is by no means trivial, as evinced by the multitude of different statechart interpretations. In the 1990s already, von der Beeck [18] identified a list of 19 different non-trivial semantical issues, and compared 24 different semantics proposals; these did not even include the "official" semantics of the original Harel statecharts (clarified later by Harel [12]) nor the many statechart variants developed since then, such as UML statecharts with its run-to-completion semantics. One of the semantical issues identified early on for statecharts is the question of *determinism*, which is not surprising as statecharts include concurrency and hence are potentially subject to race conditions. In many application areas, including the area of safety-critical applications that has motivated the work presented here, determinism is a strict requirement. Given a sequence of input stimuli, a safety-critical reactive system must always produce the same sequence of outputs, even if the internal behavior involves concurrency. Many statechart variants do not fulfill this determinism requirement; e. g., STATEMATE, the original statecharts tool, detected potential non-determinism at run-time, but not at compile time.

One approach for achieving determinism, successfully employed by the family of synchronous languages, is to abstract execution time away. This implies unique variable (or "signal") values throughout an instantaneous reaction chain, or *tick*, which eliminates race conditions. This concept has also been applied to statechart-like visual languages, such as André's SyncCharts [1]. The synchronous model of computation (SMoC) is a sound approach that solves the determinism issue. However, it is quite restrictive due to the "only one value per reaction" requirement. For example, the classical SMoC cannot directly express something like if $(x < 0)$ x = 0 for some shared variable x. This restriction may seem natural to hardware designers, who are used to the requirement of stable, unique voltage values within a clock cycle and the lack of built-in sequencing in combinational, parallel circuits. However, this limitation often causes bewilderment with programmers used to languages like C or Java, where such sequential variable accesses pose no problem and do not result in compile-time errors. This issue has motivated the *sequentially constructive* (SC) MoC proposed recently [20], which harnesses the synchronous execution model to achieve deterministic concurrency while addressing concerns that synchronous languages are unnecessarily restrictive and difficult to adopt. In essence, the SC MoC extends the classical synchronous MoC by allowing variables to be read and written in any order as long as sequentiality expressed in the program provides sufficient scheduling information to rule out race conditions.

Contributions. Concerning programming language design, we here present a new, visual modeling language for reactive systems, called *Sequentially Constructive Statecharts*, or *SCCharts*. SCCharts have been designed with safety-critical applications in mind and aim for easy adaptation. The safety-critical focus is reflected not only in the deterministic semantics, but also in the approach to defining the language; the basis of the language is a minimal set of constructs, termed *Core SCCharts*, which facilitate rigorous formal analysis and verification. Building on these core constructs, *Extended SCCharts* add expressiveness with a number of additional constructs. Concerning implementation, we present a complete compilation chain from Extended SCCharts down to software (C) or hardware (VHDL). We discuss a novel approach towards handling aborts and other complex reactive control flow patterns by model-to-model pre-processing transformations into the minimal set of language features provided by Core SCCharts. Each transformation is of limited complexity and open to inspection by

[Copyright notice will appear here once 'preprint' option is removed.]

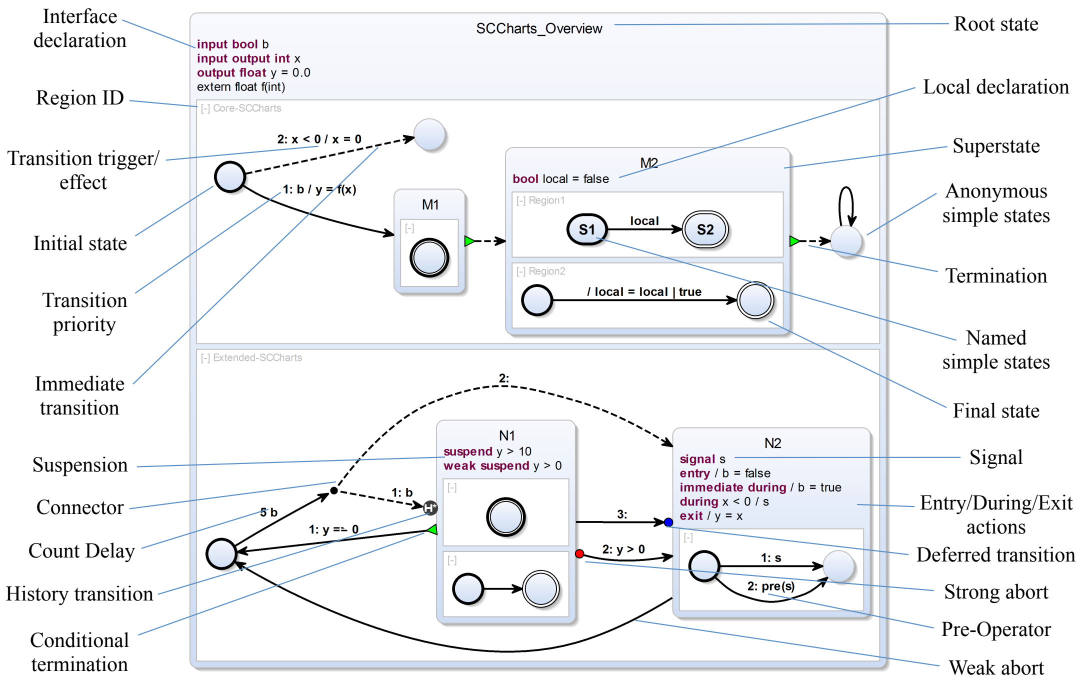


Figure 1. Syntax overview of SCCharts. The upper region contains Core SCCharts elements only (Sec. 3), the lower region illustrates Extended SCCharts (Sec. 4).

the modeler, unlike existing "monolithic" statecharts compilation approaches. Our *circuit-based* synthesis proposal presents a new, simpler alternative of the established Esterel circuit semantics [15]. Our alternative exploits the minimal nature of Core SCCharts, but, unlike the original circuit semantics, also encompasses sequentiality. Finally, as part of our *priority-based* synthesis proposal for SCCharts, we introduce a new, leaner variant of Synchronous C [19], termed SCL$_P$.

Outline. After summarizing related work in the next section, Sec. 3 presents Core SCCharts. The next three sections describe compilation phases, beginning with a high-level transformation (Sec. 4) followed by the alternative low-level synthesis steps, circuit-based in Sec. 5 and priority-based in Sec. 6. We present experimental results in Sec. 7 and conclude in Sec. 8.

