INSTITUT FÜR INFORMATIK

Sequentially Constructive Concurrency
A Conservative Extension of the Synchronous Model of Computation

Reinhard von Hanxleden, Michael Mendler,
Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann,
Christian Motika, Stephen Mercer, Owen O'Brien
and Partha Roop

Bericht Nr. 1308
August 2013
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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

Sequentially Constructive Concurrency
A Conservative Extension of the Synchronous Model of Computation

Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado,
Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen
Mercer, Owen O'Brien and Partha Roop

Bericht Nr. 1308
August 2013
ISSN 2192-6247

R. von Hanxleden, B. Duderstadt, I. Fuhrmann, and C. Motika are with the Department of Computer Science, Kiel University, Kiel, Germany.
E-mail: {rvh, bdu, ima, cmot}@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
P. Roop is with Auckland University, Auckland, New Zealand.
E-mail: p.roop@auckland.ac.nz

Parts of this report have appeared in the *Proceedings of the Design, Automation and Test in Europe Conference (DATE'13)*, Grenoble, France, March 2013

Abstract

Synchronous languages ensure deterministic concurrency, but at the price of heavy restrictions on what programs are considered valid, or *constructive*. Meanwhile, sequential languages such as C and Java offer an intuitive, familiar programming paradigm but provide no guarantees with regard to deterministic concurrency. The *sequentially constructive* model of computation (SC MoC) presented here harnesses the synchronous execution model to achieve deterministic concurrency while addressing concerns that synchronous languages are unnecessarily restrictive and difficult to adopt.

In essence, the SC MoC extends the classical synchronous MoC by allowing variables to be read and written in any order as long as sequentiality expressed in the program provides sufficient scheduling information to rule out race conditions. This allows to use programming patterns familiar from sequential programming, such as testing and later setting the value of a variable, which are forbidden in the standard synchronous MoC. The SC MoC is a conservative extension in that programs considered constructive in the common synchronous MoC are also SC and retain the same semantics. In this paper, we identify classes of variable accesses, define sequential constructiveness based on the concept of SC-admissible scheduling, and present a priority-based scheduling algorithm for analyzing and compiling SC programs.

Keywords: Concurrency, synchronous languages, determinism, constructiveness.

Contents

List of Figures

List of definitions, propositions and theorems

1 Introduction

One of the challenges of embedded system design is the deterministic handling of concurrency. The concurrent programming paradigm exemplified by languages such as Java and C with Posix threads essentially adds unordered concurrent threads to a fundamentally sequential model of computation. Combined with a shared address space, concurrent threads may generate write/write and write/read *race conditions*, which are problematic with regard to ensuring deterministic behavior [1], as the run-time order of execution of a multi-threaded program depends both on the actual time that each computation takes to execute and on the behavior of an external scheduler beyond the programmer's control .

As an alternative to this non-deterministic approach, the *synchronous* model of computation (MoC), exemplified by languages such as Esterel, Lustre, Signal and SyncCharts [2], approaches the matter from the concurrency side. The synchronous MoC divides time into discrete *macro ticks*, or *ticks* for short. Within each tick, a synchronous program reads in inputs and calculates outputs. The inputs to a synchronous program are assumed to be in synchrony with their outputs, and the time that computations take is abstracted away. Simultaneous threads still share variables, where we use the term "variable" in a generic sense that also encompasses streams and signals. However, race conditions are resolved by a deterministic, statically-determined scheduling regime, which ensures that within a tick, a) reads occur after writes and b) each variable is written only once. A program that cannot be scheduled according to these rules is rejected at compile time as being *not causal*, or *not constructive*. This approach ensures that within each tick, all variables can be assigned a unique value. This provides a sufficient condition for a deterministic semantics, though, as we argue here, not a necessary condition.

Introducing global synchronization barriers and sequences of reaction cycles is a sound basis for deterministic concurrency and applicable also for general programming languages commonly used for embedded systems. Demanding unique variable values per tick not only limits expressiveness but also runs against the intuition of programmers versed in sequential programming, and makes the task of producing a program free of "causality errors" more difficult than it needs to be. For example, a simple programming pattern such as `if (!done) { ...; done = true}` cannot be expressed in a synchronous tick because `done` must first be written to within the cycle before it can be read. Another issue that prevents this program from compiling under a synchronous interpretation is that `done` would possibly be both `false` and `true` within the same tick. However, in this example, there is no race condition, nor even any concurrency that calls for a scheduler in the first place. Thus, there is no reason to reject such a program in the interest of ensuring deterministic concurrency.

Contributions. We propose the *sequentially constructive* model of computation (SC MoC), a conservative extension to the synchronous MoC that accepts a strictly larger class of programs. Specifically, the SC MoC permits variables to have multiple values per tick as long as these values are either explicitly ordered by sequential statements within the source code or the compiler can statically determine that the final value at the end of the tick is insensitive to the order of operations. This extension still ensures deterministic concurrency, and is conservative in the sense that programs that are accepted under the existing synchronous MoC have the same meaning under the SC MoC. For example, all constructive Esterel programs are also sequentially constructive (SC). However, there exist Esterel programs that are SC, but not constructive in Berry's sense [3]. E. g., all programs that do not use the concurrency operator are SC, though they may be non-constructive in Esterel.

Outline. The next section discusses related work. Sec. 3 presents the SC language (SCL) and the SC graph (SCG), which we use as a basis for the concept of sequential constructiveness.

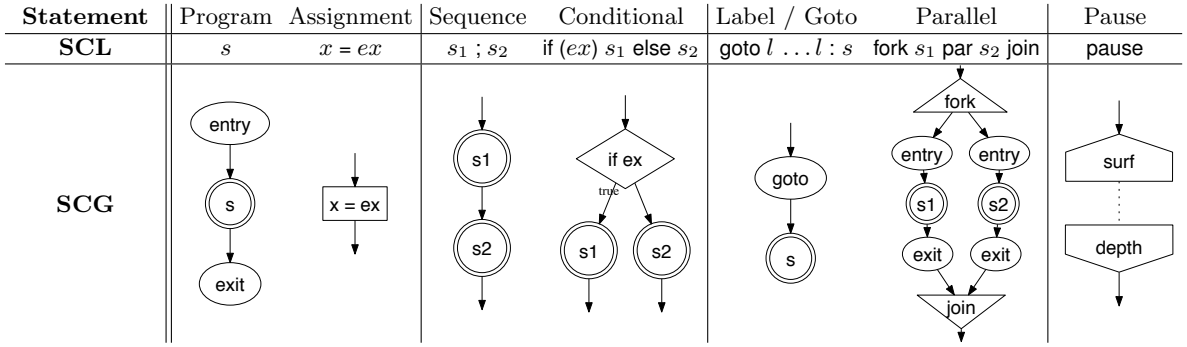


Figure 1: The mapping between SCL statements and SCG subgraphs. Double circles are place holders for SCG subgraphs. Solid arrows depict *seq* (sequential) edges, the dotted line indicates a *tick* edge.

Sec. 4 presents the general scheduling problem, on which Sec. 5 builds to define sequential constructiveness. Sec. 6 presents an approach to analyze whether programs are SC and to compute a schedule for them. Sec. 7 discusses how Esterel/SyncChart-style signals can be emulated with plain shared variables under the SC MoC. Sec. 8 presents SCCharts, a dialect of Statecharts [4] that is inspired by SyncCharts [5] but harnesses the SC MoC to provide a deterministic semantics to a larger class of Statecharts. Sec. 9 relates SC to other MoCs, such as Pnueli and Shalev [6]. We summarize in Sec. 10.

