SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

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

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth
Kiel University, Germany
{rvh, bdu, cmo, ss}@informatik.uni-kiel.de

Michael Mendler, Joaquín Aguado
Bamberg University, Germany
{michael.mendler, joaquin.aguado}@uni-bamberg.de

Stephen Mercer, Owen O’Brien
National Instruments, Austin, TX, USA
{stephen.mercer, owen.o'brien}@ni.com

Abstract

We present a new visual language, SCCharts, designed for specifying safety-critical reactive systems. SCCharts use a statechart notation and provide determinate 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, determinacy, concurrency, hw/sw-synthesis, safety-critical systems

1. Introduction

Background. Statecharts, introduced by Harel in the late 1980s [14], 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 [30] 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 [15]) 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 determinacy, which is not surprising as statecharts include concurrency and hence are potentially subject to race conditions. Adopting Milner’s distinction of determinacy and determinism [19], we consider a computation as determine if the same sequence of inputs produces the same sequence of outputs, as opposed to deterministic computations, which in addition have identical internal behavior/scheduling. In many application areas, including the area of safety-critical applications that has motivated the work presented here, determinacy 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 determinacy requirement; e.g., STATEMATE, the original statecharts tool, detected potential nondeterminacy at run-time, but not at compile time.

One approach for achieving determinacy, 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 [2]. The synchronous model of computation (SMoC) is a sound approach that solves the determinacy 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 [13], which 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 and Outline. Concerning programming language design, we here present a new, visual modeling language for reac-
SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

2. The SCCharts Language

Fig. 1 shows an overview of the SCCharts syntax. 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.

2.1 Interface Declarations

An SCChart starts at the top with an interface declaration that can declare variables and external functions. Variables can be inputs, read from the environment, or outputs, written to the environment. Variables can also be input/output 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 input outputs 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. In this case, the interface declaration serves for compile-time variable binding/re-naming. 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. We therefore advise to statically (and conservatively) check for such possible uninitialized reads. One way to avoid uninitialized reads is to augment variable declarations with explicit initializations, provided by Extended SCCharts.

### 2.2 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, with a priority p, a trigger t, and an action a, which are all optional, with the syntax “t[p] a” (or “p a”). The trigger is a side-effect-free boolean expression on local and global variables, and the action an assignment $x = e$ (or a sequence thereof) of arbitrary data type.

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, or non-immediate, 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 is always enabled, and we say that the state is transient, because it will always be in the same tick as it is entered.

### 2.3 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 at the end of a tick. 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.

A syntactic complication, solved by different statechart dialects/tools in different ways, is that the “/” may also indicate division. We here 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.

### 2.4 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.” Thus in Core SCCharts 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 also needed for scheduling purposes. 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 interpretation 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. We therefore suggest to require that a region must contain a final state if its enclosing superstate has an outgoing termination transition. However, unlike suggested for SyncCharts \[2\], 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.

### 2.5 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 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 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 GetAB, which is triggered by the termination of regions 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. Note, however, that this behavior is only achieved if the action of the transition to DoneA that writes B is performed before the trigger of the transition to DoneB reads B. In other words, there is a race condition on
2.6 Extended SCCharts

Extended SCCharts are quite rich and include, for example, all of the language features proposed for SyncCharts [2]. Not all extensions may be of equal use for all applications, and a toolsmith 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.

A report describes Extended SCCharts in full detail [32]. For space considerations, we here merely provide one example of one of the more important extensions, namely aborts. The ABRO SCChart (Fig. 3a), the “hello world” of synchronous programming, compactly illustrates concurrency and preemption. The reset signal R triggers a strong abort of the superstate ABthenO, which means that if R is present, ABthenO is instantaneously re-started. The exact semantics of ABRO is expressed by the equivalent ABRO-xp (Fig. 3b), which only uses Core SCCharts features. In ABthenO, 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 ABthenO, followed by a self transition.