2. Related Work

The proper handling of concurrency has a long tradition in computer science, yet, as argued succinctly by Hansen [10] or Lee [13], still has not found its way into mainstream programming languages such as Java. Synchronous languages were largely motivated by the desire to bring determinism to reactive control flow, which covers concurrency and preemption [3]. Syntax and semantics of SCCharts have taken much inspiration from André's SyncCharts [1], introduced as Safe State Machines (SSMs) in Esterel Studio, and its predecessor Argos [14]. SyncCharts combines a statechart syntax with a semantics very close to the synchronous, textual language Esterel [4]. Colaço et al. [6, 7] have presented a SyncCharts/SSM variant, now implemented in the *Safety Critical Application Development Environment* (SCADE), whose semantics is an extension of the synchronous data-flow semantics of Lustre [9]. They use an elegant construct that basically refines boolean clocks into "state clocks." The functional synchronous Lucid Synchrone [7] allows the definition of local names, which can be used to encode sequential orderings, as in `let x = ... in x = x + 1`; the same effect can be achieved by converting a program into static single assignment (SSA) form [2]. In Lucid Synchrone, this is motivated also by the desire to sequentialize external function calls with side effects, such as "`print`." Caspi et al. [5] have extended Lustre with a shared memory model. However, they adhere to the current synchronous model of execution in that they forbid multiple writes even when they are sequentially ordered. Unlike these SyncCharts/Lustre variants, SCCharts presented here are not restricted to constructiveness in Berry's sense [4], but relax this requirement to sequential constructiveness (SC). Thus SCCharts are a conservative extension of SyncCharts, in the sense that Berry-constructive SyncCharts are also valid SCCharts, but there is a large class of valid SCCharts that are perfectly deterministic under SC scheduling but would be rejected by a SyncCharts compiler. In a nutshell, like Harel has [11] stated "statecharts = state-diagrams + depth + orthogonality + broadcast communication," one may say "SyncCharts = statecharts syntax + Esterel semantics" and "SCCharts = SyncCharts + sequential constructiveness + extensions," where the extensions are mostly drawn from SCADE (e. g., deferred transitions) and Quartz [16]/Esterel v7 (e. g., weak suspend).

Edwards [8] and Potop-Butucaru et al. [15] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages. We present an alternative compilation approach that handles most constructs that are challenging for a synchronous languages compiler by a sequence of model-to-model transformations, until only a small set of Core SCChart constructs remains.

3. Core SCCharts

An overview of the elements of SCCharts is shown in Fig. 1. The upper part illustrates Core SCCharts, which contain the key ingredients of statecharts, namely concurrency and hierarchy. The lower region contains elements from Extended SCCharts.

3.1 Interface declarations

An SCChart starts at the top with an *interface declaration* that can declare variables and external functions. Variables can be *inputs*, which are read from the environment, or *outputs*, which are written to the environment. Variables can also be *inputoutput* variables, which are both inputs and outputs; these are read from the environment, optionally modified, and written back to the environment. In the following, when we refer to inputs or to outputs, this generally includes inputoutputs as well. The environment initializes inputs at the beginning of the tick (*stimulus*), e. g., according to some sensor data. Outputs are used at the end of a tick (*response*), e. g., to feed some actuators. Output variables that are not also input variables are not initialized by the environment at each tick. During a tick, variables may be incrementally updated by the SCChart through internal computations not observable by the environment.

The interface declaration also allows the declaration of local variables, which are neither input nor output. An interface declaration may be attached to states other than the top-level state. This also allows the modularization of SCCharts, at lower levels, using a static macro referencing/expansion mechanism not detailed further here. In this case, the interface declaration serves for compile-time variable binding/renaming. Then the interaction of an SCChart with its environment via input/output variables must not be limited to the beginning and the end of a tick, but can happen arbitrarily, as governed by the SC scheduling rules described later.

Non-input variables are persistent across tick boundaries, even if their scope is left and re-entered, since they are statically allocated. However, they are per default uninitialized, like in C. This means that when a variable v is read and has not been written before, the read value is undefined. It is therefore sensible to statically (and necessarily conservatively) check for such possible uninitialized reads. One way to avoid uninitialized reads is to augment variable declarations with explicit *initializations*, as provided by Extended SCCharts.

3.2 States and transitions

The basic ingredients of SCCharts are *states* and *transitions* that go from a *source state* to a *target state*. When an SCChart is in a certain state, we also say that this state is *active*.

Transitions may carry a *transition label* consisting of a *trigger* and an *effect*, both of which are optional. A "/" separates trigger and effect, but may also indicate division. We suggest to disambiguate the two interpretations, where necessary, by putting divisions into parentheses. I. e., the leftmost, not parenthesized "/" is interpreted as a trigger/action separator, others are interpreted as division operators.

When a transition trigger becomes true and the source state is active, the transition is taken instantaneously, meaning that the source state is left and the target state is entered in the same tick. However, transition triggers are per default *delayed*, meaning that they are disabled in the tick in which the source state just got entered. This convention helps to avoid instantaneous loops, which can potentially result in causality problems. One can override this by making a transition *immediate* (shown as dashed transition). Multiple transitions originating from the same source state are disambiguated with a unique *priority*; first the transition with priority 1 gets tested, if that is not taken, priority 2 gets tested, and so on.

If a state has an immediate outgoing transition that has no trigger condition, we refer to this transition as *default transition*, because it will always be taken, and we say that the state is *transient*, because it will always be left in the same tick as it is entered.

3.3 Hierarchy and concurrency

A state can be either a *simple state* or it can be refined into a *superstate*, which encloses one or several concurrent *regions* (separated with dashed lines). Conceptually, a region corresponds to a thread. Each region must have exactly one *initial state* (thick border). When a region enters a *final state* (double border), then the region *terminates*.

A superstate may have an outgoing *termination* transition (green triangle), also called *unconditional termination* or, in SyncCharts, *normal termination*, which gets taken when all regions have reached a final state. Termination transitions are always immediate. They may be labeled with an action, but—in Core SCCharts—do not have an explicit trigger condition. Hence a superstate should have at most one outgoing termination, as in case of multiple terminations only the one with highest priority can ever be taken. In Core SCCharts, superstates cannot be marked final; this is allowed in Extended SCCharts.

3.4 Termination

Region termination, final states and termination transitions may seem like straightforward concepts. However, their precise semantics and the choices we made in SCCharts deserve some further discussion, as different interpretations have emerged in the past.

Region termination here means that a region "does not do anything anymore." This implies that final states have no outgoing transitions, no refinements, no interface declaration, and no During/Exit actions (introduced in Extended SCCharts) associated with them. Thus final states here have a fairly strong interpretation, i. e., they are quite restricted. This allows a very straightforward implementation, as one can then re-use the information on which regions are active, which is needed for scheduling purposes anyway. An alternative semantics for final states would be to just say that the surrounding superstate terminates when all its regions have reached a final state. This interpration of final states would be weaker in the sense that it would still allow a region to leave a final state again, and a final state might still perform actions or execute refinements. For Core SCCharts, this choice was rejected, due to the aforementioned efficiency reasons. However, the weaker, more permissive interpretation is included in Extended SCCharts.

