INSTITUT FÜR INFORMATIK

SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquin Aguado, Stephen Mercer, and Owen O'Brien

Bericht Nr. 1311

December 2013

ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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

SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications

Reinhard von Hanxleden, Björn Duderstadt,
Christian Motika, Steven Smyth, Michael Mendler,
Joaquin Aguado, Stephen Mercer, and Owen O'Brien

Bericht Nr. 1311
December 2013
ISSN 2192-6247

R. von Hanxleden, B. Duderstadt, C. Motika, and S. Smyth are with the Department of Computer Science, Kiel University, Kiel, Germany.
E-mail: {rvh, bdu, cmot, ssm}@informatik.uni-kiel.de
M. Mendler and J. Aguado are with Bamberg University, Germany.
E-mail: {michael.mendler, joaquin.aguado}@uni-bamberg.de
S. Mercer and O. O'Brien are with National Instruments, Austin, TX, USA.
E-mail: {stephen.mercer,owen.o'brien}@ni.com

Technical Report

Contents

List of Figures

Abstract

We present a new visual language, SCCharts, designed for specifying safety-critical reactive systems. SCCharts uses a new 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 key features of SCCharts are defined by a very small set of elements, the *Core SCCharts*, consisting of state machines plus fork/join concurrency. Conversely, *Extended SCCharts* contain a rich set of advanced features, such as different abort types, signals, history transitions, etc., all of which can be reduced via model-to-model transformations into Core SCCharts. This approach enables a simple yet efficient compilation strategy and aids verification and certification.

1 Introduction

Statecharts, introduced by Harel in the late 1980s [17], 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, and 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, von der Beeck identified a list of 19 different non-trivial semantical issues, and compared 24 different semantics proposals [32], which did not even include the "official" semantics of the original Harel statecharts (clarified later by Harel [18]) nor the many statechart variants developed since then, including, e. g., UML statecharts with its run-to-completion semantics.

Determinism. One of the semantical issues identified early on for statecharts is the question of *determinism*, which is not surprising as statecharts is a concurrent language and hence 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. A safety-critical reactive system must, given a sequence of input stimuli, always produce the same sequence of outputs, even if the internal behavior involves concurrency.

Synchronous Languages. One approach for achieving determinism, successfully employed by the family of synchronous languages, is to abstract execution time away, hence—at the semantical level—eliminating race conditions, and to require unique variable (or "signal") values throughout an (instantaneous) reaction chain, or *tick*. This is the approach taken, e. g., by Maraninchi's Argos [22] and André's SyncCharts [2].

The synchronous model of computation (MoC) 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 synchronous model of computation (MoC) cannot directly express something like `if (x < 0) x = 0`. This 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, it 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 [34], which harnesses the synchronous execution model to achieve determinis-

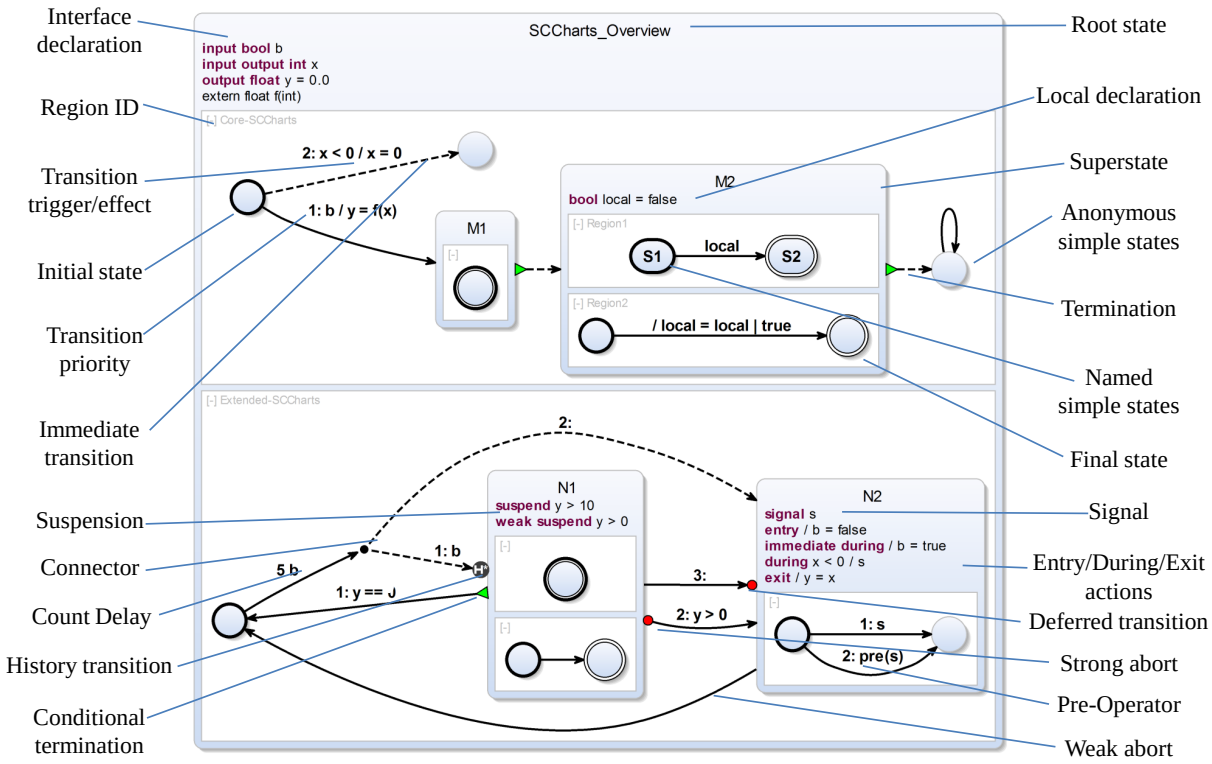


Figure 1.1: Syntax overview. The upper region contains Core SCCharts elements only (Sec. 3), the lower region illustrates Extended SCCharts (Sec. 4).

tic 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.

1.1 Contributions

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, as 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 that can

be reduced to Core SCCharts through a sequence of model-to-model transformations. Each transformation is of limited complexity and open to inspection by the modeler.

1.2 Outline

After summarizing related work in the next section, we present Core SCCharts in Sec. 3. There, the semantics of SCCharts are defined by a mapping to the SC Graphs [34] that define the SC MoC. Sec. 4 gives an overview of Extended SCCharts features that are classfied into three categories: 1. Statecharts [17] features, 2. SyncCharts [2] features, and 3. further features from other synchronous languages. These are discussed in the following sections. Statecharts features like entry actions, exit actions or strong and weak preemption are discussed in Sec. 5. Sec. 6 explains how to encompass SyncCharts features like signals or suspension. Finally, Sec. 7 describes further features borrowed from other synchronous languages like weak suspension from Quartz [27] or deferred transitions from SCADE [10]. We present experimental results in Sec. 8 and conclude in Sec. 9.

2 Related Work

The proper handling of concurrency has a long tradition in computer science, yet, as argued succinctly by Lee [20], has still 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 aborts [4]. SCCharts have taken much inspiration from André's SyncCharts [2], introduced as Safe State Machines (SSMs) in Esterel Studio. SyncCharts combines a statechart syntax with a semantics very close to the synchronous, textual language Esterel [5]. Colaço et al. [11, 10] 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 [16]. They use an elegant construct that basically refines Boolean clocks into "state clocks." The functional synchronous Lucid Synchrone [10] 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 [3]. In Lucid Synchrone, this is motivated also by the desire to sequentialize external function calls with side effects, such as "`print`." Caspi et al. [9] 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 are not restricted to constructiveness in Berry's sense [5], 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 still perfectly deterministic under SC scheduling but would be rejected by a SyncCharts compiler.

The presentation of the SC MoC by von Hanxleden et al. [34] covers the semantic foundations, proposes an efficient scheduling algorithm, and illustrates the SC MoC with a minimalistic, textual programming language termed SCL. We here make use of a more concrete variant of SCL which also includes scheduling information, to illustrate a possible code synthesis for SCCharts (further details in App. **??**). SCL can be viewed as a light-weight variant of Synchronous C [33] or PRET-C [1], which both also provide deterministic reactive control flow and permit sequential assignments.

Various other approaches with their own admissible scheduling schemes have been considered for statecharts. The three most prominent approaches are due to Pnueli and Shalev [24, 12], Boussinot [7] and Berry and Shiple [29, 5]. None of them considers sequential control flow as SC does.

Edwards [13] and Potop-Butucaru et al. [25] 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. This applies in particular to aborts in combination with concurrency, which we reduce to normal terminations.

Esterel [5, 14] provides deterministic concurrency with shared *signals*. Signals can be written ("emitted") and read ("tested for presence") concurrently. They are *absent* per default, and become *present* in a tick whenever any thread chooses to emit them in the current tick. In this sense, signals can be written to concurrently, but there is no write-write race because any signal emission just sets the signal present, and it does not matter which thread performs this signal emission first or last. Furthermore, within each tick, any signal emissions must be performed before any signal presence tests. Causal Esterel programs on pure signals satisfy a strong scheduling invariant: they can be translated into constructive circuits which are *delay-insensitive* [8] under the non-inertial delay model [23] , which can be fully decided using ternary Kleene algebra [21, 23].

The algebraic transformations proposed by Schneider et al. [28] increase the class of programs considered constructive, but do not permit sequential writes within a tick. The notion of sequential constructiveness introduced here is weaker regarding schedule insensitivity, but more adequate for the sequential memory models available for imperative languages.

Signals in Esterel may also be *valued*, in which case they do not only carry a presence status, but also a value of some type. The emission of a valued signal sets a signal present and assigns it a value. Concurrent emissions of a valued signal are allowed if the signal is associated with a *combination function*. This function must be associative and commutative, which allows to resolve write-write races and ensures a deterministic outcome regardless of the order in which the signal emissions are performed. E. g., consider a valued signal x of type int with combination function + and some initial value x_0; if at some tick two concurrent signal emissions emit x(ex_1) and emit x(ex_2) are performed, which emit x with the values of the expressions ex_1, ex_2, respectively, the resulting value for x will be $x_0 + ex_1 + ex_2$, regardless of the order in which the additions (signal emissions) are performed. The SC MoC adopts this concept of a combination function, and considers such assignments via a combination function as a *relative write*.

Finally, Esterel also has the concept of *variables* that can be modified sequentially within a tick. However, they cannot be used for communication among threads, only concurrent reads are allowed. The variable access mechanism of the SC MoC proposed here can be viewed as a combination of Esterel's signals and variables that is more liberal than either one, without compromising determinism.

Lustre [16], like Signal [15], is a data-flow oriented language that uses a declarative, equation-based style to perform variable (stream of values) assignments. Write-write races are ruled out by the restriction to just one defining equation per variable. Write-read races are addressed by the requirement that, within a tick, an expression is only computed after all variables referenced by that expression have been computed. This requires that the write-read dependencies form a partial order from which a schedule can be derived [26]. I. e., there must be no cyclic write-read dependencies. A *clock calculus* takes account of the fact that not every stream variable is evaluated in every tick. From

the result of this schedulability analysis [6] imperative C or Java code can be obtained. To generate this target code, an SC MoC semantics such as presented here is needed.

Caspi et al. [9] have extended Lustre [16] with a shared memory model. Similar to the admissibility concept used in this paper, they defined a *soundness* criterion for scheduling policies that rules out race conditions. However, they adhere to the current synchronous model of execution in that they forbid multiple writes even when they are sequentially ordered.

Synchronous C, a.k.a. SyncCharts in C [33], augments C with synchronous, deterministic concurrency and aborts. It provides a coroutine-like thread scheduling mechanism, with thread priorities that have to be explicitly set by the programmer. PRET-C [1] also provides deterministic reactive control flow, with static thread priorities.

SHIM [30] provides concurrent Kahn process networks with CSP-like rendezvous communication [19] and exception handling. SHIM has also been inspired by synchronous languages, but it does not use the synchronous programming model, instead relying on communication channels for synchronization.

3 Core SCCharts

Core SCCharts contain the key ingredients of statecharts, namely concurrency and hierarchy. In the following, we describe their language elements and illustrate them with the ABO example (Sec. 3.2), followed by a summary of the SC MoC and the SC Graph (Sec. 3.3), and the definition of the SCCharts semantics via a mapping to SC Graphs (Sec. 3.4).

3.1 Language elements

An overview of the elements of SCCharts is shown in Fig. 1.1. The upper part illustrates Core SCCharts; the lower region contains elements from Extended SCCharts.

3.1.1 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 *inputoutputs* 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.

At the top level, this means that the environment initializes inputs at the beginning of the tick (stimulus), e. g., according to some sensor data, and that 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, but are persistent from one tick to the next. 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 other states than the top-level state. This also allows the modularization of SCCharts, at lower levels, using a 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, as mentioned above, but per default uninitialized, like in C. This means that when a variable v is read before and has not been written since its scope was last entered, 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 are explicit variable initializations as part of their declaration, see App. 5.5.

It is also possible to persist variables across exiting and re-entering their scope, with the static modifier. This is similar to "internal static variables" — as opposed to "automatic variables" — in C.