2 Related Work

Edwards [7] and Potop-Butucaru et al. [8] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages and classical work such as Ferrante et al.'s Program Dependence Graph (PDG) [9]. The SC Graph introduced here can be viewed as a traditional control flow graph enriched with data dependence information akin to the PDG for analysis and scheduling purposes.

Esterel [10, 11] 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* [12] under the non-inertial delay model [13], which can be fully decided using ternary Kleene algebra [14, 13]. The algebraic transformations proposed by Schneider et al. [15] 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 irrespective 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 $x_0 + ex_1 + ex_2$, irrespective 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 [17], 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 [18]. 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 [19] 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. [20] have extended Lustre 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 [21], augments C with synchronous, deterministic concurrency and preemption. It provides a coroutine-like thread scheduling mechanism, with thread priorities that have to be explicitly set by the programmer. The algorithm presented in Sec. 6.2 can be used to automatically synthesize priorities for Synchronous C. PRET-C [22] also provides deterministic reactive control flow, with static thread priorities.

SHIM [23] provides concurrent Kahn process networks with CSP-like rendezvous communication [24] 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.

The concept of sequential constructiveness not only applies to textual C/Java-like languages, but also to a graphical formalism such as Statecharts [4]. In fact, the development of a semantically sound, yet flexible and intuitive Statechart dialect was the original motivation for developing the SC MoC. We have developed such a Statechart dialect, named Sequentially Constructive Statecharts (SCCharts) [25], to be used for the development of safety-critical embedded systems in an industrial setting.

The core semantic concepts of SCCharts are analogous to André's SyncCharts [5], which can be viewed as a graphical variant of the synchronous language Esterel [10, 11]. In Esterel Studio, SyncCharts were introduced as Safe State Machines. The *Safety Critical Application Development Environment* (SCADE) uses a variant of SyncCharts elements to augment dataflow diagrams with reactive behavior, by extending boolean Clocks towards clocks that express state [26, 27]. The main differences between SCCharts and SyncCharts (including those present in SCADE) are:

1. SCCharts are not restricted to constructiveness in Berry's sense [3], but relax this require-

ment to sequential constructiveness. This makes a significantly larger class of Statecharts acceptable without compromising determinism.

2. SCCharts do not introduce signals, but use shared variables. However signals can be fully emulated with variables under SC scheduling. I. e., one can implement signal initialization to "absent" as absolute write with *false*, potentially followed in the same tick by a signal emission implemented as relative write that performs disjunction with *true*.

An interesting question is how SCCharts and the SC MoC domain relate to other Statecharts dialects with respect to what class of programs are considered admissible. This opens up a further line of investigations that we plan to persue in the future. Briefly, SC-scheduling is not the only way to interpret the Synchrony Hypothesis, i. e., to execute concurrent threads in a clock-synchronous fashion. What is considered constructively executable depends a lot on the target execution architecture. The more scheduling choices there remain admissible in the target, and thus out of control of the compiler, the more restrictive the compiler has to be in admitting programs, in order to guarantee a deterministic and bounded macro-tick response. If the compiler is permitted to resolve scheduling choices, by sequentializing statements in certain coherent ways, e.g., as we do here, then fewer schedules are admissible. Thus, more programs can execute constructively.

Various other approaches with their own admissible scheduling schemes have been considered for Statecharts. Some are more restricted, some more generous and yet others incomparable with SC admissibility and sequential constructiveness (S-constructiveness). The three most prominent approaches are due to Pnueli and Shalev [6], Boussinot [28] and Berry and Shiple [29], which we refer to as P, L, and B-constructiveness, respectively. We find that B-constructiveness is most restrictive and strictly included in all the notions of {S, P, L}-constructiveness, while the latter are incomparable with each other. We believe that S-constructive programs are more practical than either P- and L-constructive programs. This gains substantial extra ground for programming synchronous interactions compared to the existing imperative synchronous code which is based on B-constructiveness. None of them considers sequential control flow as SC does.

3 The SC Language and the SC Graph

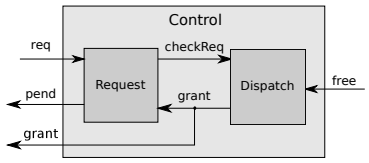
To illustrate the SC MoC, we introduce a minimal *SC Language* (SCL), adopted from C/Java and Esterel. The concurrent and sequential control flow of an SCL program is given by an *SC Graph* (SCG), which acts as an internal representation for elaboration, analysis and code generation. Fig. 1 presents an overview of the SCL and SCG elements and the mapping between them.

3.1 The SC Language

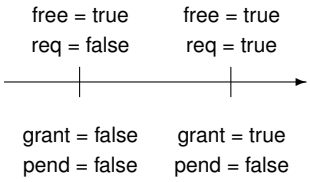
SCL program constructs have the following *abstract syntax* of *statements*

$$s \quad ::= \quad x = ex \mid s \,;\, s \mid \textsf{if}\ (ex)\ s\ \textsf{else}\ s \mid l : s \mid \textsf{goto}\ l \mid \textsf{fork}\ s\ \textsf{par}\ s\ \textsf{join} \mid \textsf{pause}$$

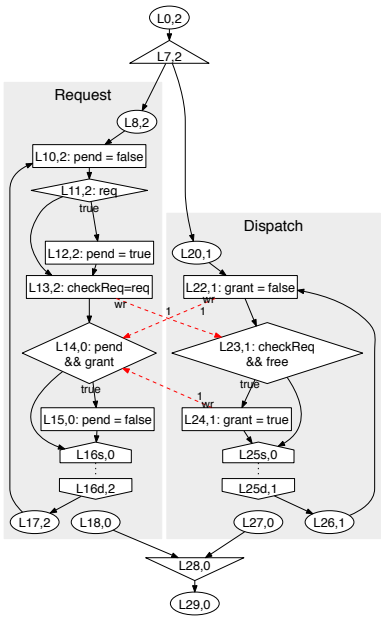
where x is a *variable*, ex is an *expression* and $l \in L$ is a *program label*. The statements s comprise the standard operations assignment, the sequence operator, conditional statements, labelled commands and jumps. As a syntactical detail on the conditional, this, as is the practice in C-like languages, does not use a then keyword, but we will still refer to the two branches as then and else branches. In addition, the two statements introducing synchrony and concurrency



(a) The dataflow view



(b) The first two ticks of an example trace, shown as tick time line; inputs are above the time line, outputs below



(c) The SC Graph (SCG), indicating sequential flow (continuous arrows), data dependencies (dashed, red arrows), and the tick delimiter edges (dotted lines). The data dependency edges are labeled with their type (here wr only) and their weight (1); other edges have weight 0. Nodes are labeled with node identifiers, which here correspond to line numbers of the SCL program, and priorities as computed by the algorithm presented in Sec. 6.2.

```
1  module Control
2  input bool free, req;
3  output bool grant, pend;
4  {
5    bool checkReq;
6
7    fork {
8      // Thread "Request"
9      Request_entry:
10     pend = false;
11     if (req)
12       pend = true;
13     checkReq = req;
14     if (pend && grant)
15       pend = false;
16     pause;
17     goto Request_entry;
18   }
19   par {
20     // Thread "Dispatch"
21     Dispatch_entry:
22     grant = false;
23     if (checkReq && free)
24       grant = true;
25     pause;
26     goto Dispatch_entry;
27   }
28   join;
29 }
```