2.7 Sequential Constructiveness

The main goal of the sequentially constructive (SC) MoC is to rule out any race conditions that might induce non-determinacy, without being unnecessarily restrictive. The idea is to enforce a certain protocol on ordering variable accesses, namely the initialize-update-read (iur) protocol – but to do so only for variable accesses that are

![Figure 2. The ABO example, illustrating the Core SCChart features.](image)

![Figure 3. ABRO, illustrating the transformation of a strong abort (triggered by R) into an equivalent SCChart without strong abort.](image)
up to the discretion of a scheduler. I.e., we enforce the iur protocol only for concurrent accesses, and not for sequential accesses.

**Definition 1 (Combination functions).** A function $f(x, y)$ is a combination function on $x$ if, for all $x, y_1, y_2, f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$.

One may construct arbitrarily complicated assignments from which a compiler might try to extract some combination function. However, to facilitate the compiler’s job and to make increments also obvious to the human reader of a program, we recommend to use, whenever possible, augmented assignment patterns (familiar from C/Java) of the form $x = e$, with $f$ being $+, \cdot$, etc., and $e$ being an expression without side effects and not involving $x$ (e.g., $x = 1$).

**Definition 2 (Initializations, Updates, Reads).** An assignment $x = f(x, e)$ where $f$ is a combination function and $e$ an expression not referencing $x$ is an update for $x$, of type $f$. Other assignments $x = e$ are initializations of $x$. A trigger expression $e$ or an assignment $x = e$ is a read of every variable $y$ referenced by $e$, unless $x$ and $y$ are the same variable and the assignment is an update.

Note that an update $x = f(x, e)$ does not count as a read of $x$ although $f(x, e)$ references $x$. A transition (trigger and action) can contain several read and write accesses.

We also refer to initializations as absolute writes and to updates as relative writes. The motivation for distinguishing absolute and relative writes is two-fold. First, the concept of relative writes allows to merge concurrent writes, if they are relative and of the same type, in a determinate fashion. This has been used in the past e.g. for parallelizing computations [25], or also in Esterel, with the combine operators, which are a slightly restricted form of our combination functions (Esterel requires commutativity, which we don’t—consider, e.g., subtraction). Second, we allow both an initialization and updates to the same shared variable, while maintaining determinacy by ordering the initialization before the update. This allows to emulate Esterel-style signals fairly easily. In a nutshell, we encode signals with booleans, with signal absence encoded by false and signal presence by true; a signal $s$ is initialized to false, and updated to true with $s = s \lor t$. The iur protocol assures that signals are initialized before they are emitted (set to true), and that signals are emitted before they are tested for presence. Having explicit access to signal initialization allows, for example, to handle so-called reincarnated signals very efficiently, as detailed elsewhere [1].

**Definition 3 (Static concurrency).** A pair of variable accesses is (statically) concurrent, if they belong to concurrent regions of some superstate $S$ (including arbitrary superstate nesting).

**Definition 4 (iur scheduling relations).** Given two statically concurrent accesses $n_1, n_2$ on some variable $x$, we define relations

- $n_1 \leftrightarrow_{ww} n_2$ iff $n_1$ and $n_2$ both initialize $x$ or both perform updates of different type. We call this a $ww$ conflict.
- $n_1 \rightarrow_{ir} n_2$ iff $n_1$ initializes $x$ and $n_2$ reads $x$.
- $n_1 \rightarrow_{iu} n_2$ iff $n_1$ initializes $x$ and $n_2$ updates $x$.
- $n_1 \rightarrow_{ur} n_2$ iff $n_1$ updates $x$ and $n_2$ reads $x$.
- $\rightarrow_{iur} = \mathrm{def} \leftrightarrow_{ww} \cup \rightarrow_{ir} \cup \rightarrow_{iu} \cup \rightarrow_{ur}$. This summarizes the constraints induced by concurrent accesses.

We also refer to these relations as $iur$ ($ww$, $ir$, $iu$, $ur$) order. This order refines the standard write-before-read requirement of synchronous languages, with the key difference that we here restrict our attention to concurrent accesses. Note that these orders must not be acyclic (antisymmetric); for example, due to the symmetry of the $ww$ order, a $ww$ conflict induces an iur cycle.

**Definition 5 (Sequential order, instantaneous order).** Given accesses $n_1, n_2$, we define

- $n_1 \rightarrow_{seq} n_2$ iff there is an instantaneous sequential control flow path $n_1 \rightarrow n_2$ (sequential order).
- $\rightarrow_{ins} = \mathrm{def} \rightarrow_{seq} \cup \rightarrow_{iur}$ (instantaneous order).