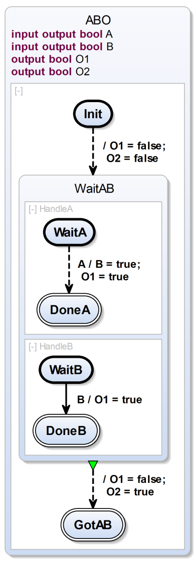
Conversely, a region effectively terminates whenever a region has no During action and reaches a state with no outgoing transitions, no refinements, and no associated actions. However, for clarity, we here require that the state must be explicitly marked as final if we want to enable termination of the surrounding superstate. Thus "reaching a final state" is a stronger condition than "region termination," and we link termination of the superstate to all of its regions reaching final states, not to region termination. This implies that a termination transition of a superstate can never be taken if any region enclosed by that superstate does not contain any final state. A reasonable style guide might therefore require that when a superstate has a termination transition associated with it, then every region in that superstate must contain a final state. However, unlike suggested for SyncCharts [1], we argue that one should permit final states even if there is no enclosing termination, to clearly indicate termination of a region.

Note that even when all regions of a superstate have reached a final state, and hence have terminated, the enclosing superstate is still considered active until it is left. Thus During actions for the superstate keep getting executed until the state is left.

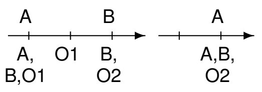
3.5 The ABO Example

The ABO example shown in Fig. 2a illustrates the concepts of Core SCCharts: ticks, concurrency (with forking and joining), deterministic scheduling of shared variable accesses, and sequential overwriting of variables.

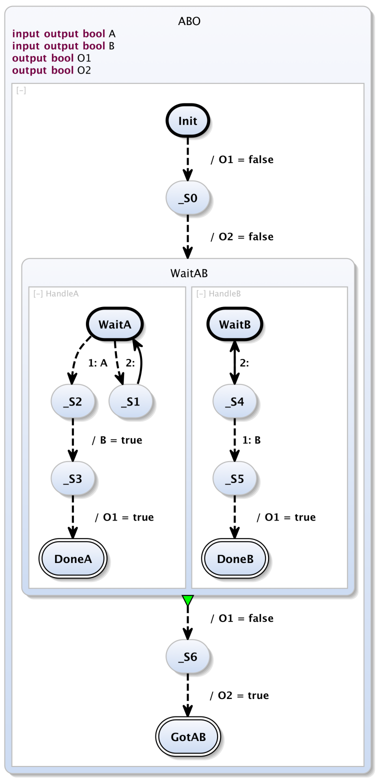
The interface declaration of ABO states that A and B are Boolean inputs as well as outputs. O1 and O2 are Boolean outputs. Two possible execution traces are shown in Fig. 2b. The first trace begins



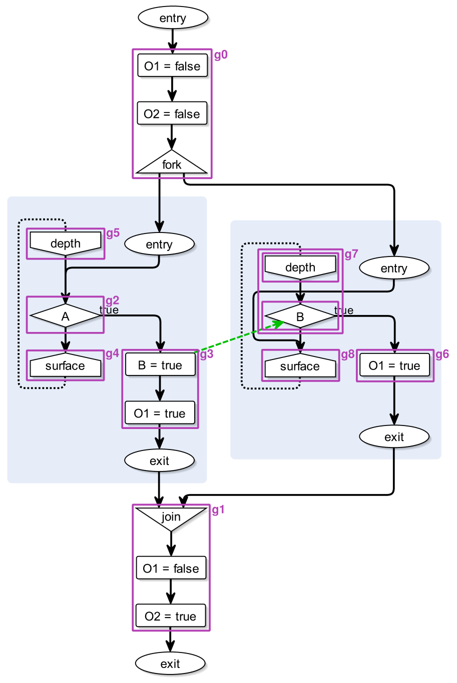
(a) Core SCChart ABO.



(b) Two possible execution traces, with true-valued inputs above the tick time line and true-valued outputs below.



(c) Normalized SCChart ABO-norm.