(d) The SCL program

Macro tick	a	1											1	2												2
Micro tick	i	1	2	3	4	5	6	7	8	9	10	11	12	12	1	2	3	4	5	6	7	8	9	10	11	12	13	13
Input vars	free	***t***												*t*	***t***													*t*
	req	***f***												*f*	***t***													*t*
Output vars	grant	⊥								*f*				***f***	*f*							*f*		*t*				***t***
	pend	⊥				*f*								***f***	*f*		*f*		*t*								*f*	***f***
Local var	checkReq	⊥						*f*						*f*	*f*					*t*								*t*
Continuations	C_{Root}	L0	L7	[L28]										[L28]	[L28]													[L28]
	C_{Request}	⊥		L8	L10	L11	L13	L14	L14	L14	L14	L14	L16s	(L16s)	L16d	L10	L11	L12	L13	L14	L14	L14	L14	L14	L14	L15	L16s	(L16s)
	C_{Dispatch}	⊥		L20	L20	L20	L20	L20	L22	L23	L25s	(L25s)	(L25s)	(L25s)	L25d	L25d	L25d	L25d	L25d	L25d	L22	L23	L24	L25s	(L25s)	(L25s)	(L25s)	(L25s)
Scheduled nodes	R_i^a	L0	L7	L8	L10	L11	L13	L20	L22	L23	L25s	L14	L16s		L16d	L10	L11	L12	L13	L25d	L22	L23	L24	L25s	L14	L15	L16s	

(e) An admissible sequence of macro ticks. The values *true* and *false* are abbreviated as t and f. At tick granularity, this run corresponds to the example trace. We see for each micro tick the current variable values, where ⊥ denotes "uninitialized". The input values provided by the environment and the output values visible to the environment are shown in **bold**. To avoid cluttering the table, values that do not change from one micro tick to the next are omitted, except at the end of a macro tick. Continuations denote for each thread the statement to be executed next, as further explained in Sec. 4.1. Threads may be disabled (denoted ⊥), active, waiting (square brackets), or pausing (parentheses).

Figure 2: The Control example.

are the pause statement, which deactivates a thread until the next tick commences, and parallel composition, which forks off two threads and terminates (joins) when both threads have terminated. In Esterel, parallel composition is denoted ‖; we here use fork/par/join instead, to provide additional structure and to avoid confusion with the *logical or* used in expressions. For simplicity, we here only consider parallelism of degree two; larger numbers of concurrent threads can be accommodated by nesting of parallel compositions, or by a straightforward extension of syntax and semantics to support arbitrary numbers of concurrent threads directly.

A *well-formed* SCL program is one in which (i) expressions and variable assignments are type correct, (ii) there are no duplicate or missing program labels and (iii) no goto jumps into or out of a parallel composition.

To present SCL examples in *concrete textual* as opposed to abstract syntax, more syntactic information is needed. E.g., we typically add braces for structuring the code, subject to conventions regarding the binding strength of the operators (the conditional binds weaker than the sequence). We may also omit empty else branches, or enhance the unstructured goto with structured loops (for, while, etc.). Also, there may be comments and local variable declarations, including their data types, initial values and input/output assignations. However, as our formal development will be based on the internal representation of SCL programs as SC Graphs, we may leave the concrete SCL syntax informal. An illustration of the concrete SCL syntax is the Control example program shown in Fig. 2d and elaborated below.

SCL is a concurrent imperative language with shared variable communication. Variables can be both *written* to and *read* from by concurrent threads. Reads and writes are collectively referred to as variable *accesses*. The sublanguage of *expressions* ex used in assignments and conditionals is not restricted. All we assume is a function *eval* to evaluate ex in a given memory ρ and return a value $v = eval(e, \rho)$ of the appropriate data type. However, we rule out side effects when evaluating ex.

Esterel *signals* can be coded in SCL using variable accesses as described in Sec. 7. We are also omitting Esterel's abortions and traps, which can be emulated with variables, too; see the ABSWO example discussed in Sec. 8.

3.2 The Control Example

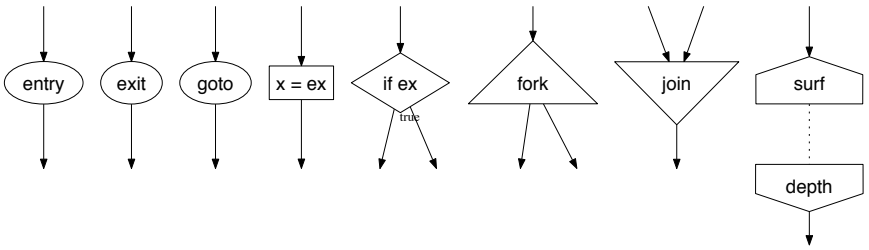
Turning our attention again to the Control example from Fig. 2, we see that this program executes two concurrent *threads*, named Request, and Dispatch. Together with the Root thread, which always runs at the top level, this program thus consists of three threads. Strictly speaking, we must distinguish threads and run-time *thread instances*, since in general one (static) thread can be instantiated multiple times, even within one tick. But for now, this distinction is not important, as in Control there is exactly one run-time instance of each thread.

The functionality of Control is inspired by Programmable Logic Controller software used in the railway domain. It processes requests (as indicated by the input flag req) to a resource, which may be free or not. As indicated in the dataflow/actor view in Fig. 2a, there are two separate functional units, corresponding to the Request and Dispatch threads, which process the requests and dispatch the resource. The output variables indicate whether the resource has been granted or is still pending.

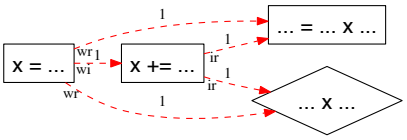
The execution of Control is broken into discrete reactions, the aforementioned (macro) ticks. During each tick, the following sequence is performed:

1. read input variables from the environment,
2. execute all active (currently instantiated) threads until they either terminate or reach a pause statement,
3. write output variables to the environment.

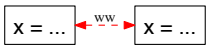
Only the output values emitted at the end of each macro tick are visible to the outside world. The internal progression of variable values within a tick, i. e., while performing a sequence of *micro ticks* (cf. Sec. 3.9), is not externally observable. Hence, when reasoning about deterministic behavior, we only consider the outputs emitted at the end of each macro tick.



(a) Node types, with control flow and tick boundary edges (weight 0)



(b) Data dependency edges (weight 1)



(c) Write-Write conflict

Figure 3: The syntactical elements of the SC Graph (SCG)

The execution of Control begins with a fork that spawns off Request and Dispatch. These two threads then progress on their own. Were they Java threads, a scheduler of some run time system could now switch back and forth between them arbitrarily, until both of them had finished. Under the SC MoC, their progression and the context switches between them are disciplined by a scheduling regime that prohibits race conditions. Determinism in Control is achieved by demanding that in any pair of *concurrent* write/read accesses to a shared variable, the write must be scheduled before the read. For example, the write to checkReq in node L13 of the SCG (Fig. 2c), corresponding to line 13 of the SCL program (Fig. 2d), is in a different concurrent thread, relative to the read of checkReq (L23). Hence thread Request must be scheduled such that it executes L13 before Dispatch executes L23.

A common means to visualize program traces in synchronous languages is a *tick time line*, as shown in Fig. 2b. As can be seen there, in the first tick, the inputs free = *true*, req = *false* produce the outputs grant = pend = *false*, under the concurrent write-before-read scheduling sketched above.

An interesting characteristic of Control is that the concurrent threads not only share variables, but also modify them sequentially. E. g., Dispatch first initializes grant with *false*, and then, in the same tick, might set it to *true*. Similarly, Request might assign to pend the sequence *false*/*true*/*false*. Due to the prescribed sequential ordering of these assignments, this does not induce any non-determinism. However, this would not be permitted under the strict synchronous model of computation, which requires unique variable values per tick. Similarly, pend is read (L14) and subsequently written to (L15); this (sequential) write-after-read is again harmless, although forbidden under the existing synchronous MoC. However, because it is possible to schedule Control such that all concurrent write-before-read requirements are met and all such schedules lead to the same result, we consider Control *sequentially constructive*. The rest of this paper consists of making this notion precise, and describing a practical strategy to analyze sequential constructiveness and to implement schedules that adhere to the SC model of computation. One building block is the graph abstraction presented in the next section.