The sequential order becomes apparent in the SCG (Sec. 3.3). For example, it is $n_1 \rightarrow_{seq} n_2$ for accesses $n_1, n_2$ if 1) $n_1$ is in the trigger and $n_2$ is in the action of some transition label, or 2) if $n_1, n_2$ belong to two sequentially ordered assignments within one action, or 3) if there is a state $S$ where $n_1, n_2$ is part of a transition that enters $S$ and $n_2$ is part of a transition with immediate trigger that leaves $S$. Note that in the presence of immediate self-loops on superstates, accesses may be sequential as well as concurrent to each other.

We also assume that (i) within an assignment, all read accesses are executed before the write, and (ii) each assignment is executed atomically with respect to variable accesses from concurrent regions.

The $\rightarrow_{ins}$ order is a conservative approximation of all scheduling constraints within a tick, which leads to the following conservative approximation of sequential constructiveness.

**Definition 6 (Acyclic Sequential Constructiveness (ASC)).** An SCChart is acyclic sequentially constructive (ASC) if its $\rightarrow_{ins}$ order is acyclic.

**Theorem 7.** An ASC program scheduled according to $\rightarrow_{ins}$ produces deterministic input/output behavior.

This can be seen by inspecting the definitions of sequentiality, concurrency and of the iur scheduling relations. For a formal proof see elsewhere [33].

In ABO, only B has concurrent write/read accesses, in the concurrent regions HandleA and HandleB. Clearly, ABO is ASC. The iur protocol requires 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 deterministic results.

A distinguishing feature of the SC MOC is that it allows arbitrary sequential variable accesses to shared variables within a tick. In ABO, $O_1$ 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.

### 2.8 Sequential Constructiveness beyond ASC

Sec. 2.7 introduced (acyclic) SC in a fairly conservative fashion, based solely on structural, static analysis. We recommend this as a “base line,” meaning that any SCChart compiler should accept models that are ASC as defined above, and that models should be ASC to ensure portability. However, the SC MoC allows to relax this in several ways without losing determinacy:

1. The $\rightarrow_{iur}$ relation can be reduced to a subset of what results from Def. 4. For example, we can drop $n_1 \rightarrow_{iur} n_2$ if (i) $n_1, n_2$ are confluent [33], or (ii) $n_1, n_2$ cannot be run-time concurrent, meaning that their activations are always in separate ticks, or in the same tick but in different activations of their common superstare $S$ (due to an instantaneous self-loop on $S$) and thus sequentially ordered.

2. We can permit $\rightarrow_{ins}$ cycles that solely contain $\rightarrow_{seq}$ edges. Such cycles due to purely sequential control flow do not open any room for scheduler-induced non-determinacy. We refer to such programs (SCCharts) that do not contain any cycles with $\rightarrow_{iur}$ edges as iur acyclic programs.
The SCCharts language has been designed with pragmatics in mind, i.e., modelers should be able to work productively with complex SCCharts. Specifically, the visual syntax is designed such that it lends itself to fully automatic layout. Automatic layout facilitates a separation of model and view, which not only frees the designer from the burden of drawing and modifying diagrams manually, but also permits customized, filtered views. Customized views highlight certain model aspects and omit others, with the user/modeling tool acting as controller. This “MVC pattern” is well-established in SW engineering, and it is just as useful for visual programming. However, as of today, few modeling languages and tools harness this potential. For example, one might hide transition labels to focus on state relations, or during simulation, one may choose to show the contents of superstates only when they are currently active. This guides the modeler and helps to reconcile model complexity with screen/paper real estate limitations.

Compared to SyncCharts, we chose to express transition priorities not in separate labels, which are difficult to place automatically without overlapping with transitions, but instead made priorities part of the transition labels. Also, we show immediate transitions as dashed lines, rather than hash-prefixes of the transition triggers, to reduce clutter and to facilitate the visual detection of possible instantaneous loops.