(d) The SC Graph. 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;
}
```

(e) SCL code.

Figure 2. The ABO example, illustrating the Core SCChart features.

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 delayed 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 B and O2 will be true. 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.

3.6 Sequential Constructiveness

The basic goal of the sequentially constructive MoC is to rule out any race conditions that might induce non-determinism. Roughly, the idea is to forbid conflicting concurrent writes to the same variable, and to schedule a write to some variable before any concurrent read to the same variable. We say that a program "is SC" (is sequentially constructive) if it can be scheduled according to the SC rules. A compiler must reject SCCharts that it cannot prove to be SC. The full SC MoC has further characteristics that we invite the interested reader to study elsewhere [20]. However, the general concurrent-writes-before-reads-rule should suffice to understand the concepts of SCCharts, and in fact one could also customize the SCCharts semantics to scheduling regimes that differ from SC.

In ABO, only B has concurrent read/write accesses. The SC scheduling rules require the write in HandleA to precede the concurrent read in HandleB. This can be achieved by scheduling HandleA before HandleB, and once this is assured, all executions of ABO will produce the same result. Hence, ABO is SC.

A distinguishing feature of the SC MOC is that it allows sequential variable accesses to shared variables within a tick. In ABO, O1 can be first assigned to true and then to false within the same tick. This is a significant extension of the classical synchronous MoC, which would reject ABO due to the multiple writes to O1 within a tick, and thus would not accept ABO as a valid SyncChart. Furthermore, the SC MoC also allows confluent concurrent writes whose execution order does not matter. This applies to identical writes, such as the assignment O1 = true performed possibly concurrently both in HandleA and in HandleB.

4. High-Level Compilation, Extended SCCharts

SCCharts can be synthesized into hardware and software, and the best approach depends on the characteristics of the execution platform and of the models to be compiled. As SCCharts is a new language, we are still at the beginning of exploring these trade-offs and optimal synthesis strategies. However, we can report on a compilation chain that we have implemented as part of an open-source modeling environment. The compilation begins with a *high-level compilation phase*, described further in this section, which consists of the following steps:

1. Expand Extended SCCharts features (Sec. 4.1), resulting in a Core SCChart.
2. Reduce transition complexity (Sec. 4.2), resulting in a normalized Core SCChart.
3. Map into an *SC Graph* (SCG), annotated with scheduling constraints due to concurrent shared variable accesses (Sec. 4.3).

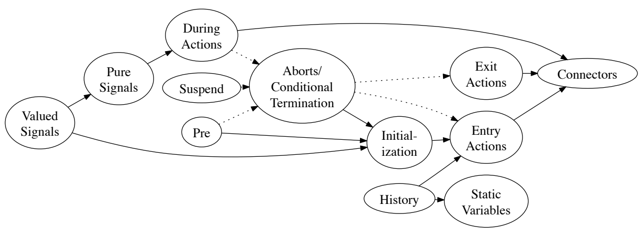


Figure 3. The transformation interdependencies of Extended SCCharts features. When a dependency edge leads from transformation T_1 to T_2, then T_1 must be performed before T_2. A continuous edge means that T_1 produces elements that require subsequent transformation by T_2; e. g., valued signals get transformed into pure signals. A dotted edge means that T_2 cannot handle elements that T_1 must first transform away; e. g., the abort transformation assumes that models do not contain suspension.

4.1 Step 1: Expand Extended SCCharts

Each feature f defined in Extended SCCharts is defined in terms of a transformation rule that expands an SCChart C that uses f into another, semantically equivalent SCChart C' that does not use f. The transformation rules are not only used for synthesis via model-to-model (M2M) transformations, but also serve to unambiguously define the semantics of the extensions. Each such transformation is of limited complexity, and the results can be inspected by the modeler, or also a certification agency. This is something we see as a main asset of SCCharts for the use of safety-critical systems.

Fig. 3 provides an overview of all extensions and their interdependencies. As can be easily seen, the dependencies form a partial order, i. e., there are no cycles. Thus Extended SCCharts can be compiled into equivalent Core SCCharts in a single pass.

Extended SCCharts are quite rich and include, for example, all of the language features proposed for SyncCharts [1]. Not all extensions may be of equal use for all applications, and a tool smith might well decide to not support all features in an SCChart modeling tool. E. g., (valued) signals, the pre operator, suspension, and history could all be omitted without affecting the other elements of Extended SCCharts and their transformations.

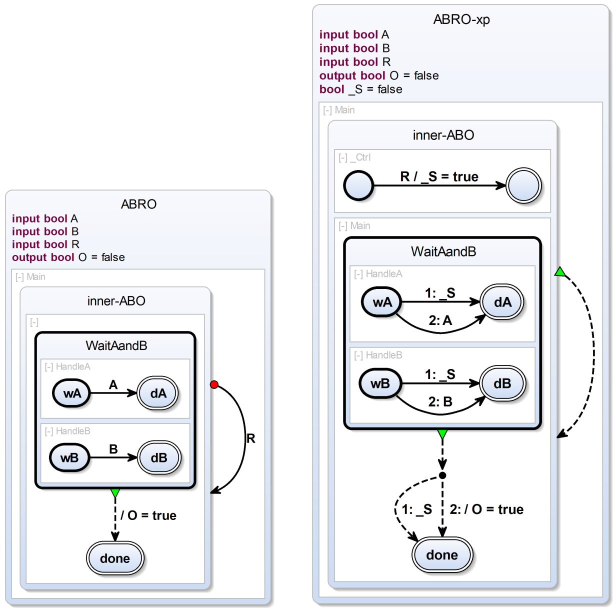
For space considerations, we here merely provide one example of one of the more important extensions, namely aborts. The ABRO SCChart (Fig. 4a), the "hello world" of synchronous programming, compactly illustrates concurrency and preemption. The reset signal R triggers a *strong abort* of the superstate inner-ABO, which means that if R is present, inner-ABO is instantaneously re-started. ABRO gets expanded into the equivalent ABRO-xp (Fig. 4b), which just uses Core SCCharts features. In inner-ABO, a new region _Ctrl sets an internal "strong abort flag" _S in case the abort trigger R becomes true. This flag then prompts a termination of inner-ABO, followed by a self transition.

This transformation approach for aborts follows the "write-things-once" (WTO) principle, in that the original trigger R appears only once in the transformed model. It adds a constant model increase per original (super-)state and per abort transformation, which makes it a useful default strategy that scales well.

As an alternative transformation approach for aborts, one could also replace _S directly by R, and we would not require an explicit _Ctrl-thread anymore, which would result in a more compact ABRO-xp. However, in case of multiple, possibly complex trigger conditions, this trigger propagation might lead to a larger model increase.

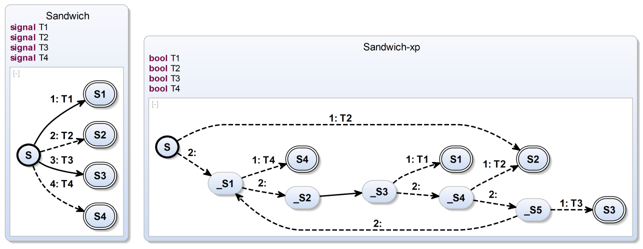
4.2 Step 2: Normalize SCCharts

To facilitate Step 3, the mapping to the SCG, we *normalize* Core SCCharts such that states have only certain "primitive" outgoing



(a) Original ABRO (b) After expansion: ABRO-xp

Figure 4. ABRO, illustrating the transformation of a strong abort (triggered by R) into an equivalent Core SCChart.



(a) Before (b) After normalization

Figure 5. The normalization of Sandwich requires duplication of an immediate trigger (T2), because there is a delayed trigger of lower priority (T3).

transition patterns. The resulting *normalized SCCharts* can be broken down into the primitive patterns shown in the top part of Fig. 6. A *thread* is a set of states connected through transitions with one initial state and an arbitrary number of final states. A *parallel* is a superstate that contains one or more threads. A *conditional* is a state with two outgoing immediate transitions, one of which has a trigger. An *assignment* is a state with one outgoing immediate transition that only has an effect. A *delay* is a state with one outgoing delayed transition, without any trigger or effect.

If a superstate S does not have an outgoing termination transition, normalization adds a normal termination transition from S to a new, non-final state T. This ensures that join nodes in the SCG have a defined control flow successor. For similar reasons, if a non-final state S does not have any outgoing transitions, we add a delayed self transition. This is equivalent to a halt statement in Esterel.

After these initial steps, normalization transforms states with arbitrary outgoing transition patterns as follows.

1. *Split actions.* For each transition T from some state S_1 to another state S_2, where T has an action A, and T has a trigger