3.3 The SC Graph

The SCG syntactical elements are presented in Fig. 3. The SCG for the Control example is shown in Fig. 2c. An SCG is a labelled graph $G = (N, E)$ whose *statement nodes* N correspond to the statements of the program, and whose edges E reflect the sequential execution ordering and data dependencies between the statements. Every edge $e \in E$ connects a source $e.src \in N$ with a target node $e.tgt \in N$. Nodes and edges are further described by various attributes.

A node n is labelled by the statement type $n.st$ that it represents, viz. $n.st \in \{\mathsf{entry}, \mathsf{exit}, \mathsf{goto}, x = ex, \mathsf{if}\,(ex), \mathsf{fork}, \mathsf{join}, \mathsf{surf}, \mathsf{depth}\}$, where x is some variable and ex is some expression.[1] Nodes labelled with $x = ex$ are referred to as *assignment nodes*, those with if ex as *condition nodes*, those with surf as *surface nodes*; all other nodes are referred by their statement type (*entry nodes*, *exit nodes*, etc.). As illustrated in Fig. 3, in the graphical representations of the SCG the shape of a node indicates the statement type, except for entry/exit/goto nodes, which all are shown as ovals; they share the characteristic that they mainly serve to structure the SCG and could be eliminated without changing the meaning of an SCG. The statement types of SCG nodes are closely related, but not identical to the primitive statements of the SCL language presented in Sec. 3.1; how these statements relate to each other is elaborated in Sec. 3.4.

Every edge e has a type $e.type \in \{seq, tick, wr, wi, ir, ww\}$ that specifies the nature of the particular ordering constraint expressed by e. We write $e.src \rightarrow_\alpha e.tgt$, pronounced "$e.src$ α-precedes $e.tgt$," if $e.type = \alpha$.

Edges of the form $n_1 \rightarrow_{seq} n_2$ and $n_1 \rightarrow_{tick} n_2$ are induced directly from the source program. In the former case the two nodes n_1 and n_2 are *sequential* successors, and in the latter case *tick successors*. Collectively they are referred to as *flow successors*, and edges of type *seq* or *tick* are referred to as *flow edges*. A path consisting exclusively of flow edges is referred to as *flow path*. We use $\twoheadrightarrow_{seq}$ for the reflexive and transitive closure of $\rightarrow_{seq}$.

Informally, $n_1 \rightarrow_{seq} n_2$ holds if the statements may execute in the same tick, and a sequential control flow enforces n_1 to be executed immediately before n_2. In other words, the program order never generates a situation in which the scheduler is free to choose between n_1 or n_2. Note that $n_1 \rightarrow_{seq} n_2$ does not necessarily mean that there is a fixed run-time ordering between n_1 and n_2. For example, when n_1 and n_2 are enclosed in a loop, there might be an execution sequence n_1, n_2, n_1, n_2 within the same tick.

The relationship $n_1 \rightarrow_{tick} n_2$ says that there is a tick border between n_1 and n_2, i. e., the control flow passes from n_1 to n_2 not instantaneously in the same tick, as with $n_1 \rightarrow_{seq} n_2$, but only upon a global clock tick. All other types of edges $n_1 \rightarrow_\alpha n_2$ for $\alpha \in \{wr, wi, ir, ww\}$ are derived for the purpose of scheduling analysis and discussed later in Sec. 6.

3.4 Mapping SCL to an SCG

Fig. 1 gives a schematic overview of how the SCL statements introduced in Sec. 3.1 correspond to an SCG. This is formally described in the following.

To handle compound statements, we need the concept of an SCG subgraph.

Definition 1 (SCG subgraphs)**.** *For an SCG $G = (N, E)$, an* SCG subgraph $G_{sub} = (N_{sub}, E_{sub}, in, Out)$ *consists of a set of statement nodes $N_{sub} \subseteq N$, a set of edges $E_{sub} \subseteq N \times N$, an* incoming

[1] Strictly speaking, "$x = ex$" and "$\mathsf{if}\,(ex)$" each denote a multitude of statements, ranging over all variables x and expressions ex. However, to not make the notation unnecessarily heavy, we here treat them like the other statements that are not parameterized.

node $in \in N_{sub}$, *and a set of* outgoing nodes $Out \subseteq N_{sub}$. *These elements are also referred to as* $G_{sub}.N_{sub}$, *etc.*

The SCG subgraph corresponding to an SCL statement s is denoted $SCG(s)$.

- An SCL program s corresponds to an SCG consisting of the nodes and edges of $SCG(s)$, plus an entry node n_e, an exit node n_x, a *seq*-edge from n_e to $SCG(s).in$, and *seq*-edges from all $n \in SCG(s).Out$ to n_x.

- For s being an assignment x = ex, we stipulate $SCG(s) = (\{n\}, \emptyset, n, \{n\})$, where n is an assignment node with $n.st =$ "x = ex."

- For s being a sequence s_1 ; s_2, we have $SCG(s) = \{SCG(s_1).N_{sub} \uplus SCG(s_2).N_{sub}$, $SCG(s_1).E_{sub} \uplus SCG(s_2).E_{sub} \uplus E', SCG(s_1).in, SCG(s_2).Out\}$, where $\uplus$ is disjoint union and E' consists of *seq*-edges from the nodes in $SCG(s_1).Out$ to $SCG(s_2).in$.

- If s is a conditional if (ex) s_1 else s_2, then $SCG(s) = \{SCG(s_1).N_{sub} \uplus SCG(s_2).N_{sub} \uplus \{n\}, SCG(s_1).E_{sub} \uplus SCG(s_2).E_{sub} \uplus E', n, SCG(s_1).Out \cup SCG(s_2).Out\}$, where n is a conditional node with $n.st =$ "if ex," and E' contains edges $n \rightarrow_{seq} true(n)$ and $n \rightarrow_{seq} false(n)$, in which $true(n) := SCG(s_1).in$ and $false(n) := SCG(s_2).in$ are the two uniquely defined *true* and *false branch nodes* of the conditional.

- For s being a goto l statement that jumps to a labeled statement $l : s'$, it is $SCG(s) = \{\{n\}, \{e\}, n, \emptyset)\}$, where n is a goto node, i. e., $n.st =$ goto, and e an edge $n \rightarrow_{seq} SCG(s').in$.

- For s being a parallel statement fork s_1 par s_2 join, it is $SCG(s) = \{SCG(s_1).N_{sub} \uplus SCG(s_2).N_{sub} \uplus \{n_f, n_j, n_{e1}, n_{e2}, n_{x1}, n_{x2}\}, SCG(s_1).E_{sub} \uplus SCG(s_2).E_{sub} \uplus E', n_f, \{n_j\}\}$, where n_f is a fork node and n_j a join node; furthermore, n_{e1}, n_{e2} are entry nodes and n_{x1}, n_{x2} exit nodes for each created thread that are connected to the fork and join node, respectively, and to $SCG(s_1)$ and $SCG(s_2)$ as indicated in Fig. 1. Observe that the join node n_j is uniquely associated with the fork n_f, which can be expressed by writing $n_j = join(n_f)$. As noted earlier, for simplicity we here only consider parallelism of degree two, but an extension to higher degrees would be straightforward.

- For s being a pause statement, it is $SCG(s) = \{\{n_s, n_d\}, \{e\}, n_s, \{n_d\}\}$, where n_s is a surface node and n_d a depth node, and e is a *tick*-edge from n_s to n_d. This models the fact that the statement can be active at the end of a tick and at the beginning of the subsequent tick. In a well-formed SCG n_d is the unique tick successor $tick(n_s) = n_d$ of n_s. i. e., whenever $n.st =$ surf then $n \rightarrow_{tick} tick(n)$ and $tick(n).st =$ depth.