In addition to the visual SCCharts syntax presented here, we also developed an SCCharts Textual syntax (SCT), not further detailed here. In our SCCharts modeling prototype, one can create a leading, detailed model description in SCT, while a graphical SCChart with filters set by the modeler is continuously synchronized. This combines advantages of textual entry (speed, ease of revision management, etc.) and graphical view (ease of overview, highlighting of state patterns).

### 2.9 SCCharts Pragmatics

3. Generalizing an →

4. Similarly, we can replace →

5. Static cycles might be “false” in the sense that it might be possible to respect → run-time, due to mutual run-time exclusiveness of parts of the cycle. This could be handled with dynamic scheduling, or by conversion into an equivalent, acyclic program.

### 2.9 SCCharts Pragmatics

The SCCharts language has been designed with modeling pragmatics in mind, i.e., modelers should be able to work productively with complex SCCharts. Specifically, the visual syntax is designed such that it lends itself to fully automatic layout. Automatic layout facilitates a separation of model and view, which not only frees the designer from the burden of drawing and modifying diagrams manually, but also permits customized, filtered views. Customized views highlight certain model aspects and omit others, with the user/modeling tool acting as controller. This “MVC pattern” is well-established in SW engineering, and it is just as useful for visual programming. However, as of today, few modeling languages and tools harness this potential. For example, one might hide transition labels to focus on state relations, or during simulation, one may choose to show the contents of superstates only when they are currently active. This guides the modeler and helps to reconcile model complexity with screen/paper real estate limitations.

Compared to SyncCharts, we chose to express transition priorities not in separate labels, which are difficult to place automatically without overlapping with transitions, but instead made priorities part of the transition labels. Also, we show immediate transitions as dashed lines, rather than hash-prefixes of the transition triggers, to reduce clutter and to facilitate the visual detection of possible instantaneous loops.

In addition to the visual SCCharts syntax presented here, we also developed an SCCharts Textual syntax (SCT), not further detailed here. In our SCCharts modeling prototype, one can create a leading, detailed model description in SCT, while a graphical SCChart with filters set by the modeler is continuously synchronized. This combines advantages of textual entry (speed, ease of revision management, etc.) and graphical view (ease of overview, highlighting of state patterns).

### 3. High-Level Compilation

SCCharts can be synthesized into hardware and software, and the best compilation 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 (see Sec. 3.1). The compilation begins with a high-level compilation phase, which consists of the following steps:

1. Expand Extended SCCharts features, resulting in a Core SCChart (Sec. 3.1).

2. Reduce transition complexity, resulting in a normalized Core SCChart (Sec. 3.2).

3. Map into an SC Graph (SCG), annotated with scheduling constraints due to concurrent shared variable accesses (Sec. 3.3).

### 3.1 Step 1: Expand Extended SCCharts

Extended SCCharts provide a set $F$ of extended features, listed in Fig. 4, grouped according to their origins. Each feature $f \in F$ is defined in terms of a transformation rule $T_f$ that expands an SCChart $C$ that uses $f$ into another, semantically equivalent SCChart $C'$ that does not use $f$. More precisely, $T_f$ produces an SCChart not containing $f$ but possibly containing features in $\prod_f \subseteq F$. Also, $T_f$ can preserve a certain set of features $\text{Handle}_f \subseteq F$. Thus, we must perform the transformations in a certain order, indicated in Fig. 4. Specifically, for $f, g \in F$, we must perform $T_f$ before $T_g$ if (i) $g \in \prod_f$ (solid edges), or (ii) $f \notin \text{Handle}_g$ (dotted edges). 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.

The transformation rules are not only used to implement 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.

Considering again the aborts extension (Sec. 2.6), the default transformation approach for aborts follows the “write-things-once” (WTO) principle, as illustrated for ABRO in Fig. 3. E.g., the original trigger $R$ appears only once in the transformed model. This transformation adds a constant model increase per original (super)state and per abort transformation, which makes it a useful default strategy that scales well. For all Extended SCCharts properties, we provide default transformation rules that follow the WTO principle. Thus each extended feature can be transformed away in time linear in model size, and the size of the fully expanded SCChart is linear in the original model size.

As an alternative transformation approach for aborts, one could also replace $S$ directly by $R$. This would not require an explicit $\text{Ctrl}$-thread anymore, which would result in a more compact ABRO-xp. However, this transformation would not be WTO anymore, and in case of multiple, possibly complex trigger conditions, this trigger propagation might lead to a larger model increase than the default transformation.