```
1  def void transformTriggerActions(Transition transition ) {
2      if ((( transition . trigger != null || ! transition .immediate)
3              && ! transition .actions.nullOrEmpty) || transition.actions.size > 1) {
4          val targetState = transition .targetState
5          val parentRegion = targetState.parentRegion
6          val transitionOriginalTarget = transition .targetState
7          var Transition lastTransition = transition
8
9          for (action : transition .actions.immutableCopy) {
10             val actionState = parentRegion.createState(targetState.id + action.id)
11             actionState.setTypeConnector
12             val actionTransition = createImmediateTransition.addAction(action)
13             actionTransition .setSourceState(actionState)
14             lastTransition .setTargetState(actionState)
15             lastTransition = actionTransition
16         }
17
18         lastTransition .setTargetState( transitionOriginalTarget )
19     }
20 }
```

Figure 7. Xtend implementation of splitting transition actions.

or is delayed: remove A from T's transition label, create a new target state $\_S$ for T, and create an immediate transition from $\_S$ to S_2 with action A. If T contains multiple actions, create a sequence of new transitions.

2. *Split triggers.* For each state S whose outgoing transitions are not yet in primitive form: create a sequence of conditionals that check the immediate triggers (*surface check*), followed by a delay, followed by another sequence of conditionals that check all triggers (*depth check*) before looping back to the delay. The depth check can reuse triggers from the surface check by looping back not to the delay, but to the first trigger of the surface check for which there exists no delayed trigger with lower priority.

Like the transformation from Extended SCCharts to Core SCCharts, we have implemented normalization as a M2M transformation with Xtend.[1] To illustrate the compact, modular nature of the M2M transformations, Fig. 7 shows the "split actions" transformation described above. The precondition is checked in lines 2 and 3. The original parent region and target state are cached in lines 4–6. The lastTransition variable is set to the original transition first in line 7. When iterating over all effects, the last transition gets updated in line 15 to the last created (line 12) new transition to link all created transitions together. Additional auxiliary connector states between the new transitions are created in lines 10 and 11.

The result of normalizing ABO is shown in Fig. 2c. In ABO, as in most cases, we can construct control flow such that no trigger duplication is necessary when doing the normalization. However, there are counter examples, such as Sandwich (Fig. 5), where the depth check can loop back to (and thus reuse) T4, but not T2. The check of T2 must be duplicated because there is a delayed trigger of lower priority (T3). Thus there must be a surface check (on the path from S to _S2) of T2 that, if not taken, transfers control directly to a test of T4, and a depth check of T2 (on the path from _S2 back to itself) that subsequently tests T3.

4.3 Step 3: Map to SC Graph

Fig. 6 shows the mapping from normalized SCChart components to the SCG. A thread has one entry node, corresponding to the initial state, and one exit node, which corresponds to all final states of the thread. If a thread has no final states, the exit node is not reachable. A delay consists of *surface* and *depth* nodes that are connected through a *tick edge* (dotted). Other edges of the SCG are *sequential edges* (solid) that correspond to ordinary control flow. As part of Step 3, the SCG is also annotated with *dependency edges* (dashed, see Fig. 2d), which indicate scheduling constraints due to shared variable accesses, as specified by the SC MoC [20].

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

	Circuit	Priority
Accepts instantaneous loops	−	+
Can synthesize hardware	+	−
Can synthesize software	+	+
Size scales well (linear in size of SCChart)	+	+
Speed scales well (execute only "active" parts)	−	+
Instruction-cache friendly (good locality)	+	−
Pipeline friendly (little/no branching)	+	−
WCRT predictable (simple control flow)	+	+/−
Low execution time jitter (simple/fixed flow)	+	−

Figure 8. Comparison of circuit vs. priority low-level synthesis approaches. The lower half only applies to software. WCRT, the Worst Case Reaction Time, is the maximal execution time per tick.

The SCG also has a textual representation, the SC Language (SCL). The abstract SCL syntax and its correspondence to SCG elements is also shown in Fig. 6. In addition, SCL has a goto statement that jumps to a label l to accomodate the free control flow permitted by state transitions.

5. Low-level Synthesis I—The Circuit Approach

Once we have mapped an SCChart to its SCG, there are different options for downstream synthesis. An overview comparison is given in Fig. 8. The *circuit approach* is suitable if the SCG (including dependency edges, but excluding tick edges) is acyclic. The basic idea is to generate a netlist, which can then be realized in hardware, or can be simulated in software. This approach, including the requirement to have an acyclic flow graph, is already well established for compiling SyncCharts or Esterel [15]. We differ from the established circuit translation rules for Esterel in two ways: 1) we have simpler translation rules, mainly because aborts (and suspensions) are already transformed away during high-level synthesis, and 2) the SC MoC permits sequential assignments.

See again Fig. 6 for the mapping from SCG elements to circuits, including a textual representation as data-flow code. As we synthesize a netlist where all components are always active, we encode control flow with *guards*. However, instead of creating a guard for every node, we group nodes together into *basic blocks* that consist of nodes with a common control flow entrance and exit node. A guard g is true iff control enters g's basic block in the current tick. For example, the guard for a conditional or an assignment is true iff any of the guards g_{in} of the predecessor nodes (basic blocks) is true. An interesting guard is g_{join}, which indicates whether a set of threads terminates in the current tick. To calculate g_{join}, each thread computes a flag d ("done"), which is true when a thread is done, i. e., the guard of its exit node is true, and a flag m ("empty"), which is true when it has no active delay, i. e., the guards of its surface nodes are all false. For a delay, the guard g_{depth} of the depth node is the registered guard g_{surf} of the surface node.

To permit sequential assignments, we split multiple instances (assignments) of variables apart, akin to SSA [2]. Thus an assignment $x = e$ creates a new instance (wire) x'. The multiplexer that forwards either e or x to x' corresponds to SSA's Φ-nodes. The ordering of the instances must obey the control/dependency ordering induced by the SCG. Recall that non-input variables are persistent, and that we do not make any guarantees about uninitialized variables. Thus, if x is possibly read in a tick before it is written (possibly by the environment, if it is an input variable), it must be initialized from a register that stores the last value of x from the

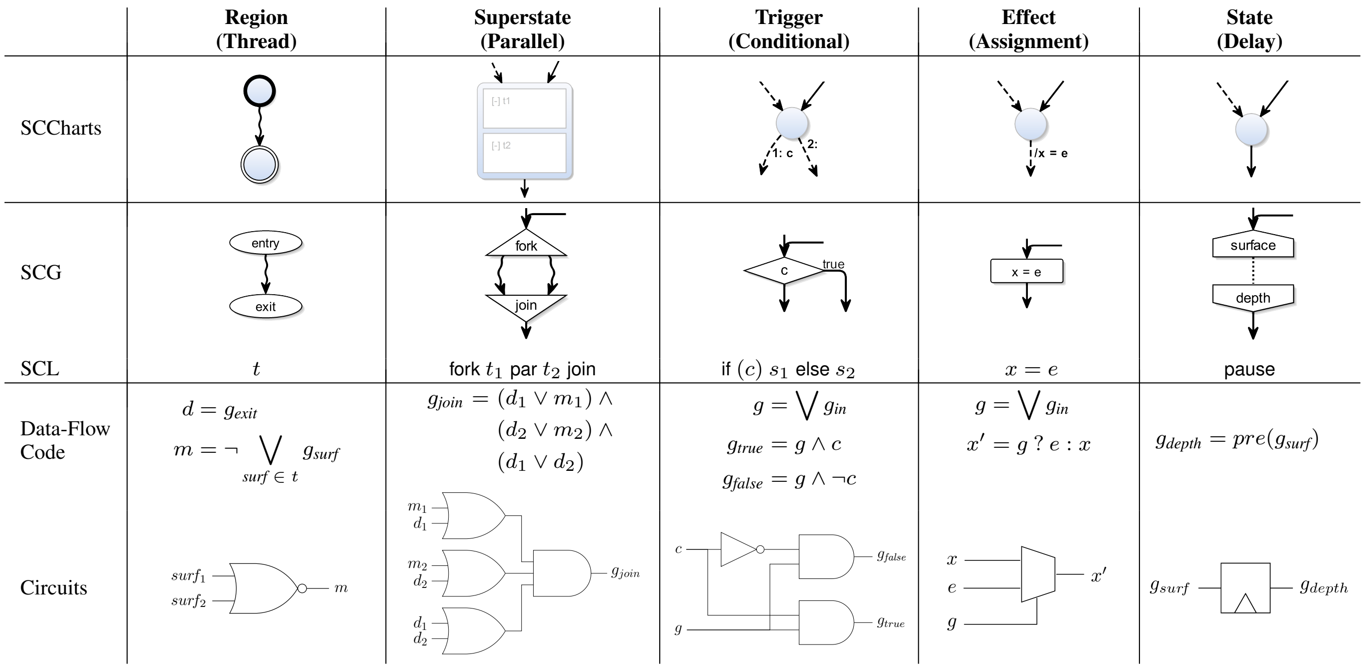


Figure 6. The "synthesis matrix" for SCCharts. The upper part shows the high-level synthesis from normalized SCCharts to the SCG, the lower part shows the circuit approach for low-level synthesis to software or hardware.

previous tick, or, if this is the initial tick for entering the scope of x, some arbitrary constant.

For hardware synthesis, the resulting netlist requires no further considerations. For software synthesis, we must ensure that the data-flow equations are computed in an order compliant with the SCG control/dependency edges. For that purpose, we subdivide basic blocks at each incoming or outgoing dependency edge into *scheduling blocks* that can be scheduled atomically.

For ABO, Fig. 9a shows the SCL code after sequentialization and guard introduction. This still assumes persistent variables (outputs and pause registers) and allows multiple variable assignments, as would be suitable for software synthesis. This SCL code could be mapped directly to a *tick function* in C, with state externalized into the surface guards (g4_pre/g6_pre). A tick function computes a single reaction and is typically embedded in some while loop that iterates in regular intervals. Fig. 9b shows the corresponding VHDL code, after SSA-transformation and explicit registering.

From the initial SCChart to the VHDL (or C), all transformation steps are model-to-model transformations, where we successively replace complex constructs with equivalent, simpler constructs, but stay on the same semantical foundation; the SCG/SCL/C/VHDL artefacts are just different graphical/textual serializations.

6. Low-level Synthesis II—The Priority Approach

While the circuit approach "flattens" all control flow into one sequence of statements, the *priority approach* presented now preserves control flow, including concurrency. While the reaction time of software produced with the circuit approach is proportional to the size of the SCChart, the reaction time of the more software-like priority approach depends only on the components that are active within a tick. Thus the priority approach can scale better to very large models.

The priority approach extends SCL with prioIDs (detailed below), which are used at run time to schedule concurrent threads.