3.1.2 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. 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* which is indicated graphically by a dashed line. 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 without any trigger, we refer to this transition as *default transition* because it will always be taken. Furthermore, if there are no incoming deferred transitions, we say that the state is *transient* because it will always be left in the same tick as it is entered.

A syntactical detail in transition labels is the handling of "/," which may either indicate division or may separate a trigger from an action. There are different ways to approach this; we here suggest to disambiguate the two cases, 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.

3.1.3 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 region compartments). Conceptually, a region corresponds to a thread. A region gets entered through its *initial state* (thick border), which must be unique to each region. 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 transition*, which gets taken when all regions have reached a final state. Termination transitions may be labeled with an action, but do not have an explicit trigger label; they are always immediate (indicated by the dashed line). Terminations are unconditional, hence there should be at most one outgoing termination,

as in case of multiple normal terminations only the one with highest priority can ever be taken.

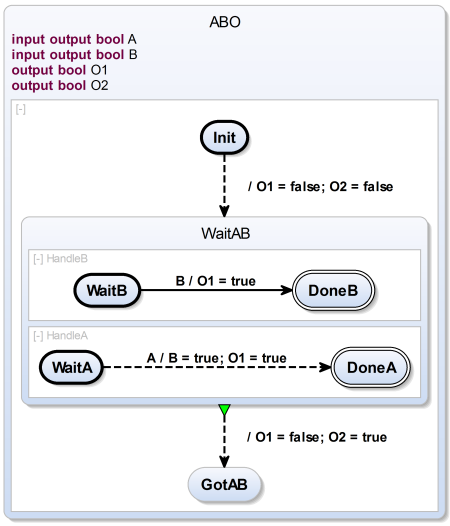
Region termination, final states and termination may seem like straightforward concepts. However, their precise semantics deserves some further discussion, as different interpretations have emerged in the past.

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 later in Extended SCCharts) associated with them; they may have entry actions (also introduced later). Thus final states here have a fairly strong interpretation. The advantage of this is that terminations become very straightforward to implement, 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 normally 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. This choice was rejected here, due to the aforementioned efficiency reasons. However, the weaker interpretation can still be recovered with auxiliary states and signals.

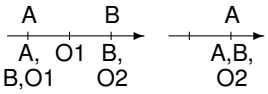
Conversely, a region effectively terminates whenever a region has no during action and reaches a state with no outgoing transitions, no refinements, and no associated actions. However, for clarity, we here require that the state must be explicitly marked as final if we want to enable termination of the 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 normal 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 [2], we argue that one should permit final states even if there is no enclosing normal 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, introduced later with Extended SCCharts, keep getting executed until a state is left.

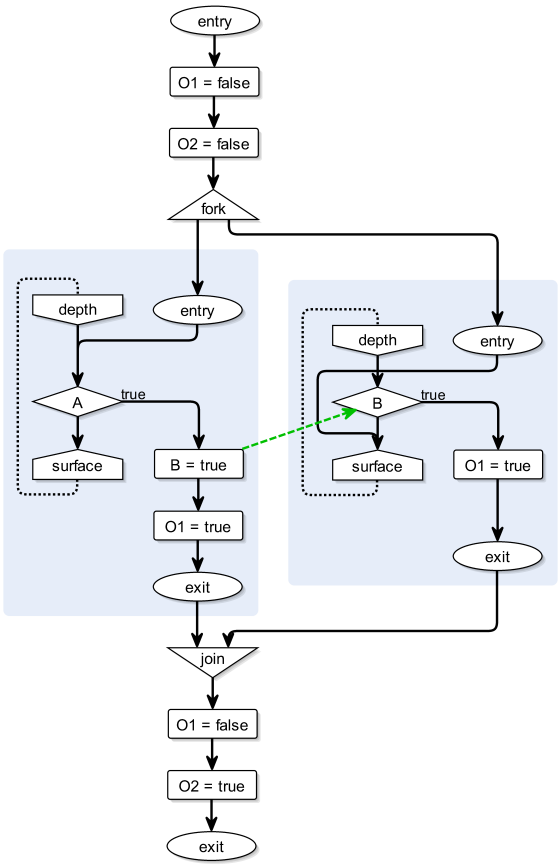
Regarding normal (unconditional) termination, the definition as "a transition that *must* be taken when all regions terminate" correspondes to a simple join operation on concurrent threads, which we consider the most elementary way to leave a macro state. Alternatively, one may consider termination transitions as transitions that *can* be taken when all regions terminate, but whose actual triggering is dependent on some further trigger condition. However, unconditional terminations are semantically much simpler, which benefits transformation rules that build on it and code synthesis. Still, as illustrated later, when discussing more advanced transition types such as aborts and conditional termination (Sec. 5.6), terminations are sufficiently powerful to derive the other transition types.



(a) SCChart ABO



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



(c) The SC Graph. Node priorities are indicated in square brackets, tsID is the thread segment ID.

Figure 3.1: The ABO example, illustrating the Core SCChart features.

3.2 The ABO Example

The ABO example shown in Fig. 3.1a illustrates the concepts of Core SCCharts: ticks, concurrency (with forking and joining), deterministic scheduling of shared variable accesses, and sequential overwriting of variables.

The execution of an SCChart is divided into a sequence of logical ticks. The interface declaration of ABO states that A and B are Boolean inputs as well as outputs. O1 and O2 are Boolean outputs.

The execution of this SCChart is as follows. 1) The system enters state Init and immediately transitions to superstate WaitAB, with a transition action that initializes O1 and O2. WaitAB consists of regions (threads) HandleA and HandleB. Transitioning into a superstate does not trigger transitions nested within that state unless those transitions are immediate. 2) HandleA stays in the initial state WaitA, until the Boolean input A becomes true. Then it sets B and O1 to true and transitions to state DoneA, which is final and hence terminates HandleA. 3) Similarly, WaitB waits for B to become true, sets O1 to true, and transitions to final state DoneB. 4) Once both HandleA and HandleB have

terminated, WaitAB is left, O1 is set to false, O2 to true, and state GotAB is entered. The dashed line denotes the transition to DoneA to be immediate, meaning that HandleA does not pause for a tick before it is ready to detect the transition trigger. In contrast, the transition to DoneB in HandleB is not immediate and thus does not get triggered in any tick in which WaitB is just entered.

Two possible execution traces are shown in Fig. 3.1b. The first trace begins with A set to true by the environment in the initial tick. This triggers the transition to DoneA and sets both B and O1 to true. As this is the initial tick, the non-immediate transition from WaitB to DoneB does not get triggered by the B. In the next tick, all inputs are false, no transitions are triggered, and O1 stays at true. In the third and last tick, B then triggers the transition to DoneB, which sets O1 to true, but sequentially afterwards, O1 is set to false again as part of the transition to GotAB, which is triggered by the termination of HandleA and HandleB. Hence, at the end of this tick, only O2 will be true. 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.

3.3 Sequential Constructiveness

We now briefly recapitulate the basics of the SC MoC, a much more detailed description can be found elsewhere [34]. The basic goal 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; no restrictions are made on sequential accesses. For most programs/models, this understanding should suffice to determine whether a program is SC, and if so, how to schedule it. However, the full story is a bit more involved, as we aim to have a flexible model that, for example, allows us to capture signal-based communication and encompasses all of Berry-constructiveness [5].

3.3.1 Variable accesses, S-admissibility

The SC MoC distinguishes different types of variable accesses. Variable accesses are *confluent* if the order in which they are executed does not matter. If all concurrent variable accesses occurring during a tick were confluent with each other, determinism would be guaranteed. The fact that they are not, in general, forces the synchronous MoCs to impose restrictions on how variable accesses are scheduled. In SC we permit more concurrent accesses within a tick compared to standard approaches and provide a scheduling policy that allows the compiler to either determine conflict-free macro tick schedules or, if that is not possible, to reject a program. Specifically, we organise non-confluent concurrent variable accesses under a strict "initialize-update-read" protocol, based on the distinction between *absolute writes*, *relative writes* and *reads*. *Relative writes* have the form $x = f(x, e)$ where the mathematical function f is so that such assignments are also confluent with each other (e. g., $x = x+1$). These can be executed concurrently and used to implement a distributed update process for variables as a slightly generalized variant

of the *combination functions* present in Esterel and SyncCharts. All other writes, not classified as relative, are *absolute writes*. If the absolute writes are confluent with each other (e. g., constant assignments $x =$ false), they can run concurrently, too, but must be scheduled before the relative writes. Finally, a *read* is any read access to a variable that is not part of a relative write. Reads are always scheduled last since they are not, in general, confluent with either absolute or relative writes. Once a variable is read, it is "owned" by the reading thread in the sense that it cannot be overwritten concurrently anymore; however, it can still be overwritten sequentially.

Based on this classification, SC defines the concept of *S-admissible runs*, as executions that adhere to this "initialize-update-read" protocol for concurrent variable accesses. Only confluent writes are permitted to violate it. A program is called *sequentially constructive*, or *S-constructive* for short, if there exists an S-admissible run and every S-admissible run generates the same deterministic output response. All the compiler has to do is to make sure the run time platform executes S-admissible runs. This can be done by adding priorities as explained below in Sec. 3.3.4.

The runs admissible under the standard synchronous MoC are a superset of those permitted under the SC MoC since the former permit the reordering of sequential composition, e.g., in Esterel which models circuit semantics. In the SC MoC scheme, however, sequential ordering prescribed by the programmer is enforced which leaves less room for non-determinism in the run-time system. Hence, the SC MoC is a conservative extension of the synchronous MoC: all programs accepted by the synchronous MoC are also accepted under the SC MoC, and they will behave the same way. Similarly, SCCharts are a conservative extension of SyncCharts. This implies the code synthesis approach presented here for SCCharts can also be used to compile valid SyncCharts.

3.3.2 The SC MoC in ABO

In ABO, only B has concurrent read/write accesses. S-admissibility 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 the same result. Hence, ABO is sequentially constructive.

A distinguishing feature of the SC MoC is that sequential variable accesses during a tick are allowed. Consider, for example, O1, which 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. 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, which again would be rejected under the classical synchronous MoC.

3.3.3 SC Graphs

The definition of the SC MoC is based on *SC Graphs*, or *SCGs* in short [34]. The SCG for ABO is shown in Fig. 3.1c. An SCG is a pair (N, E), where N is a set of

statement nodes and E is a set of control flow edges. The node types are *entry* and *exit* connectors (ovals), *assignments* (rectangles), *conditionals* (diamonds), *forks* (triangles) and *joins* (inverted triangles), and *surface* (houses) and *depth* (inverted houses) nodes that jointly constitute tick boundaries (corresponding to Esterel's pause statements); one can think of these nodes together as an intersected stop sign. The edge types are *flow* edges (solid edges), which denote instantaneous control flow, *pause* edges (dotted lines), which delineate ticks between a surface/depth node pair, and *dependency* edges (dashed edges), which are added for scheduling purposes.

3.3.4 Determining S-constructiveness, scheduling

In general, determining S-constructiveness is of the same computational complexity as Berry-constructiveness [5], i. e., at least in co-NP [28]. However, S-constructiveness can be conservatively approximated by testing whether a program is *acyclic SC schedulable*, or *ASC-schedulable* in short [34]. ASC-schedulability means that a static schedule exists, expressed as priority assignments to the SCG nodes, that produces deterministic S-admissible runs. This implies that there are no concurrent, non-confluent writes, and that there are no cycles of concurrent writes/reads. Like S-constructiveness, ASC-schedulability considers only concurrent accesses; SCGs that do not contain concurrency are thus trivially ASC-schedulable.

ASC-schedulability can be determined based on the SCG, enriched with data dependency edges that order concurrent variable accesses according to the S-admissibility rules defined in Sec. 3.3.1. This can be done with a longest weighted path analysis algorithm of a complexity that, for this type of graph, is linear in the size of the SCG [34]. The algorithm determines whether a program, represented as SCG, is ASC-schedulable, and hence S-constructive, hence deterministic. If it is ASC-schedulable, then the algorithm produces a static, S-admissible schedule by assigning each SCG node a static *node priority* (not to be confused with the transition priorities introduced in Sec. 3.1.2). If the program is not ASC-schedulable, then it gets rejected. This approach is similar to the general compilation practice for synchronous programs, which most compilers only accept when there exists a static schedule for them. As with synchronous programming in general, the advantage of this type of analysis is that scheduling problems are detected statically, at compile time, and not deferred to simulation/run time.

In ABO, there is one dependency edge induced by the concurrent access to B, which indicates that the assignment B = true in HandleA must be scheduled before the conditional that tests B in HandleB. Hence the assignment gets priority 1, and the conditional remains at the lowest possible priority (0). These priorities get propagated throughout the SCG, resulting in the priority assignment indicated in Fig. 3.1c.

3.4 From SCCharts to SC Graphs

We now define the semantics of SCCharts as a mapping from a Core SCChart C to an SCG $G = (N, E)$, a pair of nodes N and edges E. The mapping is chosen such that

the structure of C gets preserved in G, and such that code can be generated efficiently without too much further analysis. For example, this mapping distinguishes between superstates and simple states, even though semantically the latter could be seen as a specialization of the former. For similar reasons, we directly resolve transient nodes into instantaneous control flow. The SCG shown in Fig. 3.1c results from applying the mapping to the ABO SCChart in Fig. 3.1a.