### 3.2 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 transition patterns as shown in Fig. 5. A thread is a set of states connected through transitions with one initial state and an arbitrary
SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

### Figure 5
The “synthesis matrix” for SCCharts. The upper part shows the high-level synthesis from normalized SCCharts to the SCG, the lower part shows the data-flow approach for low-level synthesis to software or hardware.

<table>
<thead>
<tr>
<th>Region (Thread)</th>
<th>Superstate (Parallel)</th>
<th>Trigger (Conditional)</th>
<th>Action (Assignment)</th>
<th>State (Delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized SCCharts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-Flow Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circuits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image_url" alt="Diagram" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 6
The normalization of Sandwich requires duplication of an immediate trigger (T2), because there is a delayed trigger of lower priority (T3).

1. **Split actions.** For each transition T from some state S1 to another state S2, where T has an action A, and T has a trigger 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 S2 with action A. If T contains multiple actions, create a sequence of new transitions/states.

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.

The result of normalizing ABO is shown in Fig. 6. 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.

### 3.3 Step 3: Map to SC Graph
Fig. 5 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 (see Def. 5).

As part of Step 3, the SCG is also annotated with dependency edges (dashed, see Fig. 2a), which indicate scheduling constraints.

---

2 In synchronous programming, the surface of a (possibly compound) statement refers to its behavior in the first tick of its execution, and the depth refers to its behavior in subsequent ticks.
Figure 7. Comparison of data-flow vs. priority low-level compilation approaches. The lower part only applies to software. WCRT, the Worst Case Reaction Time, is the maximal execution time per tick.

according to the iur protocol due to shared variable accesses (see Def. [4]).

The SCG also has a textual representation, the SC Language (SCL). See Fig. [5] for the abstract SCL syntax and its correspondence to SCG elements. In addition, SCL has a goto statement that jumps to a label \( l \) to accommodate the free control flow permitted by state transitions.

### 4. Data-Flow Low-Level Compilation

Once we have mapped an SCChart to its SCG, there are different options for downstream synthesis. An overview comparison is given in Fig. [7]. The data-flow approach is suitable if the SCG including dependency edges, but excluding tick edges) is data-flow acyclic (see Sec. [2.5]). 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 [20]. 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. [5] for the mapping from SCG elements to data-flow code and circuits. As we synthesize a netlist, 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 \( g_{\text{pre}} \) of the predecessor nodes (basic blocks) is true. An interesting guard is \( g_{\text{join}} \), which indicates whether a set of threads terminates in the current tick. To calculate \( g_{\text{join}} \), 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 \( g_{\text{depth}} \) of the depth node is the registered guard \( g_{\text{surf}} \) of the surface node.

To permit sequential assignments, we split multiple instances (assignments) of variables apart, akin to SSA [3]. 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 \( \Phi \)-nodes. The ordering of the instances must obey the control/dependency ordering induced by the SCG and the iur protocol.

If such an order is not possible, due to a control/dependency cycle, we must reject the program and cannot proceed further. As in other hardware compilation approaches (including compilation of synchronous languages), we cannot synthesize loops directly; these would require preprocessing, such as loop unrolling.

Recall that non-input variables are persistent, and that we do not make any guarantees about uninitialized variables. Thus, if a non-input variable \( x \) is possibly read in a tick before it is written, it must be initialized from a register that stores the last value of \( x \) from the 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. [8] 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 \( g_{\text{pre}} \). A tick function computes a single reaction and is typically embedded in some while loop that iterates in regular intervals. Fig. [8b] 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.
Figure 9. The tick function synthesized for the ABO example (Fig. 3) 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_TERM 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.

5. Priority-Based Low-Level Compilation

While the reaction time of software produced with the data-flow approach (Sec. 4) is proportional to the data-size of the SCchart, the reaction time of the more software-like priority approach presented now depends only on the components that are active within a tick. Thus the priority approach can scale better to very large models. Furthermore, the priority approach targets only software and thus, unlike the data-flow approach, has no trouble accepting control flow cycles, i.e., it accepts all iur acyclic programs (see Sec. 2.5).