```
module ABO−seq
input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, e2, e6, g2,
  g3, g4, g5, g6, g7, g8;
bool g4_pre, g6_pre;
{
  g0 = GO;
  if (g0) then
    O1 = false;
    O2 = false;
  end;
  g5 = g4_pre;
  g7 = g6_pre;
  g2 = g0 || g5;
  g3 = g2 && A;
  if (g3) then
    B = true;
    O1 = true;
  end;
  g4 = g2 && ! A;
  g8 = g7 && B;
  if (g8) then
    O1 = true;
  end;
  g6 = g0 || (g7 && ! B);
  e2 = ! g4;
  e6 = ! g6;
  g1 = (g3 || e2) &&
      (g8 || e6) && (g3 || g8);
  if (g1) then
    O1 = false;
    O2 = true;
  end;
  g4_pre = g4;
  g6_pre = g6;
}
```

(a) SCL code, permitting multiple assignments per variable.

```
ARCHITECTURE behavior OF ABO IS
−− local signals definition, hidden
begin
−− main logic
g0 <= GO_in;
O1_1 <= false WHEN g0 ELSE O1_pre;
O2_1 <= false WHEN g0 ELSE O2_pre;
g5 <= g4_pre;
g7 <= g6_pre;
g2 <= g0 or g5;
g3 <= g2 and A_in;
B <= true WHEN g3 ELSE B_in;
O1_2 <= true WHEN g3 ELSE O1_1;
g4 <= g2 and not A_in;
g8 <= g7 and B;
O1_3 <= true WHEN g8 ELSE O1_2;
g6 <= g0 or (g7 and not B);
e2 <= not (g4);
e6 <= not (g6);
g1 <= (g3 or e2) and
        (g8 or e6) and
        (g3 or g8);
O1_4 <= false WHEN g1 ELSE O1_3;
O2_2 <= true WHEN g1 ELSE O2_1;

−− Assign outputs
A_out <= A_in;
B_out <= B;
O1 <= O1_4;
O2 <= O2_2;
```

(b) VHDL code (behavioral part), with single assignments per variable/wire. Variable name suffixes "_in/_out" denote inputs/outputs, "x_pre" indicates registered value of x, and "x_i" denotes instance i of x.

Figure 9. Illustration of the circuit low level synthesis approach with ABO.

```
// Boolean type
typedef int bool;
#define false 0
#define true  1

// Enable/disable threads with prioID p
#define _u2b(u)            (1 << u)
#define _enable(p)         _enabled |= _u2b(p);  \
                           active |= _u2b(p)
#define _isEnabled(p)      ((_enabled & _u2b(p)) != 0)
#define _disable(p)        _enabled &= ~_u2b(p)

// Set current thread continuation
#define _setPC(p, label)  _pc[p] = &&label

// Pause, resume at <label>
#define _pause(label)      _setPC(_cid, label); \
                           goto _L_PAUSE

// Pause, resume at pause
#define _concat_helper(a, b)   a ## b
#define _concat(a, b)      _concat_helper(a, b)
#define _label_            _concat(_L, __LINE__)
#define pause              _pause(_label_); _label_:

// Fork/join sibling thread with prioID p
#define fork1(label, p)    _setPC(p, label); _enable(p);
#define join1(p)           _label_: if (_isEnabled(p)) \
                           { _pause(_label_); }

// Terminate thread at "par"
#define par                goto _L_TERM;
```

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

```
int tick ()
{
  tickstart(2);
  O1 = false;
  O2 = false;

  fork1(HandleB, 1) {
    HandleA:
    if (!A) {
      pause;
      goto HandleA;
    }
    B = true;
    O1 = true;

  } par {

    HandleB:
    pause;
    if (!B) {
      goto HandleB;
    }
    O1 = true;
  } join1(2);

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

(b) ABO SCL$_P$ tick function

```
int tick ()
{
  if ( _notInitial ) { active = enabled; goto _L_DISPATCH; } else {
          _pc[0] = &&_L_TICKEND; enabled = (1 << 0); active =
          enabled; _cid = 2; ; enabled |= (1 << _cid); active |= (1
          << _cid); _notInitial = 1; } ;
  O1 = 0;
  O2 = 0;

  _pc[1] = &&HandleB; enabled |= (1 << 1); active |= (1 << 1); {
    HandleA:
    if (!A) {
      _pc[_cid] = &&_L94; goto _L_PAUSE; _L94:;
      goto HandleA;
    }
    B = 1;
    O1 = 1;

  } goto _L_TERM; {

    HandleB:
    _pc[_cid] = &&_L103; goto _L_PAUSE; _L103:;
    if (!B) {
      goto HandleB;
    }
    O1 = 1;
  } _L108: if (((enabled & (1 << 2)) != 0)) { _pc[_cid] = &&_L108;
          goto _L_PAUSE; };

  O1 = 0;
  O2 = 1;
  goto _L_TERM; _L_TICKEND: return (enabled != (1 << 0));
          _L_TERM: enabled &= ~(1 << _cid); _L_PAUSE: active &=
          ~(1 << _cid); _L_DISPATCH: __asm volatile("bsrl␣%1,%0\n
          " : "=r" (_cid) : "r" (active) ); goto *_pc[_cid];
}
```

(c) ABO SCL$_P$ tick function after macro expansion. The flag _notInitial is initially 0, indicating the initial tick.