For each region in C, including the top-level region, an entry and an exit node are created in N. The regions in ABO are HandleA, HandleB, and the top-level region ABO, which is also referred to as the *root state*; hence three entry/exit node pairs are created.

For each superstate M in C with multiple regions, except for the root state, a pair of fork/join nodes is created in N. In ABO, there is one such superstate, WaitAB. Corresponding edges connecting to the entry and exit nodes of the regions inside M are added to E.

Each non-final, non-transient simple state S in C gets translated into a surface/depth node pair in N. In ABO, there are two such states, WaitA and WaitB. For the outgoing transitions of S, the transition triggers get translated into conditionals in N and the transition actions get translated into assignments in N. Edges interconnecting the surface, depth, conditional and assignment nodes get created in E according to the logic expressed by the (non-)immediate nature of the triggers and the transition priorities indicated by the priority labels at the transitions tails. This logic is not detailed further here, but fairly straightforward. A state with a mix of immediate and non-immediate outgoing transitions may necessitate the duplication of some triggers. In ABO, trigger A is immediate, thus A gets tested immediately when entering HandleA. Trigger B is not immediate, hence the entry of HandleB connects to the surface node of WaitB, which makes HandleB pause for a tick regardless of whether the trigger of the outgoing transition is present or not.

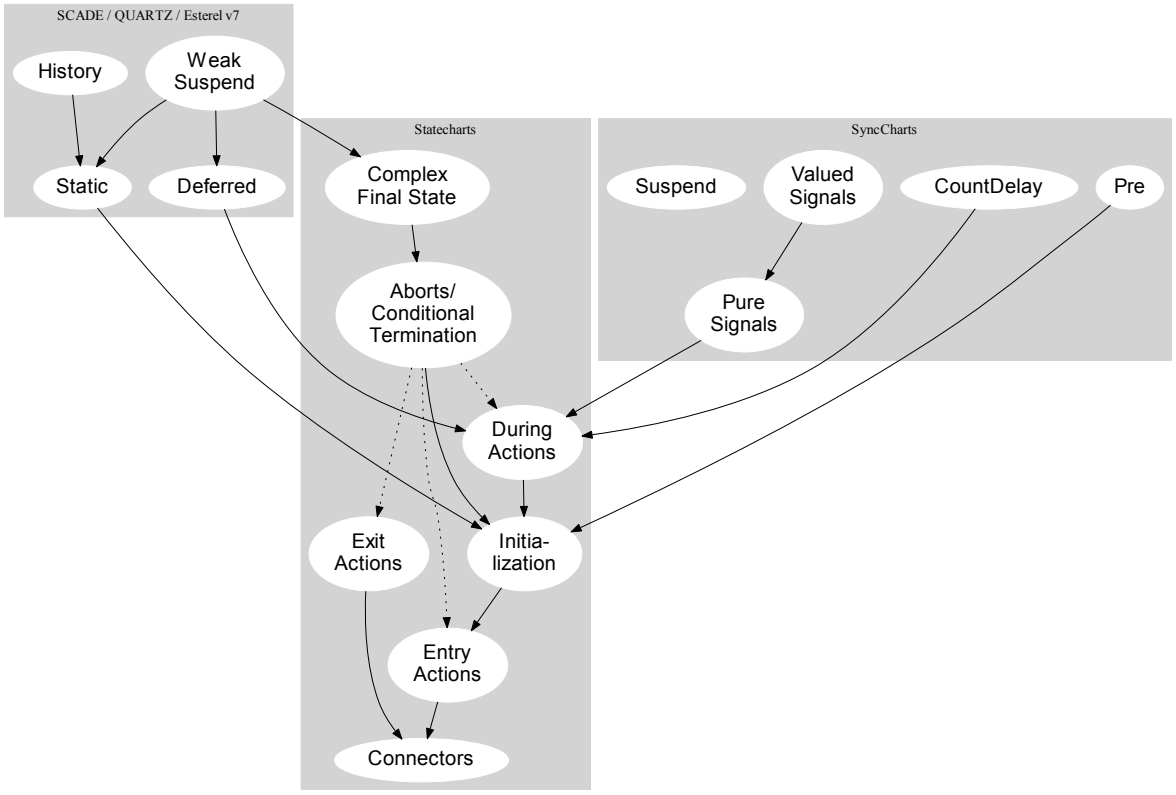


Figure 4.1: Extended SCCharts features and their interdependencies.

4 Extended SCCharts Overview

After introducing Core SCCharts in Sec. 3, we now present a number of extensions that can be derived from Core SCCharts. Each extension is defined in terms of a transformation rule that expands an SCChart C using that extension into another, semantically equivalent SCChart C' that does not use that extension.

The transformation rules not only serve to unambiguously define the semantics of the extensions, but can also be used for code generation through a sequence of primitive model-to-model (M2M) transformations. Each such transformation is of limited complexity, and the results can be inspected by the modeler, or also, e. g., a certification agency. This is something we see as a main asset of SCCharts for the use of safety-critical systems.

Fig. 4.1 provides an overview of all extensions and their interdependencies.

When a dependency edge leads from transformation $T1$ to $T2$, then $T1$ must be

performed before $T2$. A continuous edge means that $T1$ produces elements provided by $T2$; e. g., valued signals get transformed into pure signals. A dashed edge means that $T1$ must precede $T2$ for other reasons; e. g., aborts must not be suspended, or exit actions assume that superstates are left only through termination. 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, provided one adheres to this partial order.

Extended SCCharts are quite rich and include, for example, all of the language features proposed for SyncCharts [2].

We classify all Extended SCCharts features into three categories:

Statecharts features. Common features of various statecharts dialects as known from Harel statecharts [17], e. g., entry actions, exit actions or strong and weak preemption. We will discuss these features in the following Sec. 5.

SyncCharts features. These are features borrowed from Charles André's SyncCharts [2], e. g., synchronous signals or suspension. More details about these transformations are given in Sec. 6.

Further features. We comprised some additional features borrowed from other synchronous languages like weak suspension from Quartz [27] or deferred transitions from SCADE [10]. Sec. 7 will give detailed transformations for these features.

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 harming the other elements of Extended SCCharts. However, we cover all constructs to illustrate that they can be reduced fairly easily to Core SCChart should one want to provide them.

The following three sections give details about all Extended SCCharts features and their transformations as classified above.

5 Classical Statecharts Features

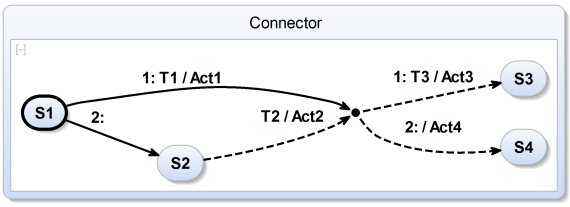
5.1 Connectors

To begin with a fairly simple extension, *connector nodes*, sometimes also referenced as *conditional nodes*, link multiple *transition segments* together to form a *compound transition*. Connectors typically serve to make a model more compact, and to facilitate the *write-things-once* (WTO) principle, without the introduction of further (transient) states. In the Connector example in Fig. 5.1, S1 can transition to S3 or S4; whether a transition is taken or not depends on T1, and if T1 holds, then T3 decides whether S3 or S4 is targeted. Similarly, S2 can transition to S3 or S4. The transition actions Act1, Act2, etc. are placeholders for arbitrary sequences of assignments or (external) function calls; the same applies to actions in later examples.

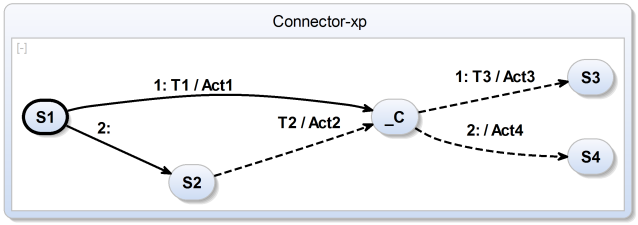
When a state S can be left via a compound transition that begins with some segment TS with trigger T, then T decides whether any transition beginning with TS is taken. In other words, once T holds, there must be a path onwards to some target state S' that is enabled. This avoids the possible need for a "roll back" in case TS or any subsequent transition segment have an actions associated with them. An alternative would be to execute all actions only after a complete transition has been tested successfully, but this would lead to a more complicated semantics. Thus, to avoid a compound transition to "get stuck," all connectors must be transient (see Sec. 3), i. e., must have an outgoing immediate default transition. In Connector, this is the transition from the connector to S4. Because of similar reasoning, all transition segments but the first one are considered immediate implicitly (indicated by a dashed line). Whether a compound transition is immediate or not is thus decided by the initial transition segment.

One approach to transform connectors would be to build the cross product of the associated transition segments, resulting in one transition per possible compound transition. However, this could result in a large model increase. An alternative that we propose here is to simply replace each connector by some state, called _C in the transformed example Connector-xp. As explained, _C must be a transient state that is entered and immediately left again as part of a transition. Therefore, all outgoing transitions must explicitly be made immediate. Thus the outgoing transitions from _C in Fig. 5.1b are set to be immediate explicitly.

Of course, the modeler could have used such a transient state in the first place, instead of using a connector; however, we feel that connectors are still valuable to clarify the nature of the model.



(a) Original SCChart



(b) After expansion

Figure 5.1: The transformation for Connector.

5.2 Entry Actions

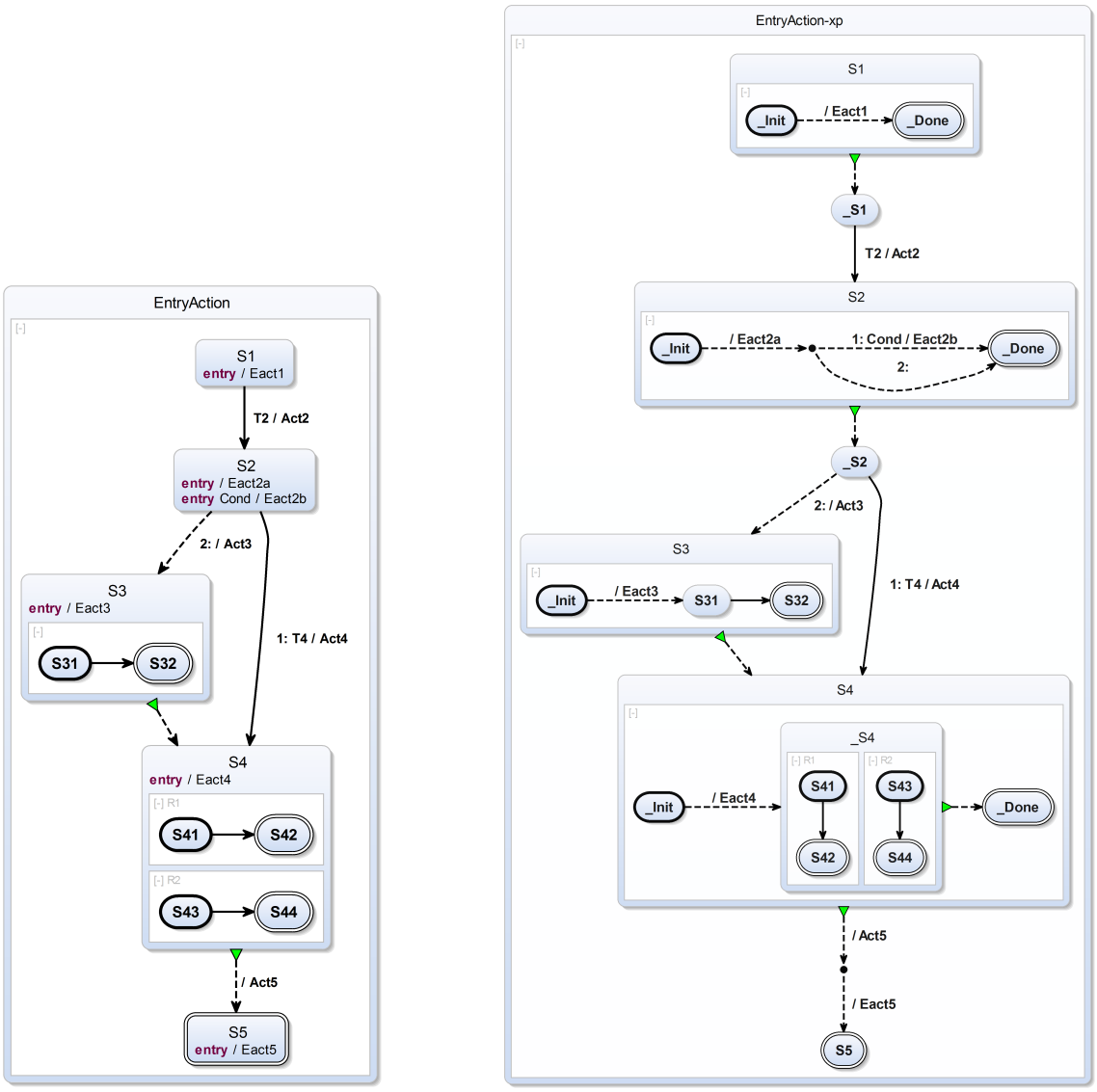
When a state S has an associated entry action A, then A should be performed whenever S is entered, before any internals of S are executed. If multiple entry actions are present, they are performed in sequential order. This differs from during actions, which are performed concurrently to each other because entry actions (like exit actions) can be clearly ordered relative to S, and there is also no immediate/non-immediate issue that would complicate sequential ordering.