3.5 Thread Terminology

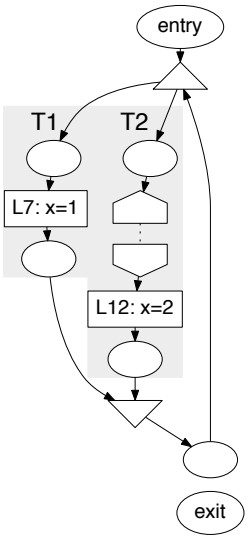
We distinguish the concept of a static *thread*, which relates to the structure of a program, from a dynamic *thread instance* (see Sec. 3.10), which relates to a program in execution. We here define our notion of (static) threads, building on the SCG program representation $G = (N, E)$ with statement nodes N and control flow edges E.

Let T denote the set of threads of G. Each thread $t \in T$, including the top-level Root thread, is associated with unique entry and exit nodes $t.en$, $t.ex \in N$ with *statement types* $t.en.st =$ entry and $t.ex.st =$ exit.

Each $n \in N$ belongs to a thread $th(n)$, defined as the immediately enclosing thread $t \in T$ such that there is a flow path to n (as defined in Sec. 3.3) that originates in $t.en$ and that does

```
1  module Reinc
2  output int x, y;
3  {
4   loop:
5    fork
6      // Thread T1
7      x = 1;
8
9    par
10     // Thread T2
11     pause;
12     x = 2;
13
14   join;
15   goto loop;
16 }
```

(a) The SCL program



(b) The SCG

Figure 4: The Reinc example, illustrating thread reincarnation. The assignments to x are both executed in the same tick, yet are sequentialized.

not traverse any other entry node $t'.en$, unless that flow path subsequently traverses $t'.ex$ also. For each thread t we define $sts(t)$ as the set of statement nodes $n \in N$ such that $th(n) = t$. For example, the Control program (Fig. 2) consists of the threads $T = \{\mathsf{Root}, \mathsf{Request}, \mathsf{Dispatch}\}$, and the Root thread consists of the statement nodes $sts(\mathsf{Root}) = \{\mathsf{L0}, \mathsf{L7}, \mathsf{L28}, \mathsf{L29}\}$. The remaining statement nodes of N are partitioned into $sts(\mathsf{Dispatch})$ and $sts(\mathsf{Request})$, as indicated in Fig. 2c.

We define $fork(t)$ to be the fork node that immediately precedes $t.en$. Every thread $t \neq \mathsf{Root}$ has an immediate *parent thread* $p(t)$, defined as $th(fork(t))$. In the Control example, $p(\mathsf{Request}) = p(\mathsf{Dispatch}) = \mathsf{Root}$.

We are now ready to define (static) thread concurrency and subordination:

Definition 2 (Ancestor/Subordinate/Concurrent Threads)**.** *Let t, t_1, t_2 be threads in T.*

1. *The set of* ancestor *threads of t is recursively defined as $p^*(t) = \{t, p(t), p(p(t)), \ldots, \mathsf{Root}\}$.*
2. *t_1 is* subordinate *to t_2, written $t_1 \prec t_2$, if $t_1 \neq t_2$ and $t_1 \in p^*(t_2)$.*
3. *t_1 and t_2 are* concurrent*, denoted $t_1 \parallel t_2$, iff they are descendants of distinct threads sharing a common fork node, i. e., iff there exist $t_1' \in p^*(t_1)$, $t_2' \in p^*(t_2)$ with $t_1' \neq t_2'$ and $fork(t_1') = fork(t_2')$. We then refer to this fork node as the* least common ancestor fork*, denoted* $\mathrm{lcafork}(t_1, t_2)$*. This is lifted to nodes, i. e., if $th(n_1) \parallel th(n_2)$ then $lcafork(n_1, n_2) = lcafork(th(n_1), th(n_2))$.*

In Control, it is $\mathsf{Root} \prec \mathsf{Request}$ and $\mathsf{Root} \prec \mathsf{Dispatch}$. It is also $\mathsf{Request} \parallel \mathsf{Dispatch}$, whereas Root is not concurrent with any thread. Note that concurrency on threads, in contrast to concurrency on node instances (Def. 4), is purely static and can be checked with a simple, syntactic analysis of the program structure.

3.6 Thread Reincarnation—The Reinc Example

The Reinc example shown in Fig. 4 illustrates that *static* thread concurrency is not sufficient to capture whether individual statements are *run-time concurrent*, in the sense that it would be

up to the discretion of a scheduler how they should be ordered. Consider the assignments x = 1 (L7) and x = 2 (L12). These are in the concurrent threads T1 and T2, and can be activated in the same tick, but they are still sequentially ordered and thus not run-time concurrent. This is due to the *reincarnation* of T2, which takes place as follows.

In the initial tick, T1 executes L7 and terminates, and T2 rests at L11 (pause). Thus there is one instance of each thread that executes in the initial tick. In subsequent ticks, first T2 continues with L12 and terminates, which enables the join (L14) and, after the loop, both T1 and T2 get started again, in a second instance during the same tick, and as in the initial tick, T1 executes L7 and terminates, and T2 rests at L11. Thus, L12 gets executed in the *first* instance of T2, but it is the *second* instance of T2 that is concurrent to the execution of L7. As a result, the concurrent writes to x in L12 and L7 are cleanly separated through the sequential loop, and the deterministic result for x at the end of each tick is the value 1.

The definition of the least common ancestor fork (Def. 2) helps to capture this. The fact that L12 and L7 are not run-time concurrent is because their executions go back to different executions of $lcafork(\mathsf{L12}, \mathsf{L7}) = \mathsf{L5}$. L12 is executed in an instance of T2 that was forked off by an execution of L5 in the previous tick, whereas L7 is executed in an instance of T1 forked off by an execution of L5 in the current tick. Thus our definition of run-time concurrency of two statements n_1, n_2, provided in the following, also refers to instances (executions) of $lcafork(n_1, n_2)$.

3.7 Thread Trees—The Reinc2 Example

The Reinc2 example shown in Fig. 5 is a more elaborate variant of Reinc that illustrates a more complex thread structure. Specifically, we have T21 || T22, T23 || T24, and T1 || t for all $t \in \{$T2, T21, T22, T23, T24$\}$.

The *thread tree*, shown in Fig. 5c, is a means to visualize the thread relationships. It contains a subset of the SCG nodes, namely the thread entry nodes, labeled with the names of their threads, and the fork nodes, which are attached to the entry nodes of their threads. Threads are concurrent iff their least common ancestor (lca) in the thread tree is a fork node. E. g., T21 and T23 are not concurrent, because their lca is the thread entry node T2, meaning that T21 and T23 are sequential within T2. The thread tree is conceptually similar to the AND/OR tree used to illustrate state relationships in Statecharts [4]; fork nodes correspond to AND states, denoting concurrency, whereas the thread entry nodes correspond to OR states, denoting exclusive/sequential behavior.