Figure 10. The tick function synthesized for the ABO example (Fig. 2) with the priority approach. In the initial tick, the tickstart(p) macro initializes the root thread (prioID 0) and sets the prioID of the current thread to p; in subsequent ticks, it only activates the enabled threads. The tickreturn does some bookkeeping: at the _L_TICKEND label, the return value indicates whether any threads other than the root thread are still enabled; at _L_TERM, we disable the current thread; at _L_PAUSE, we deactivate the current thread, retrieve in _cid the active thread with highest prioID (on the x86 and with gcc, we can embed the "Bit Scan Reverse" assembler instruction), and jump to its continuation.

The resulting language, termed *SCL$_P$*, consists of the sequential core of C, plus a set of predefined C macros that emulate reactive control flow (Fig. 10a). Control is not realized with guards for each basic block, as was the case in the circuit approach, but instead with continuation points for each thread. In gcc, this is done with computed gotos; if computed gotos are not available, as in ANSI C, this can be emulated with switch-case logic. Hence SCL$_P$ can be compiled with any ordinary C compiler.

6.1 Thread segment IDs, node priorities, prioIDs

Each thread consists of a set of *thread segments*, which are delineated by fork/join nodes in the SCG and have a *thread segment ID* (*tsID*). Furthermore, each SCG node has a *priority*. SCL$_P$ dispatches threads based on a *prioID*, which combines the node priority with the tsID in a lexicographic fashion. For an SCL$_P$ program with tsIDs in $\mathbb{N}_{<n}$ for some n, prioIDs are computed as *priority* $\times$ n + *tsID*. This induces an ordering dominated by priorities, identical priorities are resolved by tsIDs. Encoding both thread IDs and priorities into a single scalar permits efficient thread book keeping. For example, in the implementation of the SCL$_P$ operators the prioID indexes an array that stores for each thread its continuation, and also indexes bit vectors that keep track of which threads are currently active. The chosen encoding permits on the one hand dynamic priority changes of a thread, and thus instantaneous back-and-forth communication between concurrent threads, and on the other hand still allows a fast dispatching based solely on which thread has the highest prioID.

The priorities and tsIDs must be assigned such that the resulting prioIDs induce a scheduling order that respects the scheduling constraints induced by the SC MoC. This is achieved, e. g., with the priority assignment algorithm [20] that runs in linear time. There is also a _TickEnd thread, with priority 0 and tsID 0, which is always running and manages the return from the tick function.

In ABO, the concurrent access to shared variable B induces one dependence, which can be handled by assigning priority 1 to the nodes in HandleA and in the initial thread segment of the root thread (up to the fork), and priority 0 to the other nodes. In ABO there are no concurrent threads of same priority, hence the assignment of tsIDs does not influence any scheduling decisions; we can thus arbitrarily assign tsID 2 to HandleA and tsID 1 to HandleB. However, we can avoid unnecessary jumps and run-time prioID changes by 1) ordering HandleA before HandleB in the fork/join, so that the thread with lowest prioID comes last and gets to execute the join, 2) propagating HandleA's tsID 2 up to the initial segment of the root thread, and 3) propagating HandleB's tsID 1 down to the final segment of the root thread (from the join onwards). Thus, there are $n = 3$ tsIDs in use, and the resulting prioIDs are $1 \times 3 + 2 = 5$ (HandleB/root), $0 \times 3 + 1 = 1$ (HandleA/root), and $0 \times 0 + 0 = 0$ (_TickEnd).

To minimize storage requirements and to maximize the number threads/priorities that can be encoded with only scalar bit vectors, we compress the range of prioIDs by skipping unused prioIDs. For ABO, shown in Fig. 10b, this results in shifting prioID 5 to prioID 2.

6.2 The SCL$_P$ operators

The SCL$_P$ macros are tickstart(p), which starts the root thread with prioID p; tickreturn, which contains some pausing/dispatching logic and returns 1 or 0 depending on whether the root thread is still running or not; pause, which pauses a thread until the next tick starts. fork$n(l_1, p_1, \ldots, l_n, p_n)$, which forks off n sibling threads with start labels l_i and prioIDs p_i; par, which acts as a thread barrier by terminating a thread; and join$n(p_1, \ldots, p_n)$, which joins sibling threads with n different prioIDs p_i. Note that the join$n$ is not performed by the parent thread, but by one of its child threads; the parent thread does not get started again until its children have terminated. To catch the termination of sibling threads instantaneously, the thread executing the join$n$ must run at a prioID that is lower than that of these siblings. In case a sibling thread to be joined may perform a priority change, it must be considered with all prioIDs that it may take on. A further operator, not required in ABO, is prio(p), which allows to change the prioID of a thread.

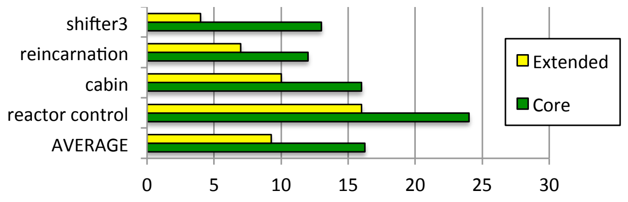
When a parent thread forks off child threads, the child thread that is started immediately after the fork gets to reuse the tsID of the parent thread. Similarly, after the join, the resuming parent thread reuses the tsID of the child thread performing the join. This implies that the forking of a single thread requires neither an extra tsID nor the associated bookkeeping information (resumption address etc.), which is one of the aspects where SCL$_P$ is more efficient than the original Synchronous C.

Fig. 10 shows the SCL$_P$ version of ABO, along with selected SCL$_P$ macro definitions and the result of macro-expanding ABO (gcc -E). The continuation point for the thread with prioID p is stored in _pc[p]. The prioID of the currently running thread is _cid. The threads that still have work to do in the current tick are represented in a bit-vector active, similarly enabled indicates threads that have not terminated yet. Thus the enabled bits are the inverse of the m ("empty") flags computed in the circuit approach. The macros presented here represent the bit vectors with scalars, thus the word size limits the maximum prioID; we have also developed alternative macros that use arrays instead and thus do not have that limitation.

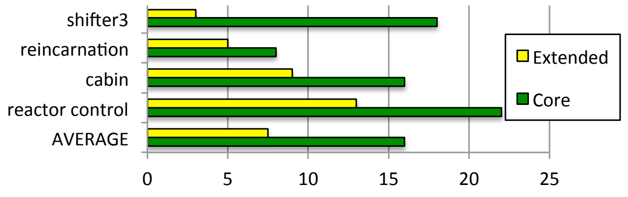
7. Implementation and Experiments

As this paper is about a new modeling language, there are only limited quantitative comparison points. However, there are some questions to ask that warrant practical experimentation. To answer these, we have implemented the compilation stacks just described as part of an Eclipse-based open-source modeling environment. All phases, from Extended SCCharts to C/VHDL, are implemented as model-to-model transformations written in Xtend. SCCharts and intermediate SCChart/SCG models are visualized with KLighD[2], with fully automatic layout; this includes all SCChart/SCG figures in this paper.