A is performed even in case S is immediately left again, including leaving through a strong abort. Thus the entry action transformation should be performed after the abort transformation.

A non-trivial issue when defining the transformation is that we would like allow entry actions to still refer to locally declared variables. Hence we cannot simply attach entry actions to incoming transitions, as these would then be outside of the scope of local variables. Our transformation handles this issue by handling all entry actions within the state they are attached to. This also handles naturally the case of initial states, which do not have to be entered through an incoming transition.

The EntryAction example shown in Fig. 5.2 illustrates the different cases. S1 is a simple state associated with an unconditional entry action Eact1, and an outgoing transition triggered by T2. The entry action gets transformed into a refinement of S1, i. e., S1 becomes a superstate with one internal region. That region immediately executes Eact1 and then terminates. The normal termination transition out of S1 then transfers to a new state _S1 which then waits for the trigger T2 before proceeding further to S2.

S2 is also a simple state, associated with an unconditional entry action Eact2a, followed by a conditional entry action Eact2b. This also gets transformed into a refinement of S2 that sequentially performs the actions and then, analogously to _S1, transfers to a new



(a) Original SCChart (b) After expansion

Figure 5.2: The transformation for EntryAction.

state _S2.

S3 is a superstate with one internal region. In this case, a fresh initial state _Init is introduced that immediately transitions to the original initial state S31 and performs Eact3.

superstate S4 has multiple internal regions. These get encapsulated into a new superstate _S4, and analogously to S3, the action Eact4 gets executed as action immediately performed on a transition originating in a new initial state _Init leading to _S4. To allow normal termination of S4, a final state _Done is added that gets reached whenever _S4 terminates.

S5 is a final state, which may have an entry action associated with it, in this case Eact5. As final states cannot have any internal behavior, as discussed in Sec. 3.1.2, the transformation of their entry actions simply introduces a connector node to which all incoming transitions are connected, and which connects to the final state with a transition segment that performs the entry action.

5.3 During Actions

A non-final state M may be associated with a collection of *during actions*. During actions get executed in a tick whenever M has been active from the beginning of the tick on, meaning, M has been entered in a previous tick. This "non-immediate" default interpretation of during actions avoids multiple executions of during actions within one tick in case of self transitions. To achieve *immediate* during actions, which are useful for example when defining signals (see Sec. 6.1), one must prefix the during keyword with an immediate modifier. Immediate during actions get re-executed every time a self transition is taken. During actions can also be *conditional*, meaning that an additional trigger condition must hold for them to execute.

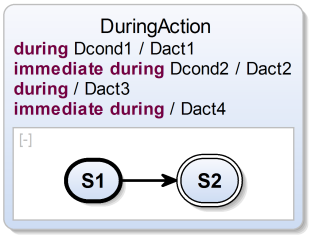
During actions get executed concurrently to any refinement of M, and they get executed concurrently to each other. For variables accessed in during actions, the same scheduling rules apply as for variables accessed concurrently in any regions of M.

The DuringAction state shown in Fig. 5.3a has four during actions associated with it, reflecting the different possible combinations of delayed/immediate and conditional/unconditional during actions. DuringAction and the equivalent DuringAction-xp shown in Fig. 5.3b illustrate the transformation that eliminates during actions. This transformation assures that the During$n$ regions can run without blocking termination of M. It also assures that during actions still get executed in the current tick in case of termination.

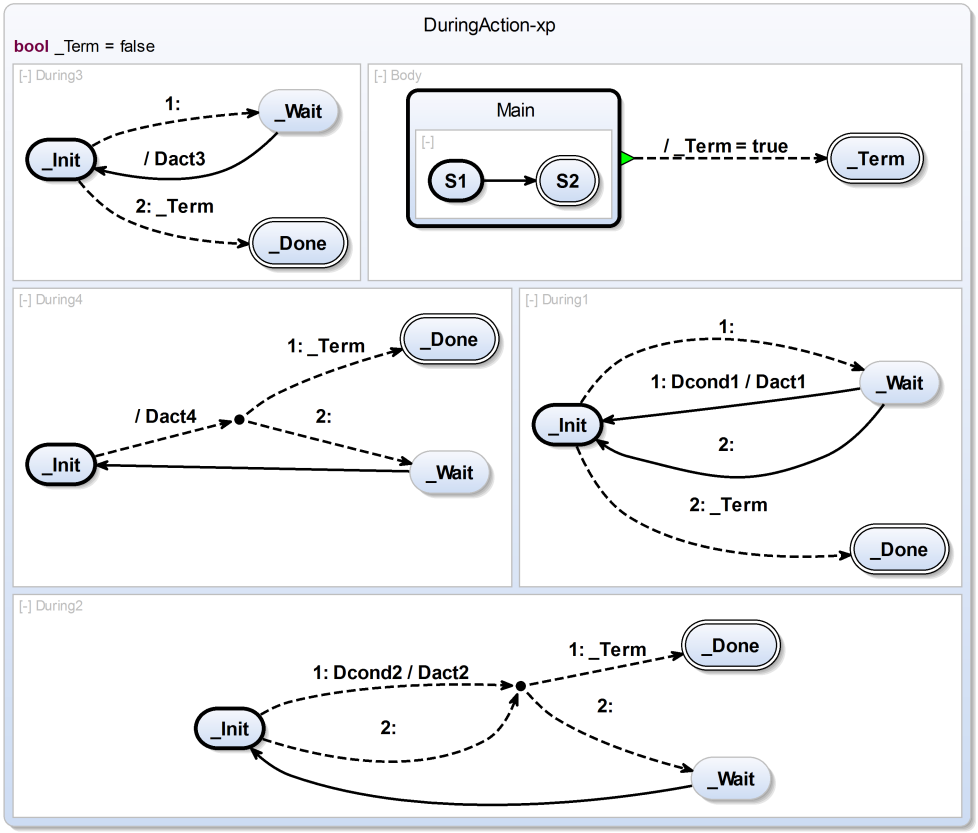
The following procedure transforms a state M associated with one or more during actions into an equivalent state that does not have during actions.

1. If M has an outgoing termination: encapsulate M into a state Main; place that state into a region Body inside M; declare a fresh Boolean variable _Term in M; add a termination transition from Main to a new, final state Term, which sets _Term to true.

2. For the n-th during action D associated with M, where n iterates over all during actions of M: add to M another region During$n$, and in that region, perform the following steps.
   a) Add an initial state Init, and another state Wait.
   b) If D is delayed (no immediate modifier):
      i. Add an immediate transition (dashed line), without further condition, from Init to Wait. Add a transition T with the action of D back from Wait to Init.



(a) Original SCChart



(b) After expansion

Figure 5.3: The transformation for DuringAction.

ii. If D is conditional: make T conditional on the trigger of D, and add another, lower-priority, unconditional transition from Wait to Init. See Region During1 in Fig. 5.3b for an example.

c) If D is immediate (immediate modifier):

i. Add a transition from Wait to Init. Add a conditional node C, with an outgoing transition segment to Wait. Add an immediate transition T with the action of D from Init to C.

ii. If D is conditional: make T conditional on the trigger of D, and add another, lower-priority, unconditional transition from Init to C. See Region During2 in Fig. 5.3b for an example.

d) If M has an outgoing termination: add a final state Done, and a transition to Done triggered immediately by _Term. The source of that transition is Init in the delayed case, and the conditional node C in the immediate case.

We do not have to be concerned about transitions originating in connectors, as the first segment of a compound transition already determines whether a compound transition is taken or not.

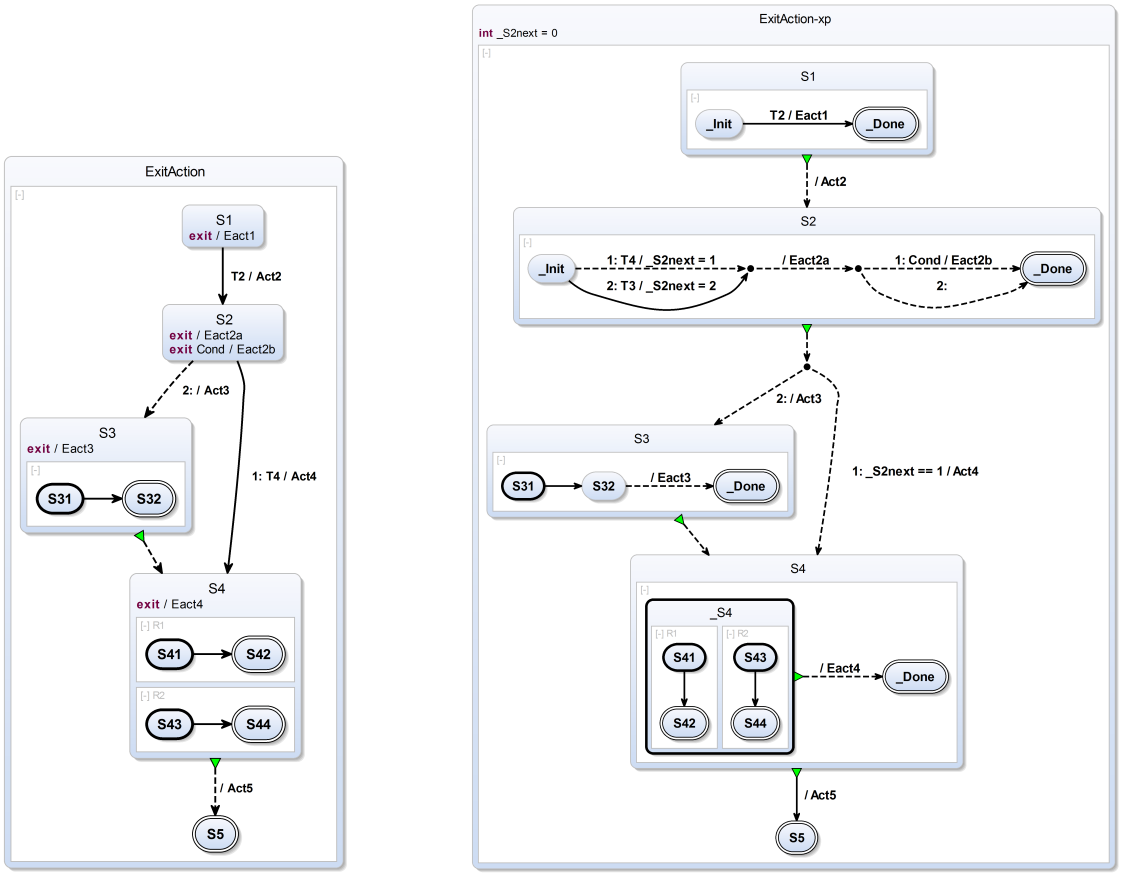
5.4 Exit Actions

A state S may be associated with an *exit action* that is executed whenever S is left, be it by termination or an abort. This exit action is performed after the body of S has possibly been executed, which includes the case of immediate strong aborts, and before any outgoing transition actions are executed. Again, one may have multiple/conditional actions, similar to entry and during actions. As it turns out, the emulation of exit actions is less trivial than the other action types. Quoting André [2]: "Contrary to entry actions, exit actions are not simple factorizations of instantaneous actions. Note that strong and weak aborts have the same effect on exit actions. This explains why exit actions are primitive constructs: they cannot be expressed by a combination of the already studied constructs." However, once aborts have been transformed into terminations, as discussed in Sec. 5.6, we can in fact also derive exit actions in a straightforward manner. After the abort transformation, superstates can only be left by taking a termination transition; this simplifies their handling significantly.

Like entry actions, multiple exit actions are performed in sequential order. The ExitAction example in Fig. 5.4 illustrates different cases. They are similar to the EntryAction example in Fig. 5.2.

The simple state S1 gets transformed into a superstate that waits until the trigger T2 of the outgoing transition becomes enabled. It then performs the exit action Eact1 and terminates, which results in the transition to S2, which performs the action Act2.

One detail here is that T2 has now moved into the scope of S1. If T2 happens to reference a variable V that is re-declared locally at the scope of S1, then T2 would refer to a different variable instance. In this—hopefully rare—case one needs to disambiguate



(a) Original SCChart

(b) After expansion

Figure 5.4: The transformation for ExitAction.

the V instances by giving one of them a fresh name. For safety critical systems, a style guide would probably rule against such variable shadowing in the first place.

The simple state S2 is more complicated in that it has multiple outgoing transitions. Thus we must memorize in an auxiliary variable _S2 the transition to be taken after performing the exit actions. Again the transition triggers have moved inside the state.

S3 and S4 are comparatively straightforward to handle, as they already are superstates that only have one outgoing termination.

5.5 Initializations

An SCChart with variable initializations within variable declarations can be transformed into an equivalent SCChart without variable initialization simply by moving the initialization into an entry action, as illustrated in Fig. 5.5.

This transformation exploits the fact that entry actions do not get moved outside of the state that they are attached to, hence entry actions can also make use of locally

(a) Original SCChart

(b) After expansion

Figure 5.5: The transformation for variable initializations.

declared variables.