3.8 Statement Reincarnation—The InstLoop Example

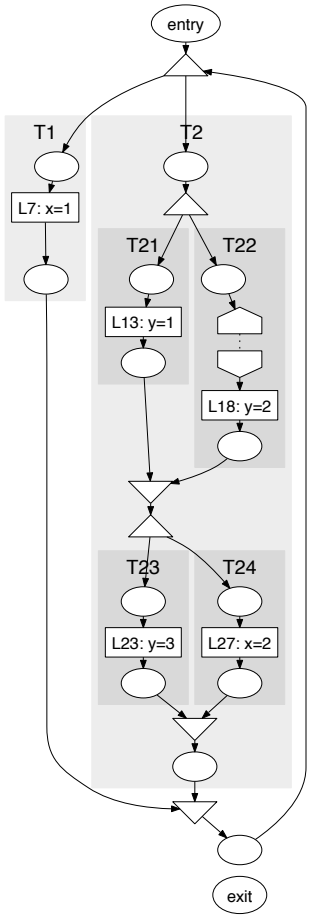
The InstLoop example shown in Fig. 6 illustrates the issue of statement reincarnation. In particular, the statically concurrent accesses to x in L7 and L11 are executed twice within a tick, because the loop iterates two times. Because of the data dependency on x, L7 must be scheduled before L11—but only within the same loop iteration.

Traditional synchronous programming would consider such a loop that does not separate iterations by pause statements to be *instantaneous*, and hence reject this program, on the justification that the instantaneous loop might potentially run forever. However, the SC MoC does not have a problem with this program, as it is still deterministic. The loop in InstLoop also happens to be executed only two times. Of course, one might still want to ensure that a program always terminates, but this issue is orthogonal to determinism and having a well-defined semantics.

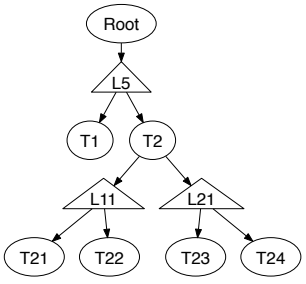
As this example highlights, it is not enough to impose an order on the program statements. To capture precisely the concept of sequential constructiveness, we need to distinguish statement

```
1  module Reinc2
2  output int x, y;
3  {
4   loop:
5    fork
6      // Thread T1
7      x = 1;
8
9    par
10     // Thread T2
11     fork
12       // Thread T21
13       y = 1;
14
15     par
16       // Thread T22
17       pause;
18       y = 2;
19
20     join;
21     fork
22       // Thread T23
23       y = 3;
24
25     par
26       // Thread T24
27       x = 2;
28
29     join
30
31   join;
32   goto loop;
33 }
```

(a) The SCL program



(b) The SC Graph (SCG)



(c) The thread tree

Figure 5: The Reinc2 example, a more elaborate example of thread concurrency and thread reincarnation

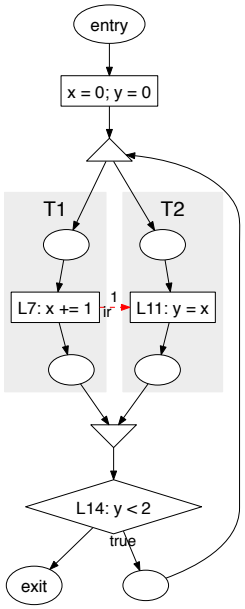
instances. The key here is again the least common ancestor fork; the specific executions of L7 and L11 that go back to the same execution of *lcafork*(L7, L11) = L5 must be ordered.

3.9 Macro Ticks, Micro Ticks, and the Thread Status

As already described, the externally observable execution of a synchronous program consists of a sequence of macro ticks. Internally, however, one typically breaks down a macro tick into a series of *micro ticks*, both for describing the semantics and for a concrete implementation. We call this series of micro ticks a *run*, whereas a *trace* describes only the externally visible output

```
1  module InstLoop
2  output int x = 0, y = 0;
3  {
4   loop:
5    fork
6      // Thread T1
7      x += 1;
8
9    par
10     // Thread T2
11     y = x;
12
13   join;
14   if (y < 2)
15     goto loop;
16 }
```

(a) The SCL program



(b) The SC Graph (SCG).

Figure 6: The InstLoop example, with an instantaneous loop that leads to statement reincarnation within a tick

values emitted at each macro tick.

Definition 3 (Ticks)**.** *For an SCG $G = (N, E)$, a* (macro) tick *R, of length $len(R) \in \mathbb{N}_{\geq 1}$, is a mapping from micro tick indices $1 \leq j \leq len(R)$, to nodes $R(j) \in N$. A* run *of G is a sequence of macro ticks R^a, indexed by $a \in \mathbb{N}_{\geq 1}$.*

Another way to look at a macro tick is as a linearly ordered set of *node instances*, viz. pairs $ni = (n, i)$ consisting of a statement node $n \in N$ and a micro tick count $i \in \mathbb{N}$. Concretely, each R can be identified with the set $\{(n, i) \mid 1 \leq i \leq len(R), n = R(i)\}$. A special case is the *empty* macro tick with $len(R) = 0$ and $R = \emptyset$. Sometimes it is convenient to view macro ticks as sequences of nodes $R = n_1, n_2, \ldots, n_k$ where $k = len(R)$ and $n_i = R(i)$ for all $1 \leq i \leq k$.

One possible run of the Control example is illustrated in Fig. 2e. We also see for each micro tick the node that is scheduled next for execution. An assignment results in an update of the written variable, reflected by the variable values of the subsequent micro tick. The *continuations*, explained further in Sec. 4, denote the current state of each thread, i. e., the node (statement) that should be executed next, similar to a program counter. In addition, a continuation denotes what execution state a thread is in. Fig. 7 illustrates the possible states using a SyncChart notation. Threads other than the Root thread are initially *disabled*, with a status denoted by $\perp$. When a thread gets forked by its parent, it becomes *enabled*. Enabled threads are initially *active*, i. e., eligible for execution, with a status denoted by the node to be executed next. When an active thread forks off some child threads, it becomes *waiting*, denoted by the node to be executed next in square brackets, until the child threads join and the parent becomes active again. An active thread that executes a pause statement and thus finishes its current tick becomes *pausing*, denoted in parentheses, until the next tick is started and it becomes active again.

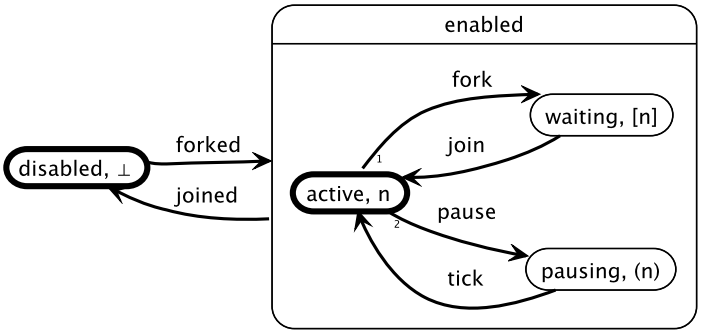


Figure 7: Execution states of a thread

3.10 Concurrency of Node Instances

For a macro tick R, an index $1 \leq i \leq len(R)$, and a node $n \in N$, $last(n,i) = max\{j \mid j \leq i, R(j) = n\}$ retrieves the last occurrence of n in R at or before index i. If it does not exist, $last(n,i) = 0$.[2] The function $last(n,i)$ is instrumental to define concurrency of node instances as discussed above in Secs. 3.6 and 3.8.

Definition 4 (Concurrent Node Instances)**.** *For a macro tick R, $i_1, i_2 \in \mathbb{N}_{\leq len(R)}$, and $n_1, n_2 \in N$, two node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$ are* concurrent *in R, denoted $ni_1 \mid_R ni_2$, iff*

1. *they appear in the micro ticks of R, i. e., $n_1 = R(i_1)$ and $n_2 = R(i_2)$,*
2. *they belong to statically concurrent threads, i. e., $th(n_1) \mid\mid th(n_2)$, and*
3. *their threads have been instantiated by the same instance of the associated least common ancestor fork, i. e., $last(n, i_1) = last(n, i_2)$ where $n = lcafork(n_1, n_2)$.*

In the Control example of Fig. 2e, the variable accesses that are *concurrent* involve checkReq and grant. We call this particular run *admissible* because all concurrent variable accesses follow certain admissibility rules, defined in detail later (Sec. 5.2, pages 5.2 ff). The run here is admissible because the write to checkReq (L13) is scheduled before the corresponding read (L23), and similarly the writes to grant (L22 and potentially L24) are scheduled before the read (L14).