First of all, the question of correctness. As stated in Sec. 2, SCCharts are a conservative extension of SyncCharts. Thus valid SyncCharts should also be acceptable (schedulable) as SCCharts, and they should behave the same. To that end, we have collected >100 validation benchmarks with input and output traces during the course of developing SCCharts and the transformations presented here, and we have validated that the SCCharts compiler does

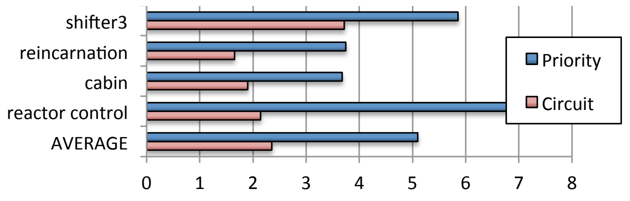


(a) Number of states

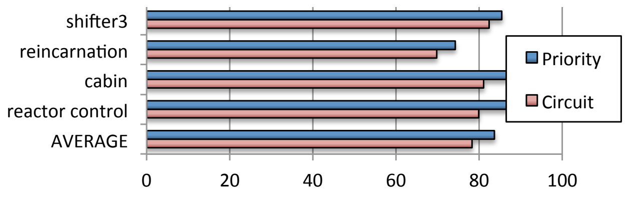


(b) Number of transitions

Figure 11. Comparison of Extended SCCharts with equivalent Core SCCharts resulting from transformations.



(a) Clock cycles per tick



(b) Executable size [KBytes]

Figure 12. Comparison of code synthesis of Extended SCCharts directly to Synchronous C with synthesis to SCL$_P$ via transformations to Core SCCharts.

produce the same result as both another SyncChart-to-Synchronous C compiler [17] and, where traces were available, Esterel Studio.

Another question to ask is how much Extended SCChart models increase when transforming them to Core SCCharts. In Fig. 11 we compare the number of nodes and transitions for some benchmarks suggested by Traulsen et al. [17]. On average, the Extended SCCharts model has 42% fewer states and 53% fewer transitions than the equivalent expanded Core SCCharts model. Thus, the expansion leads to a model size increase, which is not surprising as the main motivation for the SCChart extensions is to express complex behavior in a more compact, abstract manner than is possible with Core SCCharts only. However, the expansions do not cause model size explosion either, which confirms that the Core SCChart operations capture the essence of the MoC. It also suggests that the generic transformations are reasonably efficient, although an expert modeler might in certain cases produce even more compact Core SCCharts.

For the same set of benchmarks, Fig. 12 compares for low-level synthesis the priority synthesis approach with the circuit ap-

[2] http://kieler.cs.cau.de

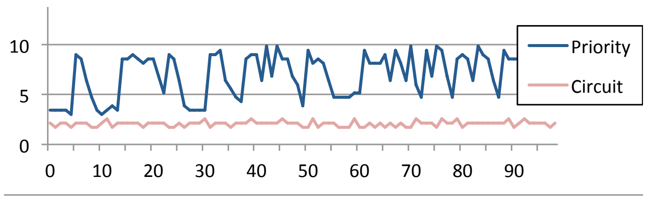


Figure 13. Jitter comparison of low-level synthesis approaches

proach. The measurements were made on an Intel Core 2 Duo P8700 (2.53GHz) architecture. At least for these small-to-medium size benchmarks, the circuit approach is faster; it has to compute more than the priority approach, as it always simulates the whole program, but its very linear control flow structure allows very fast execution. E. g., branching can be implemented with conditional moves rather than conditional jumps, which avoids branch misprediction penalties and thus helps modern, deeply pipelined architectures. The size differences are less significant.

Fig. 13 illustrates for one of the benchmarks (cabin) the execution times per reaction for a sequence of reactions. Not surprisingly, the execution times of the priority approach show a significant variance, i. e., a high jitter. The circuit approach has a much steadier response time (and is faster on average), as there is basically no internal control flow that depends on the inputs and the internal state.

8. Wrap-Up

SCCharts combine the intuitive nature of statecharts with the sequentially constructive model of computation, which naturally extends the sound basis of synchronous concurrency with sequential variable accesses. The core of SCCharts is defined by a very small set of operations, primarily state machines plus hierarchy, where superstates can be left with a join-like termination of their sub-states. Based on these core operations, we derive a number of high-level constructs, notably different types of aborts, through simple model-to-model transformations that largely preserve the write-things-once principle and thus keep the SCCharts compact.

The flexible yet deterministic semantics of SCCharts makes them particularly suitable for safety-critical applications. This is augmented by direct synthesis paths to both software and hardware, which run in linear time, scale well, and where all intermediate steps are open to inspection. We have presented two alternatives for the low-level transformation that map from the SCG to C/VHDL: 1) the circuit approach, which directly maps SCChart elements to circuit elements or code and which is a sensible default strategy in most cases, and 2) the priority approach, which produces software only, but accepts a larger class of programs and for very large programs has asymptotically better performance. Many further details, including detailed descriptions of the Extended SCChart features and of the synthesis approaches, were omitted here for space constraints but are found in an extended report[3].

Future work includes further exploration of model-to-model optimizations, applying our synthesis approach to Esterel-like languages, optimized tsID/priority assignments, experimentation with industry-scaled applications, and further improvements to the automatic layout of SCCharts and SCGs.

References

[1] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[2] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4): 17–20, Apr. 1998. ISSN 0362-1340.

[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] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press. ISBN 0-262-16188-5.

[5] P. Caspi, J.-L. Colaço, L. Gérard, M. Pouzet, and P. Raymond. Synchronous Objects with Scheduling Policies: Introducing safe shared memory in Lustre. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 11–20, Dublin, Ireland, June 2009. ACM.

[6] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, pages 173–182, New York, NY, USA, Sept. 2005. ACM.

[7] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, pages 73–82, Seoul, South Korea, Oct. 2006. ACM.

[8] S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[10] P. B. Hansen. Java's insecure parallelism. *SIGPLAN Not.*, 34(4):38–45, Apr. 1999. ISSN 0362-1340.

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[12] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4): 293–333, Oct. 1996.

[13] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[14] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Oct. 1991.

[15] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*, volume 86. Springer, P.O. Box 17, 3300 AA Dordrecht, The Netherlands, May 2007. ISBN 0387706267.

[16] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society.

[17] C. Traulsen, T. Amende, and R. von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*, pages 563–566, Grenoble, France, Mar. 2011. IEEE.

[18] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.

[19] R. von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, Oct. 2009. ACM.

[20] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. 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, Mar. 2013. IEEE.

[3] Reference omitted to preserve anonymity