5.6 Aborts and Conditional Termination

A hierarchical state can be *aborted* upon some trigger. There are two cases to consider, *strong aborts*, which get tested before the contents of the aborted state get executed, and *weak aborts*, which get tested after the contents of the aborted state get executed. In the Aborts example in Fig. 5.6a, state M is strongly aborted (red circle) whenever the trigger signal Strig occurs and weakly aborted (no circle) by signals W1trig or W2trig. The corresponding abort transition leads to one of the targets Stgt, W1tgt or W2tgt. This abort takes place regardless of which internal state (S1, S2, or S3) is active at the time of an abort. An equivalent behavior could be achieved without hierarchy, by adding such transitions to each of the internal states. However, it is more economical and more robust to specify these aborts only once, at the outer hierarchy level M, in line with the aforementioned WTO principle. In fact, the ability to specify high-level aborts is one of the most common motivations for introducing hierarchy into statecharts. Aborts are thus a powerful means to specify behavior in a compact manner, but handling them faithfully in simulation and code synthesis is not trivial.

Another non-trivial extension to Core SCCharts are *conditional terminations*. Recall from Sec. 3.1.3 that normal (unconditional) terminations are not allowed to have a trigger condition, and are always immediate. Conditional terminations have a trigger condition, and they get taken as soon as all regions of the source state have terminated, *and* the trigger condition holds. The trigger condition may be immediate or non-immediate.

Recall that termination transitions that do not list a trigger condition are always considered immediate (dashed line). To achieve a conditional termination transition that is not immediate but has no other trigger condition one should label it explicitly with true.

In case there exists an unconditional termination that guarantees that a terminating state is left immediately, conditional terminations can be eliminated simply by dispatching from the unconditional termination transition via a connector node to the different termination targets. However, if that is not the case, conditional terminations are semantically more complicated in that they may cause a superstate to still "get stuck"

after termination, which corresponds to an extra state. A naïve approach to handle this could be to add an extra waiting state to be entered via a termination, and to proceed from that state once one of the triggers of the conditional termination transitions hold. However, this would break the proper handling of any during or exit actions. Thus, one has to treat conditional terminations similar to aborts, and it is sensible to fold the transformation of conditional termination transitions into the transformation for aborts, as we do in the following.

Fig. 5.6 illustrates how expanding Aborts results in an equivalent Aborts-xp that does not use aborts anymore. The underlying idea is to make the internal regions of M terminate explicitly whenever M is aborted, and then use a termination transition to leave M. The detection of an abort is done by a new concurrent region Ctrl, which sets one of the newly declared auxiliary flags _S, _W1, _W2, _N1, _N2 whenever the corresponding abort or conditional termination is triggered. These flags are then used to induce the termination of M and to select the corresponding abort target state.

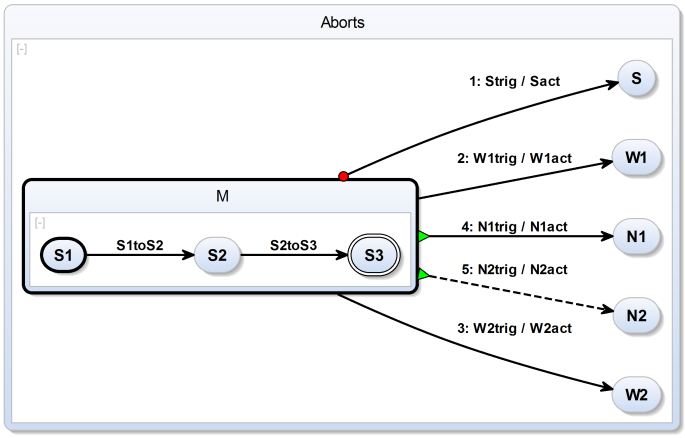
The Ctrl region should not prevent termination of M when M terminates normally, without an abort. When M does terminate normally, Ctrl should terminate as well. Therefore, the transform wraps the regions of M into a new superstate Main in a newly created region Body, parallel to Ctrl When Main terminates, it sets the flag _Term. That flag terminates Ctrl if it is still running.

Another issue is the precedence of aborts and internal behavior of M. One must first test for strong aborts, then perform internal behavior, and then test for weak aborts (see also Sec. 5.6). This is encoded by the transition priorities within Main. For example, if we are in state S1 and Strig triggers a strong abort, we transition directly to Done. If, however, W1trig triggers a weak abort, we first transition to S2, and from there take the transition to Done. Thus, we can bundle together all strong aborts into one transition trigger and can similarly combine all weak aborts, but we cannot mix the two.

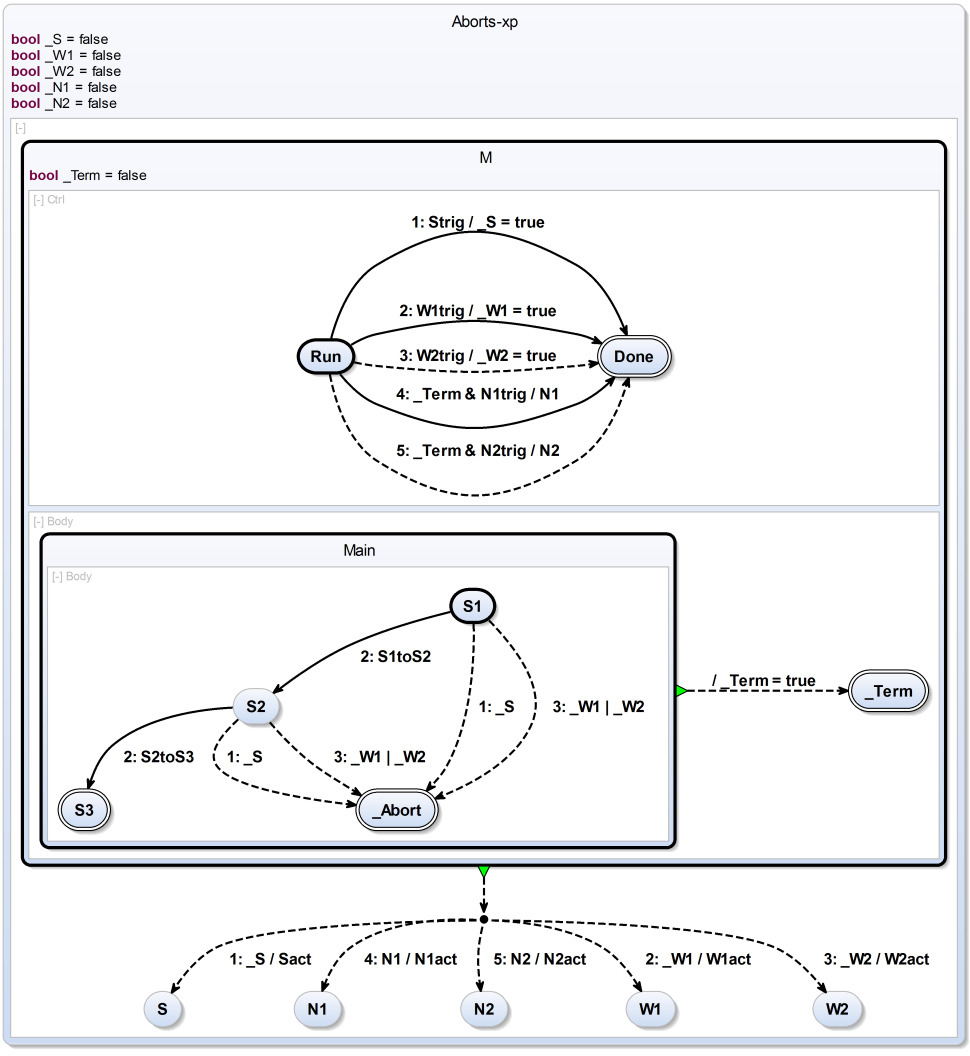
As in SyncCharts and Esterel, strong aborts in SCCharts may cause causality cycles, i. e., concurrent write/read dependency loops, and thus make an SCChart not S-constructive. This is a consequence of introducing concurrency with our transformation, as the transition triggers/effects of the newly introduced Ctrl region interact concurrently with M (now Main). If a transition in M writes to a variable that triggers a strong abort flag _S, this generates a dependency cycle, since we demand that _S must be tested before that transition is taken. However, it is only natural to reject such programs, as they mean that a macro state tries to strongly abort itself, which is not possible. There is no such issue with weak aborts, as weak abort flags are tested only after transitions of M have executed.

The transformation presented here uses auxiliary flags to stick to the WTO principle for the abort triggers. However, if these triggers do not have side effects (e. g., due to function calls) and are not too expensive to be computed, one might also consider to reuse them directly, instead of the auxiliary flags. Similarly, one may not need a new final state Done in Main if there is already a final state that can be entered without side effects, i. e., that does not contain an entry action.

Instead of using these auxiliary flags, we could also have used signals, which would have been slightly more compact in notation. However, as signals translate to variables that



(a) Original SCChart



(b) After expansion

Figure 5.6: The transformation for Aborts.

get re-initialized to false at every tick, and our flags need to be initialized only once when their scope is entered, we here chose the more direct mapping to variables. Furthermore, the generation of signals by the abort-transformation would create a dependency cycle in the order in which the transformations must be applied, see Fig. 4.1.

In case of nested superstates with aborts, this transformation must be applied from the outside in, so that inner aborts can also be triggered by outside abort triggers. Furthermore, the abort transformation assumes that no later transformations adds regions to the aborted state. Hence the abort transformation must take place after the during transformation, presented in the next section, even though the during transformation does not introduce aborts.

We now explain the abort transformation in more detail. Given a superstate M with abort transitions, the transformation into an equivalent state M without abort transitions is as follows.

1. If M has an outgoing termination: encapsulate M into a state Main; place that state into a region Body inside M; declare a fresh Boolean variable _Term in M; add a termination transition from Main to a new, final state Term, which sets _Term to true.

2. Add to M another region Ctrl with initial state Run and a final state Done. If Step 1 introduced a _Term flag, add a transition from Run to Done triggered immediately by _Term.

3. Add a termination transition from M to a newly created connector node C. If there was already some other normal termination transition T from M to some target *Tgt*, with some action A, then replace T by a transition fragment from C to *Tgt* with action A.

4. For each abort or conditional termination transition T originating in M, with some trigger *Tr* (which may or may not be immediate), action A and target state *Tgt*, perform in order of increasing priority (i. e., decreasing transition priority label) of T:

   a) Declare a fresh Boolean flag $\_TgtN$ at the scope enclosing M. N is the empty string if there is just one abort transition from M to *Tgt*; otherwise it is a counter, starting with 1, which disambiguates the abort transitions to *Tgt*. Depending on the type of abort, call this flag a *strong* or *weak abort* flag.

   b) In Ctrl, add a transition from Run to Done of higher priority than the existing transitions from Run to Done, setting $\_TgtN$ to true. For aborts, the trigger of this transition is just *Tr*; for conditional termination, *Tr* is disjuncted with _Term.

   c) Replace T by a transition fragment from C to *Tgt*, triggered by $\_TgtN$, with action A. Since the abort flags exclude each other, the priority of this transitions fragment is irrelevant, except that its priority must be higher than the transition fragment possibly created in Step 3. As connectors must always

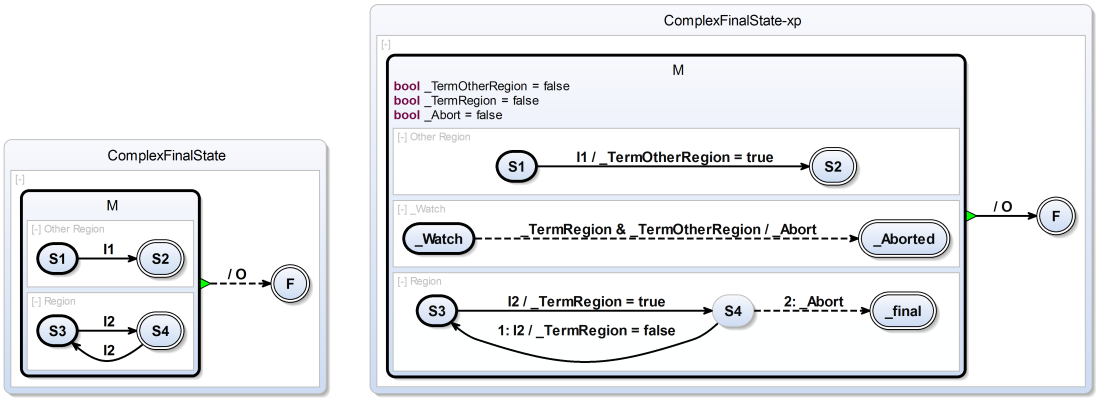
have an outgoing default transition that does not have any condition, remove the trigger condition from the lowest-priority outgoing transition, if there is any such trigger; this will be the case if M originally did not have a termination transition that cannot have a trigger and thus already serves as such a default transition.

Note: if M has only one outgoing transition, it suffices for the transformed model to have only one termination transition from M to Tgt. Neither a connector node C nor an auxiliary flag $\_Tgt$ is required.