4 "Free" Scheduling of SCGs

With the above preliminaries in place, we now come to discuss the semantics of SCL. We do this by looking at the execution and scheduling of a fixed SCG $G = (N, E)$ associated with some arbitrary program. We begin by considering the "free execution" of G based on the program orders $\rightarrow_{seq}$ and $\rightarrow_{tick}$. Our notion of sequential constructiveness will then arise from a further restriction of the "free" schedules guided by additional sequentiality orders $\rightarrow_\alpha$ for $\alpha \in \{wr, wi, ir, ww\}$.

Traditional schedulers work at machine instruction granularity. This means that thread context switches can basically occur anywhere within any statement. In principle, we could

[2]Strictly, $last(n,i)$ should be written $last(R,n,i)$ as it depends on the macro tick R, not only on n and i. For notational compactness, we leave this implicit, however.

allow this flexibility also for the SC MoC. For simplicity, we restrict ourselves to scheduling at the statement level. In particular, the evaluation of expressions and the update of variables in assignments happens atomically. On the other hand, a conditional statement if (ex) s_1 else s_2 sequentially separates the decision based on the (atomic) evaluation of ex from the execution of the appropriate branching command s_i.

4.1 Continuations and Continuation Pool

Our simulation semantics is based on *continuations* which are instances of program nodes from the SCG enriched with information about the run-time context in which the nodes are executed. In general imperative languages this may comprise explicit thread identification, instance numbers, local memory, reference to stack frames and other scheduling information. For the simple SCL language considered in this paper and for the "free" scheduling to be defined in this section, only few data are needed. It suffices to let continuations contain the currently running node in the SCG and a *scheduling status* to indicate whether the continuation is *active*, *waiting* or *pausing*.

Formally, a *continuation* c has a node $c.node \in N$ and a status $c.status \in \{\mathsf{active}, \mathsf{waiting}, \mathsf{pausing}\}$. We refrain from identifying continuations with the pairs $(c.node, c.status)$ in order to allow for other attributes in enriched versions of the semantics. Instead, we write $c[s :: n]$ when $s = c.status$ and $n = c.node$, or to express that the status and node of a continuation c are updated to be s and n, respectively. The waiting status is derivable from subordination of threads: If $th(c'.node) \prec th(c.node)$, then c' runs in a proper ancestor thread of c and thus c' must wait for c. We overload notation and write $c' \prec c$ in this case.

The statuses of continuations (defined here) and threads (introduced in Sec. 3.9) are closely related, but not identical concepts. For a thread that is enabled and hence has a continuation, the status of the thread is the status of its continuation; however, there is no disabled status for continuations, and disabled threads do not have a continuation associated with them.

At every micro tick of an execution run, the scheduler picks an active continuation from a pool of continuations and executes it. A *continuation pool* may be modelled simply as a finite set C of continuations, subject to some constraints outlined below, since all information is contained in the attributes of its elements.

Initially, C only contains the main program Root activated at its entry node Root.en. Then, every time an active fork node is executed the corresponding join node is installed in the same thread as the fork. This thread is subordinate to the threads of the children spawned by the fork which thus block the execution of the continuation in the parent thread. In this fashion, the join node of the parent thread starts with status waiting until the children are terminated, at which point its status becomes active. The switching between active and pausing happens in the execution of the pause statement. When an active surf node is scheduled, its status changes to pausing, thereby suspending it for the current macro tick. When the execution switches to the next macro tick, the thread is re-activated at the uniquely associated depth node.

Looking back at the execution run of Fig. 2b we can see the thread pool C_i^a in micro tick i of macro tick a being made up of the entries in the rows C_{Root}, C_{Request} and C_{Dispatch} at column index i. The entries show the continuations' nodes and status. Nodes in square brackets $[n]$ are waiting, those in brackets (n) are pausing and all othere are active.

Continuation pools satisfy some coherence properties. The waiting continuations $c[\mathsf{waiting} :: n] \in C$ are precisely those continuations in C that are not $\prec$-maximal in C and they must always be join nodes, i. e., $n.st = \mathsf{join}$. All $\prec$-maximal continuations have status active or

pausing. Of those, the pausing continuations $c[\mathsf{pausing} :: n] \in C$ must be surface nodes, i. e., $n.st = \mathsf{surf}$. Moreover, the threads appearing in a continuation pool preserve the tree structure. For each $c \in C$ such that $th(c.node) \neq \mathsf{Root}$, there is a unique *parent* $c' \in C$ such that $p(th(c.node)) = th(c'.node)$. By construction, the parent continuation always corresponds to a waiting join statement, i. e., $c'.node.st = \mathsf{join}$ and $c'.status = \mathsf{waiting}$. A continuation pool satisfying these constraints is called *valid*.

4.2 Configurations, Micro Step and Macro Step Scheduling

A *configuration* is a pair (C, ρ) where C is a pool of continuations and ρ is a memory assigning values to the variables accessed by G. A configuration is called *valid* if C is valid. We wish to perform a single scheduling step to move from the current configuration (C_{cur}, ρ_{cur}) to the next configuration (C_{nxt}, ρ_{nxt}). In general, this involves the execution of one or more continuations from C_{cur}. Our "free" schedule is restricted (i) to execute only $\prec$-maximal threads and (ii) to do so in an interleaving fashion:

1. *Micro Step.* If there is at least one continuation in C_{cur}, then there also is a $\prec$-maximal one, because of the finiteness of the continuation pool. The "free" schedule is permitted to pick any one of the $\prec$-maximal continuations $c \in C_{cur}$ with $c.status = \mathsf{active}$ and execute it in the current memory ρ_{cur}. This yields a new memory $\rho_{nxt} = upd(c, \rho_{cur})$ and a (possibly empty) set of new continuations $nxt(c, \rho_{cur})$ by which c is replaced, i. e., $C_{nxt} = C_{cur} \setminus \{c\} \cup nxt(c, \rho_{cur})$. Note that in C_{nxt} the status flags are automatically set to active for all continuations that become $\prec$-maximal by the elimination of c from the pool in case $nxt(c, \rho_{cur}) = \emptyset$.

   The actions *upd* and *nxt* depend on the statement $c.node.st$ to be executed and will be defined shortly. A *micro step* (a transition between two micro ticks) is written $(C_{cur}, \rho_{cur}) \xrightarrow{c}_{\mu s} (C_{nxt}, \rho_{nxt})$, where c is the continuation that is selected for execution. Since (C_{nxt}, ρ_{nxt}) is uniquely determined by the executed continuation c we may write it as $(C_{nxt}, \rho_{nxt}) = c(C_{cur}, \rho_{cur})$.