5.1 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. SCLP dispatches threads based on a prioID, which combines the node priority with the tsID in a lexicographic fashion. For an SCLP program with tsIDs in N<sub>n</sub> for some n, prioIDs are computed as priority × 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 SCLP operators the prioID indexes an array that stores for each thread its flow of data. The priority approach extends SCL with prioIDs (Sec. 5.1), which are used at run time to schedule concurrent threads. The resulting language, termed SCL<sub>P</sub>, consists of the sequential core of C, plus a set of predefined C macros that emulate reactive control flow (Fig. 2A). Control is not realized with guards for each basic block, as was the case in the data-flow 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<sub>P</sub> can be compiled with any ordinary C compiler.
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. The root thread
corresponds to the SCChart’s root state and gets started when the
program gets started: when the root thread terminates, the whole
program terminates. In ABO there are no concurrent threads of
same priority, hence the assignment of tsiDs does not influence any
scheduling decisions; we can thus arbitrarily assign tsiD 2 to Han-
dleA 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) propagat-
ing 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. [5], this results in shifting prioID 5 to prioID 2.

5.2 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; forkn\((l_1, p_1, \ldots , l_n, p_n)\), 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\((p_1, \ldots , p_n)\), which
joins sibling threads with \( n \) different prioIDs \( p_i \). Note that the
join\(m\) 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\(m\) 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 [31].

Fig. [3] 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 \( ced[p]\). The prioID of the currently running thread is
\( ced\). The threads that still have work to do in the current tick
are represented in a bit-vector active, similarly enabled indicates
targets threads that have not terminated yet. Thus the enabled bits are
the inverse of the \( m \) (“empty”) flags computed in the data-flow
approach. The macros presented here represent the bit vectors with
scalars, thus the word size limits the maximum prioID; we have
also developed alternative macros that use arrays instead and thus
do not have that limitation.

6. 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 the KIELER Eclipse-based open-source modeling en-
vironment. 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 [23], with fully automatic layout; this includes all
SCChart figures in this paper.

First of all, there is the question of correctness. As stated in
Sec. [7] SCCharts are a conservative extension of SyncCharts. Thus
valid SyncCharts should also be valid SCCharts, with the same be-
havior. To that end, we have collected \( > 100 \) validation benchmarks
with input and output traces during the course of developing SC-
Charts and the transformations presented here, and we have vali-
dated that the SCCharts compiler does produce the same result
as both another SyncChart-to-Synchronous C compiler [29] and,
where traces were available, Estrel Studio.

Another question to ask is how much Extended SCChart models
increase when transforming them to Core SCCharts. In Fig. [10]
we compare the number of nodes and transitions for some bench-
marks suggested by Traulsen et al. [29]. On average, the Extended
SCCharts model has 42\% fewer states and 53\% fewer transitions
than the equivalent expanded Core SCCharts model. Thus, the ex-
pansion leads to a model size increase, which is not surprising as
the main motivation for the SCChart extensions is to express com-
plex 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
modeling in certain cases produce even more compact Core
SCCharts.

For the same set of benchmarks, Fig. [11] compares for low-le-
scale synthesis the priority synthesis approach with the data-flow
approach. The measurements were made on an Intel Core 2 Duo
P8700 (2.53GHz) architecture. At least for these small-to-medium
size benchmarks, the data-flow 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

\[ \text{http://www.informatik.uni-kiel.de/rtys/kiefer} \]

\[ \text{http://www.eclipse.org/xtend/} \]
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 determinate 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 data-flow approach, which directly maps SCChart elements to data-flow 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. One characteristic of the SCG-based, step-wise synthesis proposed here is that there are no conceptual breaks between SCCharts and the low-level implementation. This drastically facilitates, for example, the handling of signals and the signal reincarnation problem, which we can handle at the SCChart/SCG level with linear overhead [1], whereas handling signal reincarnation at the Esterel level has potentially quadratic overhead [28]. 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 elsewhere [22].

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

Acknowledgments

This work was supported by the German Science Foundation (DFG HA 4407/6-1 and ME 1427/6-1) as part of the PRETSY project. We also thank our reviewers for their valuable comments and suggestions to improve the presentation of this paper.

References