5. For all regions R of M:
   a) If R does not contain a final state without entry action yet, create a fresh, final state _Aborted. Let F be a final state of R without entry action.
   b) For all non-final states S in R:
      i. If there are any strong aborts, add a transition from S to F with highest priority, triggered immediately by the disjunction of all strong abort flags. If S is a superstate, this transition must be a strong abort transition.
      ii. If there are any weak aborts, add a transition from S to F with lowest priority, triggered immediately by the disjunction of all weak abort flags.

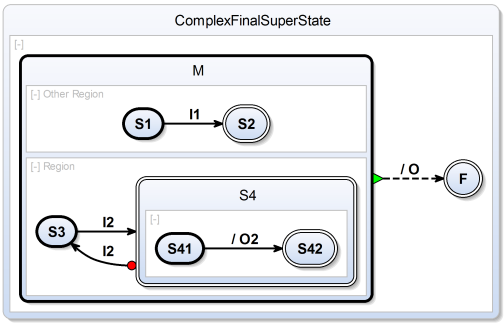
5.7 Complex Final States

When a final state is entered this is means that the corresponding thread terminates. Hence, typically outgoing transitions from final states and final superstates are not allowed. We here illustrate how this functionality can still be captured with a semantics-preserving M2M transformation that changes such final states to normal states and fulfills the necessary bookkeeping.

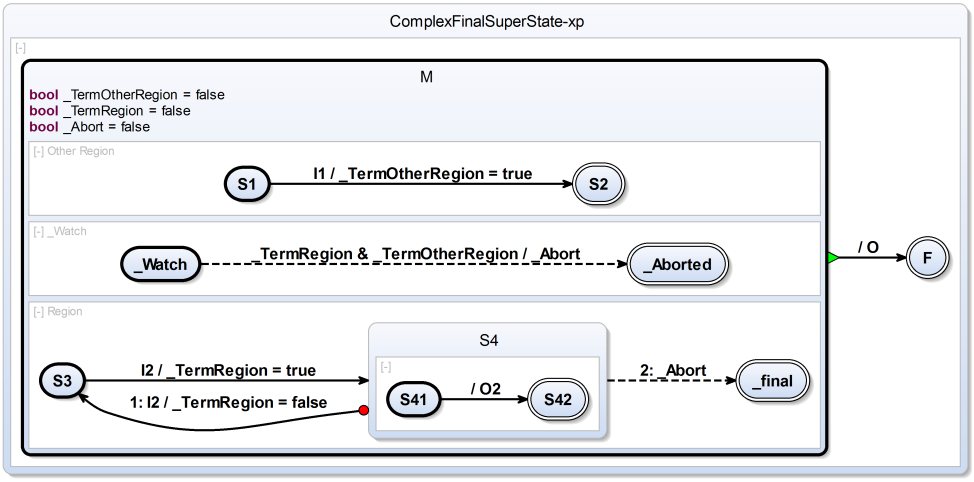


(a) Original SCChart (b) After expansion

Figure 5.7: The transformation for complex final states.



(a) Original SCChart



(b) After expansion

Figure 5.8: The transformation for complex final superstates.

For a superstate M, if there is any final state F with outgoing transitions (or F is a final superstate or F has entry, during, or exit actions) then for each region R_M of M create a termination variable $\_TermR_M$ in M that is initialized with false. For any region R_M add an action $\_TermR_M = true$ to all transitions that have a final state target. For a final state F (with outgoing transitions or if F is a final superstate or F has entry, during, or exit actions) of region R_F in M add $\_TermR_F = false$ to all outgoing transitions and $\_TermR_F = true$ to all incoming transitions of F (not considering self loops). Create a watcher region with an initial state, a final state, and an immediate transition from the initial to the final state setting a new $\_AbortF$ variable (declared in M) to true iff all $\_TermR_M == true$ and $\_TermR_F == true$. Create a new final state $\_A_F$ and add $\_AbortF$ as a trigger to a new immediate transition from F to $\_A_F$. Remove the final flag from state F.

Fig. 5.7 shows an example where state $S4$ that is final and has an outgoing transition to $S3$ is transformed. Note that $S4$ can also be a final superstate (see Fig. 5.8) with internal behavior. The abort transformation (see also Sec. 5.6) must possibly run after this transformation.

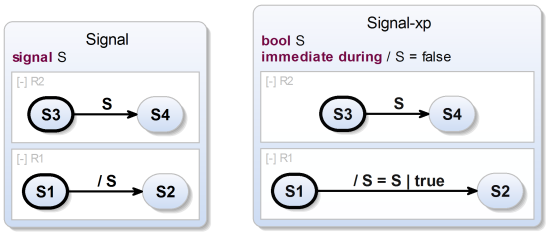
6 SyncCharts Features

6.1 Signals

Esterel and SyncCharts allow threads to communicate through *signals*. A signal is *present* throughout a tick if and only if it is *emitted* during that tick; otherwise, it is *absent*. Once a signal is emitted, it cannot be "unemitted" anymore for the rest of the tick. In SyncCharts, signals are tested for presence by just mentioning them in a transition trigger, and they are emitted by mentioning them in a transition action. In the Signal example in Fig. 6.1a, the signal S is emitted during the transition from S1 to S2, and this in turn triggers the transition from S3 to S4, within the same tick. We here discuss the case of *pure signals*, which do not carry a value; *valued signals* are presented in App. 6.2.

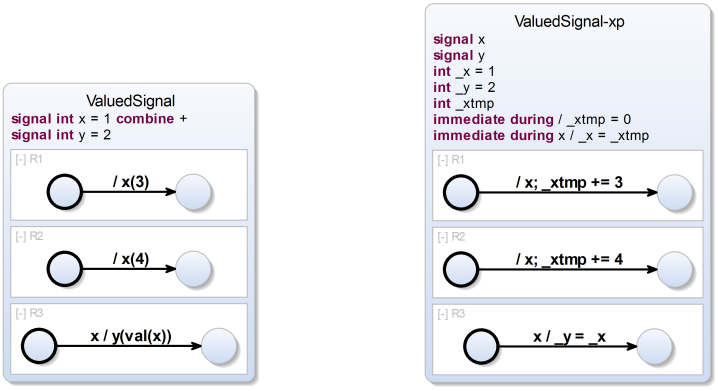
In SCCharts, signals can be emulated fairly easily with Booleans. As shown in Signal-xp (Fig. 6.1b), signal S is replaced by a Boolean variable of the same name; we consider signals and variables to use the same name space. If S is true, then it is considered present, thus the trigger of the transition from S3 to S4 does not change. S is initialized to false with a during action, which makes it absent per default. This during action is immediate, i. e., it also applies to the initial tick, even though in this example this would not be necessary, as the transition where S is tested is not immediate. However, making this during action immediate does not cause any harm, even in case it gets executed multiple times within a tick due to a self transition, as it always writes the same value. Furthermore, as S is declared as local variable, each entering of Signal-xp creates a fresh copy of S.

S is made present in the transition from S1 to S2 by performing a disjunction with true. As explained in Sec. 3.3, this disjunction constitutes a relative write, as opposed to the absolute write done in the initialization with false. The scheduling rules of sequential



(a) Original SCChart (b) After expansion

Figure 6.1: The transformation for Signal.



(a) Original SCChart (b) After expansion

Figure 6.2: The transformation for ValuedSignal.

constructiveness permit concurrent absolute and relative writes, and will schedule the absolute write before the relative write. Hence signals get initialized with false before they are possibly set to true, which faithfully models the behavior of signals.

This emulation of signals naturally handles the issue of *schizophrenia* [5], which arises when a signal scope is left and entered again within one clock tick. In our approach, as we explicitly initialize signals when their scope is (re-)entered, there is no issue of distinguishing the different signal instances.

6.2 Valued Signals

To augment the signal mechanism with data, Esterel and SyncCharts also provide *valued signals*. A valued signal is not only present or absent during a tick, as the pure signals presented in Sec. 6.1, but also carries a value of a certain type. A valued signal may also be associated with a *combination function*, which must be commutative and associative and allows the concurrent emission of multiple values.

In the ValuedSignal example in Fig. 6.2a, x and y are valued signals of type integer, initialized with 1 and 2, respectively. In the example, all transitions are delayed, hence all regions stay in their initial state, and x and y carry their initial values at the end of the initial tick.

When a valued signal gets emitted, it not only becomes present, but also gets assigned a certain value, given as an argument to the signal. E. g., region R1 in ValuedSignal emits x with value 3 in the second tick. The ? operator retrieves the value of a signal, hence y gets emitted with the value of x when x becomes present in the second tick.

In the example, x is combined with integer addition, and thus it is legal to emit it concurrently with values 3 and 4. If x would not have a combination function, this SCChart would have to be rejected. In ValuedSignal, at the end of the second tick both x and y carry the value 7.

The SC MoC naturally applies to valued signals, with or without combination functions. As illustrated in Fig. 6.2, the transformation of valued signals into an equivalent SCChart that only uses pure signals and plain variables is quite straightforward.

Consider first the case without combination function. A valued signal S of type T, possibly initialized to I, gets replaced by a variable $\_S$ of type T, possibly initialized to I, and a pure signal S. A signal emission "$S(V)$" gets replaced by "S; $\_S = V$," which first emits the pure signal S and then updates the associated value. The test for presence of S is not affected.

When a valued signal S of type T, possibly initialized to I, is associated with a combination function F, there must be a neutral element N for F, e. g., for $F = +$ it is $N = 0$. As for the case without combination function, we replace the valued signal S by a pure signal of the same name and a variable $\_S$ of type T. However, we need an additional variable $_S$tmp of type T that gets initialized to N at the beginning of each tick and serves to collect concurrently emitted values. We cannot simply use $\_S$ for that purpose because S must retain its value from the previous tick in case it does not get emitted in the current tick. The SC scheduling rules naturally order the initialization of $_S$tmp, which constitutes an absolute write in the SC MoC, before the signal emissions, which constitute relative writes, before the reads. The transformed version of ValuedSignal is presented in Fig. 6.2b.

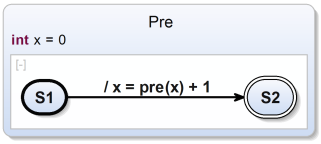
As a word of caution, combination functions assume that a valued signal has a unique value per tick and that within a tick, all signal emissions (relative writes) can be scheduled before any reads of their value. If that is not the case, i. e., if a thread reads the value of a valued signal S with a combination function and subsequently tries to emit S again, then a scheduling cycle results. This is a consequence of the concurrent during action that reads $_S$tmp, to collect the concurrent emissions, and writes to $\_S$, thus providing the value of S at the end of a tick.

6.3 Pre Operator

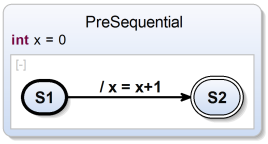
Esterel and SyncCharts provide the pre operator, which allows to access the presence status or the value of a signal in the previous tick. SCCharts provide a pre operator for variables, which can also be used for signals, introduced in Sec. 6.1. However, how to recover pre under sequential constructiveness may not seem obvious. To emulate pre(x) for some variable x, a naïve approach might be to introduce a fresh variable _pre_x, to store the value of x in _pre_x at the end of each tick in some new concurrent region _Pre, and to replace all occurrences of pre(x) by _pre_x. However, the SC scheduling rules would order the assignment to _pre_x after any assignment to x within the same tick, and thus _pre_x would effectively replicate x from the current tick, not from the previous tick.

What does work, however, is to store x in some fresh buffer variable _x at the end of a tick, and to copy this _x to _pre_x in the next tick. The SC scheduling rules will order the assignment to _x after all assignments to x, and will order the assignment to _pre_x before all references to _pre_x. This is illustrated in the Pre example in Fig. 6.3.

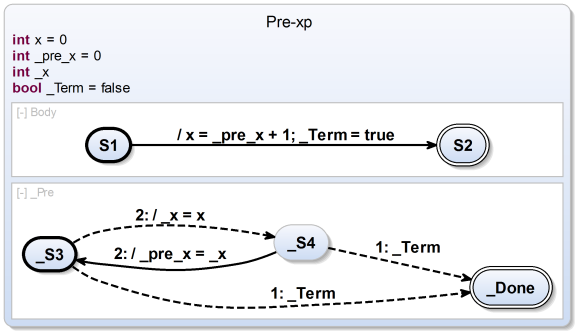
It is thus fairly straightforward to provide the pre operator. However, we also wish to



(a) Original SCChart



(b) Equivalent (optimized) SCChart, taking advantage of sequential constructiveness



(c) After expansion

Figure 6.3: The transformation for Pre.

point out that with the freedom of expressiveness brought about by the SC MoC, there should be less need to break up reactions and to refer to previous ticks than there is in the rather rigid MoC underlying SyncCharts. As explained in Sec. 3.3, SCCharts have no problem with modifying and reading the same variable multiple times within a tick, as long as these variable accesses are sequentially ordered. For example, in Fig. 6.3a, the transition action /x=pre(x)+1 might be replaced with simply /x = x+1, as in Fig. 6.3b. This does not need pre or any intermediate variable, but would be forbidden in the classical synchronous MoC.

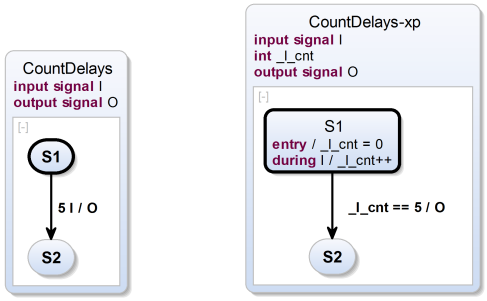
6.4 Count Delays

Count Delays allow to wait for the n-th occurrence of a trigger in order to take a transition or execute an action. Count Delays are elements of SyncCharts, and we here illustrate how this functionality can also be captured with a semantics-preserving M2M transformation that replaces count delays by an explicit counting variable.

For any state S that has outgoing transitions or actions with count delays, add a counter variable $Trig\_cnt$ for the trigger $Trig$ to its superstate M. Add an entry action $Trig\_cnt = 0$ to S in order to reset the counter when S is (re-)entered. Add a during action $Trig\_cnt{+}{+}$ that counts the occurrences of the trigger. Finally replace the count delay trigger by an appropriate equal expression referring to $Trig\_cnt$ and comparing it to the the number of the count delay definition. Note that a state with several count

delays that refer to the same trigger can share a counter.

Fig. 6.4 shows an example where state $S1$ has an outgoing transition with a signal I as the trigger and a count delay of 5. This means that when I is present for the 5th tick, then the transition will be taken.



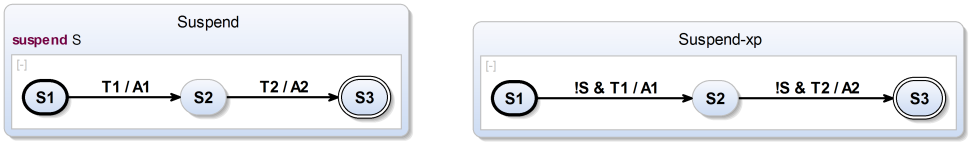
(a) Original SCChart (b) After expansion

Figure 6.4: The transformation for CountDelays.

6.5 Suspension

Suspensions allow to "freeze" behavior upon some trigger. In our experience, this operator is rarely used in practice. However, it is an element of SyncCharts, and we here illustrate how this functionality can also be captured with a semantics-preserving M2M transformation that replaces suspension by explicit conditional control.

In a superstate M that is suspended by some trigger S, for each internal transition of M, the transition trigger gets conjuncted with the negation of S, as illustrated in Fig. 6.5. This is applied recursively to all superstates of M. This procedure ensures that no transitions are taken and no transition actions get executed when M is suspended.



(a) Original SCChart (b) After expansion

Figure 6.5: The transformation for Suspend.

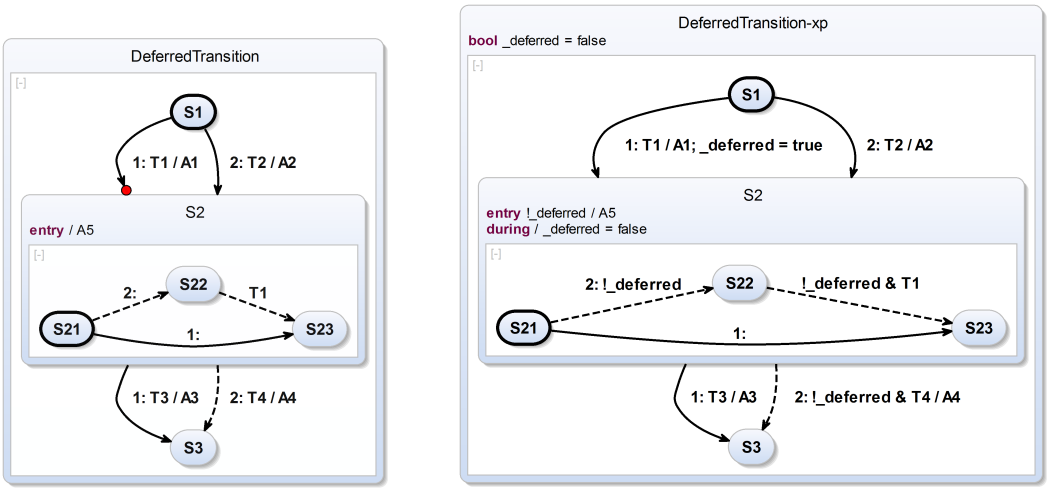
An interesting aspect here is how suspension is combined with aborts. Suspension does not prevent aborts [2]. However, suspension suppresses internal reactions. As it turns out, this case is handled naturally by our transformation rules. Here one must first expand suspension and then aborts, as the checking for aborts should still be done even when the state is suspended.

7 Further Features from SCADE/QUARTZ/Esterel v7

7.1 Deferred Transitions

Deferred transitions can help in breaking dependency cycles because they introduce a tick boundary when transitioning to the target state. This means when taking such a transition then the target state will be still entered in the current tick but all possibly immediate behavior will be preempted until the following tick. Note that possible entry actions will hence be suppressed because they belong to the immediate behavior that would occur when a state is entered by an ordinary transition. Deferred transition are not that common in synchronous languages and not a feature of SyncCharts. Though, they are part of SCADE and the default behavior of Ptolemy ModalModel transitions.

We here illustrate how this functionality can also be captured with a semantics-preserving M2M transformation that replaces deferred transitions by an explicit additional state.



(a) Original SCChart

(b) After expansion

Figure 7.1: The transformation for **Deferred** transitions.

For any state S with incoming deferred transitions create a *_deferred* flag in the outer scope initially set to false. For every incoming deferred transition T add an assignment $\_deferred = true$. In the state S create a during action that reset the *_deferred* flag

to false. For every immediate transition IT inside S (including immediate outgoing transitions of S) add !_*deferred* as a conjunction to the trigger of IT. Fig. 7.1 shows an example where a deferred transition from state $S1$ to state $S2$ is transformed.

```
1  def void transformDeferred(State state) {
2      val incomingDeferredTransitions = state.incomingTransitions. filter [deferred];
3
4      // If there are any such transitions
5      if (!incomingDeferredTransitions.nullOrEmpty) {
6
7          // Add a new deferVariable to the outer state, set it initially to FALSE and
8          // add a during action in the state to reset it to FALSE
9          val deferVariable = state.parentRegion.parentState.createBoolVariable(state.id("_deferred"))
10         deferVariable.setInitialValue(FALSE)
11         val resetDeferSignalAction = state.createDuringAction
12         resetDeferSignalAction.addEffect(deferVariable.assign(FALSE))
13
14         // For all incoming deferred transitions, reset the defer flag and add an assignment
15         // setting the deferVariable to true when entering the state
16         for ( transition : incomingDeferredTransitions) {
17             transition .setDeferred(false)
18             transition .addEffect(deferVariable.assign(TRUE))
19         }
20
21         // Prevent any immediate internal behavior of the state and any immediate outgoing
22         // transition in case deferVariable is set to TRUE, i.e., the state was entered
23         // by a deferred transition
24         val allInternalImmediateActions = state.allContainedActions
25                                             . filter (e | e.immediate || e instanceof EntryAction).toList
26         for (action : allInternalImmediateActions) {
27             val deferTest = not(deferVariable.reference)
28             if (action.trigger != null) {
29                 action.setTrigger(deferTest.and(action.trigger))
30             } else {
31                 action.setTrigger(deferTest)
32             }
33         }
34     }
35 }
```

Figure 7.2: Xtend implementation of transforming deferred transitions.

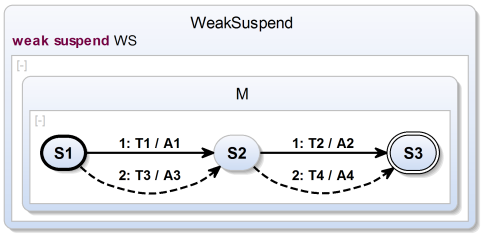
7.1.1 Implementation

We implemented all transformations from Extended SCCharts to Core SCCharts as M2M transformations with Xtend.[1] To illustrate the compact, modular nature of the M2M transformations, Fig. 7.2 shows the "deferred" transformation described above. Xtend keywords and Xtend extension functions are highlighted.

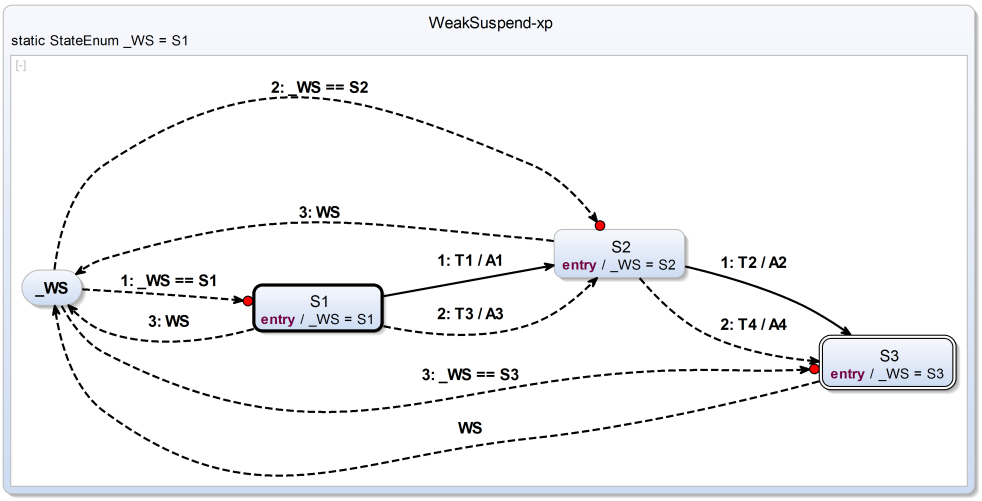
The precondition is checked in line 5. Line 9 adds the new $\_deferred$ flag variable and line 10 sets its initialization to $false$. The during action which resets this flag in each tick is created in lines 11-12. Lines 16-19 modify all deferred transitions to be normal transitions now and add an assignment effect to each transition which sets the $\_deferred$ flag to $true$. The latter indicates that we entered the state by a transition that was a

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

deferred transition prior to this transformation. Finally, lines 24-33 collect all internal transitions which are immediate and all entry and during actions, adding a test for the negated $\_deferred$ flag to their triggers. This is done in order to prevent taking these internal immediate transitions if we entered the outer state by a deferred transition.



(a) Original SCChart



(b) After expansion

Figure 7.3: The transformation for Weak Suspend.

7.2 Weak Suspension

There is also a weak form of suspend that also freezes the behavior but will allow to execute a "last whish", i.e., any instantaneous actions that would be executed in the tick but it will not change state.

In a superstate M that is weak suspended by some trigger WS, we need a variable $\_WS$ to remember the internal state S we have been into. This is done by having an entry action for each state S setting this variable to S. This is done only in cases when there is not yet a WS signal. There is an additional state $\_WS$ for M. When entering $\_WS$ that means that the "last wish" is complete. After this we immediately return to

the original state S where we were in before performing the "last wish", this depends on the $\_WS$ variable.

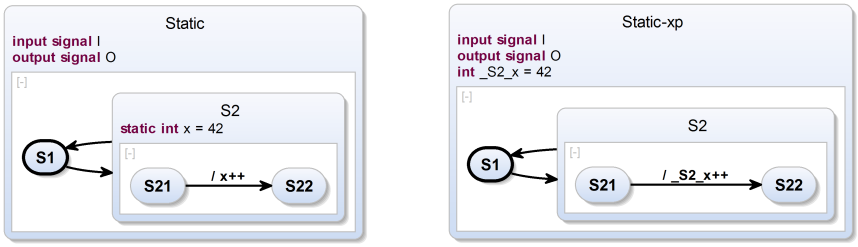
For each internal state of M, add an immediate transition with the lowest priority that if the WS signal is there (and no other transition is triggered) will go directly to the new $\_WS$ state.

To prevent immediate cycles the immediate transitions from $\_WS$ to any state S inside M are deferred meaning that when taking these transitions S is entered but no further immediate transitions or entry actions are taken, neither outgoing from S nor inside S. This is illustrated in Fig. 7.3.

This is applied recursively to all superstates of M. This procedure ensures that no transitions are taken and no transition actions get executed when M is suspended.

7.3 Static Variables

It is possible to persist variables across exiting and re-entering their scope, with the static modifier. This is similar to "internal static variables" — as opposed to "automatic variables" — in C. We here illustrate how this functionality can also be captured with a semantics-preserving M2M transformation that replaces static (local) variables by global variable.



(a) Original SCChart (b) After expansion

Figure 7.4: The transformation for Static.

In some local superstate M that has a declaration of a static variable x, move the declaration of the variable to the root state of the SCChart and rename x respecting a proper unique and qualified naming. Within the scope of x (within M) update all references (accesses) to x to the new name. Remove the static keyword from the declaration. This is applied for all superstates that contain static variable declarations. Fig. 7.4 shows an example for the superstate $S2$ that declares a local static variable x.

7.4 History Transitions

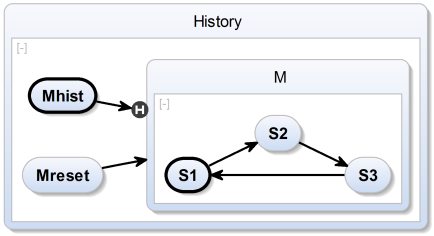
Usually, the regions of a superstate S are entered through their initial states. When S is left and re-entered, the regions are again re-entered through their initial states; the

states that the regions may have been in when S was active the last time are "forgotten." This is a strong invariant that supports modularity. However, history transitions allow to preserve these states across invocations of S.

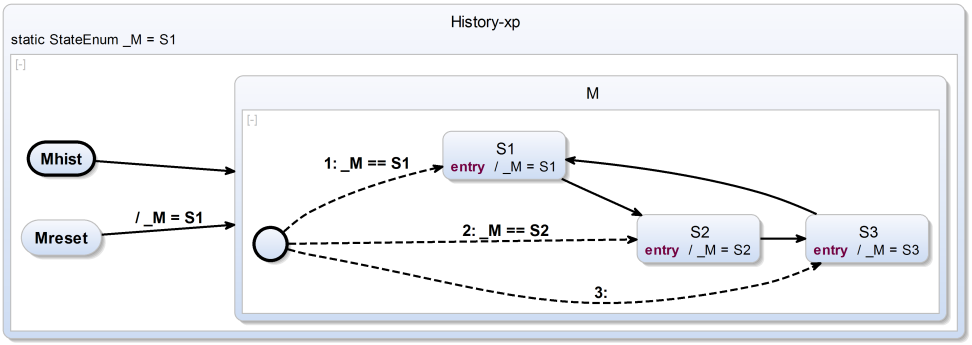
History transitions have been introduced by Harel in his original Statechart proposal [17] and are included with several other Statechart dialects. However, history transitions should be used with care, as they increase the overall state space of the model; now one has to remember not only the sub-states of active superstates, but those of inactive superstates as well. This is something that becomes clear in the transformation presented here as well.

History transitions allow to re-enter a superstate at the state it was left previously, in case it has been entered before; otherwise, it is entered through the usual default state. There exist two variants, *shallow history* (indicated with a "H"), which only memorizes state information at the top level, and *deep history* (indicated with a "H*"), where state information is recorded recursively at lower states as well.

Fig. 7.5 illustrates how to transform history transitions. A fresh variable _M is introduced to record which state of M should be entered in case of a history transition; the type StateEnum indicates that _M should store a state, e. g., as an enumeration type. _M does get initialized, to S1, but it is also static, meaning that it preserves its value even after its scope (the enclosing superstate) is left and re-entered again. Thus _M gets initialized only once, when the superstate is entered for the first time. In case of instantaneous transitions, it might be the case that _M gets assigned multiple values within one tick, which is allowed by sequential constructiveness.

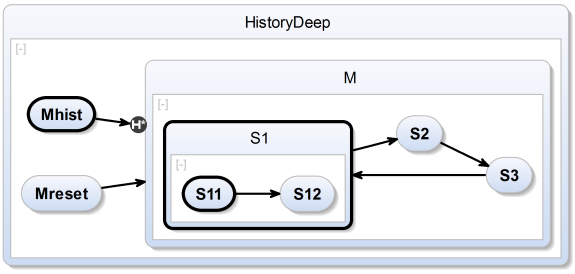


(a) Original SCChart

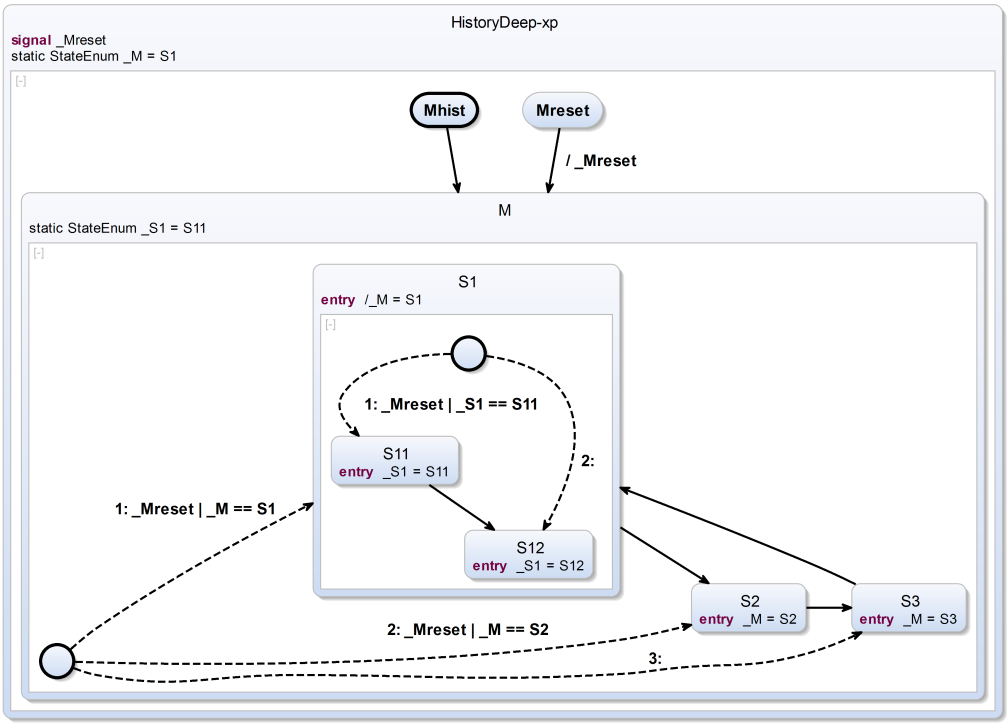


(b) After expansion

Figure 7.5: The transformation for History.



(a) Original SCChart



(b) After expansion

Figure 7.6: The transformation for HistoryDeep.

History-xp implements shallow history, where state information is only recorded at the top-level of the entered state. Adding an explicit signal _Mhist, which is emitted in case M is entered through a history transition, allows to implement deep history. This is illustrated in HistoryDeep-xp, see Fig. 7.6. To complete deep history, the lower states must also maintain state variables, analogously to _M, and must re-enter depending on _Mhist.

Conceptually, history transitions do for local states what the static modifier does for data (see Sec. 7.3). It would be conceivable to make history transitions (preservation of state) imply static data (preservation of data). However, in this language proposal, these are handled independently from each other; i. e., history transitions do not automatically make all data static, and conversely making data static does not imply that incoming transitions are history transitions.

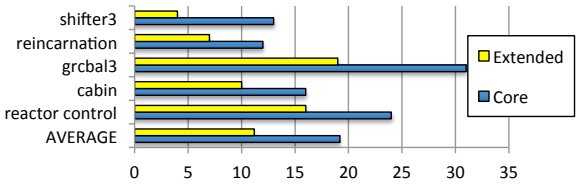
8 Validation and Experimental Results

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 an SCCharts editor and compiler in an Eclipse-based modeling environment.

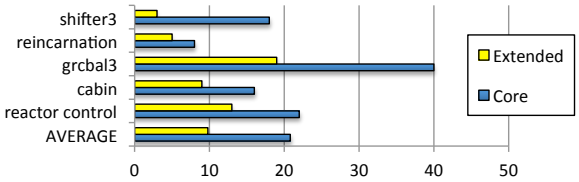
First of all, there is the question of correctness. As stated in Sec. 2, SCCharts are a conservative extension of SyncCharts. Thus one should be able to verify that valid SyncCharts are also acceptable (schedulable) as SCCharts, and that they behave the same. To that end, we have collected > 100 validation benchmarks with associated input and output traces during the course of developing SCCharts and the transformations presented here, and we have validated that the SCCharts compiler does produce the same result as both another SyncChart-to-Synchronous C compiler [31] and, where available, Esterel Studio.

Another question to ask is how much Extended SCChart models increase when transforming them to Core SCCharts. A size explosion could indicate that the Core SCChart operations do not capture the essence of the MoC, or that the transformations proposed here lead to unnecessary model blow-up. If, on the other hand, the model size would not change in any way, one might ask what benefit the extensions bring to the modeler. In Fig. 8.1 we compare the number of nodes and transitions for some benchmarks suggested by Traulsen et al. [31]. On average, the Extended SCCharts model has 42% fewer states and 53% fewer transitions than the equivalent Core SCCharts model. This suggests that on the one hand the extended constructs help to make the models more compact, but that on the other hand the code transformations do not lead to a model explosion.

It is also interesting to see how a transformation-based synthesis approach, which eliminates aborts etc. at the model level before downstream compilation, compares with a "native" implementation that implements these advanced constructs at a lower level. This comparison is a bit similar to a RISC vs. CISC comparison. Extended SCCharts ("CISC") allow a more compact representation than Core SCCharts ("RISC"); after all, this is the motivation for introducing the extended constructs in the first place. However, expanding the models into Core SCCharts allows a more light-weight synthesis downstream. In Fig. 8.2, we compare both synthesis paths when compiling for an Intel Core 2 Duo P8700 (2.53GHz) architecture. As can be seen there, even though the expansion to Core SCCharts resulted in significantly larger models, the resulting executables are fairly comparable, with the expanded models resulting on average in 19% slower but 2% smaller executables. As both synthesis paths are equally possible for SCCharts, one might chose the one that gives better performance for a specific model. One might also use both paths during development and cross-check the results, which is another aspect that is interesting for safety-critical applications.

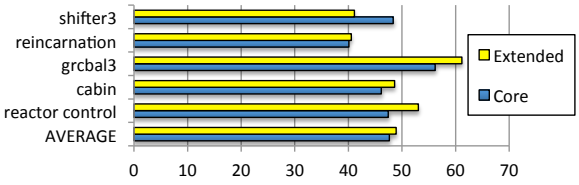


(a) Number of states

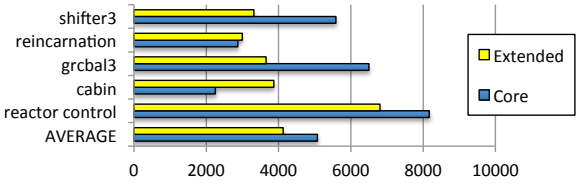


(b) Number of transitions

Figure 8.1: Comparison of Extended SCCharts with equivalent Core SCCharts resulting from transformations.



(a) Executable size [KBytes]



(b) Clock cycles per tick

Figure 8.2: Comparison of code synthesis of Extended SCCharts directly to Synchronous C with synthesis to SCL via transformations to Core SCCharts.

9 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 macro states can be left with a join-like termination of their sub-states.

The Core SCCharts operations are sufficient to allow compact encodings of complex behavior, in particular they allow to avoid the state explosion of flat automata. Based on these core operations, we can 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 expansion rules presented here not only can serve to define and explain the extended SCCharts constructs, but also turned out to be a viable option for code synthesis, a result that was not obvious from the onset.

To conclude, the flexible yet deterministic semantics of SCCharts makes them particularly suitable for safety-critical applications. This is augmented by a direct synthesis path to an efficient, imperative-style sequential program that preserves the structure of the original SCChart.

Bibliography

[1] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. *Workshop on Reconciling Performance with Predictability (RePP'09), Embedded Systems Week*, Grenoble, France, Oct. 2009.

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

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

[4] 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.

[5] 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.

[6] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In K. Flautner and J. Regehr, editors, *Proceedings of the ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM.

[7] F. Boussinot. SugarCubes implementation of causality. Research Report RR-3487, INRIA, Sept. 1998.

[8] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, New York, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] W.-P. de Roever, G. Lüttgen, and M. Mendler. What is in a step: New perspectives on a classical question. In Z. Manna and D. A. Peled, editors, *Time for Verification*, pages 370–399. Springer LNCS 6200, 2010.

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

[14] Esterel Technologies. *The Esterel v7 Reference Manual Version v7_30—initial IEEE standardization proposal.* Esterel Technologies, 679 av. Dr. J. Lefebvre, 06270 Villeneuve-Loubet, France, Nov. 2005. `http://www.esterel-technologies.com/files/Esterel-Language-v7-Ref-Man.pdf`.

[15] P. L. Guernic, T. Goutier, M. L. Borgne, and C. L. Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sept. 1991.

[16] 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.

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

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

[19] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Upper Saddle River, NJ, 1985.

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

[21] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

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

[23] M. Mendler, T. R. Shiple, and G. Berry. Constructive boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, 40(3):283–329, 2012.

[24] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Proc. Int. Conf. on Theoretical Aspects of Computer Software (TACS'91)*, pages 244–264, London, UK, 1991. Springer.

[25] 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.

[26] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation. *Design Autom. for Emb. Sys.*, 14(3):165–192, 2010.

[27] K. Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2010. `http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf`.

[28] K. Schneider, J. Brandt, T. Schüle, and T. Türk. Improving constructiveness in code generators. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *Int'l Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, pages 1–19, Edinburgh, Scotland, UK, apr 2005. ENTCS.

[29] T. R. Shiple, G. Berry, and H. Touati. Constructive Analysis of Cyclic Circuits. In *Proc. European Design and Test Conference (ED&TC'96), Paris, France*, pages 328–333, Los Alamitos, California, USA, Mar. 1996. IEEE Computer Society Press.

[30] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (EMSOFT'06)*, pages 142–151, Seoul, South Korea, Oct. 2006. ACM.

[31] 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.

[32] 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.

[33] 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.

[34] 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.