2. *Clock Step.* When there is no active continuation in C, then all continuations in C are pausing or waiting. We call this a *quiescent* configuration. In the special situation where $C = \emptyset$ the main program has terminated. Otherwise, and only then, the scheduler can perform a global clock step, i. e., a transition between the last micro tick of the current macro tick to the first micro tick of the subsequent macro tick. This is done by letting all pausing continuations of C advance from their surf node to the associated depth node. More precisely,

   $$C_{nxt} = \{c[\mathsf{active} :: tick(n)] \mid c[\mathsf{pausing} :: n] \in C_{cur}\} \cup \{c[\mathsf{waiting} :: n] \mid c[\mathsf{waiting} :: n] \in C_{cur}\}.$$

   Let $I = \{x_1, x_2, \ldots, x_n\}$ be the designated input variables of the SCG, including input/output variables. Then the memory is updated by a new set of *external input* values $\alpha = [x_1 = v_1, \ldots, x_n = v_n]$ for the next macro tick. All other memory locations persist unchanged into the next macro tick. Formally,

   $$\rho_{nxt}(x) = \begin{cases} v_i, & \text{if } x = x_i \in I, \\ \rho_{cur}(x), & \text{if } x \notin I. \end{cases}$$

   A clock step is denoted $(C_{cur}, \rho_{cur}) \xrightarrow{\alpha}_{tick} (C_{nxt}, \rho_{nxt})$, where α is the external input. Observe that since the set of inputs I is assumed to be fixed globally, both α and ρ_{nxt} can be derived from each other and from ρ_{cur}.

A *synchronous instant*, or *macro tick*, of the SCG G is a maximal sequence of micro steps of G. More concretely, the scheduler runs through a sequence

$$(C_0^a, \rho_0^a) \overset{c_1^a}{\to}_{\mu s} (C_1^a, \rho_1^a) \overset{c_2^a}{\to}_{\mu s} \cdots \overset{c_{k(a)}^a}{\to}_{\mu s} (C_{k(a)}^a, \rho_{k(a)}^a) \tag{1}$$

of micro steps obtained from the interleaving of active continuations, to reach a final quiescent configuration $(C_{k(a)}^a, \rho_{k(a)}^a)$, in which all continuations are pausing or waiting. We write $(C_0^a, \rho_0^a) \twoheadrightarrow_{\mu s} (C_i^a, \rho_i^a)$ to express that there exists a sequence of micro steps, not necessarily maximal, from configuration (C_0^a, ρ_0^a) to (C_i^a, ρ_i^a), dropping the information on continuations. The complete sequence (1) from start to end is the *macro tick*, abbreviated

$$(C_0^a, \rho_0^a) \overset{\alpha^a/R^a}{\Longrightarrow} (C_{k(a)}^a, \rho_{k(a)}^a), \tag{2}$$

where the label α^a projects the initial input, i. e., $\alpha^a(x) = \rho_0^a(x)$ for $x \in I$. The final memory state $\rho_{k(a)}^a$ of the quiescent configuration is the *response* of the macro tick a. The label R^a is the sequence of statement nodes executed during the macro tick as described in Def. 3. More precisely, $len(R^a) = k(a)$ is the length of the macro tick and R^a the function mapping each micro tick index $1 \leq j \leq k(a)$ to the node $R^a(j) = c_j^a.node$ executed at index j. For example, in the execution run shown in Fig. 2b, the (doubly indexed) node sequence $R^a(j)$ is given by the last row named "Scheduled nodes."

Note that in (2) the input label α^a may be dropped since it can be derived from I and ρ_0. Moreover, given R^a, the final quiescent configuration is functionally determined, provided no two distinct continuations can share the same statement node. We may thus write $(C_{k(a)}^a, \rho_{k(a)}^a) = R^a(C_0^a, \rho_0^a)$. When the memory state and scheduling information is not needed it is convenient to identify a macro tick with R^a as its abstraction. It can be viewed, alternatively, as a set of node instances $R^a = \{(c_i^a.node, i) \mid 1 \leq i \leq k(a)\}$ or a sequence $R^a = c_1^a.node, c_2^a.node, \ldots, c_{k(a)}^a.node$.

We call the end points of a macro tick (2) *macro (tick) configuration*, while all other intermediate configurations (C_i^a, ρ_i^a), $0 < i < k(a)$ seen in (1) are *micro (tick) configurations*. According to the Synchrony Hypothesis we assume that only macro configurations are observable externally (in fact, only the memory component of those). Hence, it suffices to ensure that the sequence of macro ticks $\Longrightarrow$ is deterministic, while the micro tick behavior $\to_{\mu s}$ may well be non-deterministic. Formally, for any two given sequences of (reachable) macro configurations

$$(C_0^a, \rho_0^a) \overset{\alpha^a/R^a}{\Longrightarrow} (C_{k(a)}^a, \rho_{k(a)}^a) \to_{tick} (C_0^{a+1}, \rho_0^{a+1})$$

and

$$(C_0'^a, \rho_0'^a) \overset{\alpha'^a/R'^a}{\Longrightarrow} (C_{k'(a)}'^a, \rho_{k'(a)}'^a) \to_{tick} (C_0'^{a+1}, \rho_0'^{a+1}),$$

if the initial continuation pool and memory are the same, $C_0^0 = C_0'^0$, $\rho_0^0 = \rho_0'^0$ and if also the input sequences are the same, i. e., for all macro ticks a, $\alpha^a = \alpha'^a$, then the sequence of responses is identical, too, i. e., $\rho_{k(a)}^a = \rho_{k'(a)}'^a$ for all a.

It remains to define the actions *upd* and *nxt* exercised by active continuations on memory ρ and continuation pool C, respectively. The former is easy to specify. The only statement $c.node.st$ to affect the memory is the assignment statement x = ex. In this case variable x is updated by the value of ex. Formally, $upd(c, \rho)(x) = eval(ex, \rho)$ and $upd(c, \rho)(y) = \rho(y)$ for $y \neq x$. In all other cases, if $c.node.st$ is not an assignment, we have $upd(c, \rho) = \rho$.

The action of a continuation on the continuation pool is only slightly more involved. For $c[\mathsf{active} :: n] \in C$ the set $nxt(c, \rho)$ is given thus:

- For $n.st \in \{\mathsf{entry}, \mathsf{goto}, x = ex, \mathsf{depth}, \mathsf{join}\}$ the continuation c passes on control to its immediate sequential successor, i. e., $nxt(c, \rho) = \{c[\mathsf{active} :: n_1]\}$, where n_1 with $n \rightarrow_{seq} n_1$ is uniquely determined.

- At an exit node $n.st = \mathsf{exit}$ we have reached the end of the continuation, which terminates and disappears from the pool, i. e., $nxt(c, \rho) = \emptyset$.

- When $n.st = \mathsf{surf}$, then we set the continuation pausing to wait at this node for the next synchronous tick. i. e., $nxt(c, \rho) = \{c[\mathsf{pausing} :: n]\}$.

- Consider a conditional statement $n.st = \mathsf{if}\,(ex)$ with the uniquely determined successor nodes $n_1 = true(n) \in N$ and $n_2 = false(n) \in N$ for its true and false branch, respectively. The execution of n takes one of the branches according to the boolean value of ex, so that $nxt(c, \rho) = \{c[\mathsf{active} :: n_i]\}$, where $i = 1$ if $eval(ex, \rho) = \mathsf{true}$ and $i = 2$ if $eval(ex, \rho) = \mathsf{false}$. Note that in each case $n \rightarrow_{seq} n_i$.

- Finally, suppose c instantiates a fork statement with edges $n \rightarrow_{seq} n_1$ and $n \rightarrow_{seq} n_2$ leading to the two entry nodes $n_1, n_2 \in N$ of its concurrent child threads. Let $n_3 = join(n) \in N$ be the join node uniquely associated with n. Then, $nxt(c, \rho) = \{c[\mathsf{active} :: n_1], c[\mathsf{active} :: n_2], c[\mathsf{waiting} :: n_3]\}$. Hence the "free" scheduler may execute n_1 or n_2 in any order, but both have to terminate before the join statement n_3 can resume.

The following two technical observations are useful to derive further facts about the operational semantics of SCGs, such as our main Thm. 1 in Sec. 6. Informally, Prop. 1, the first observation, states that for each active continuation c', there is a sequential predecessor c, and that nodes that are sequentially related cannot be active simultaneously.

Proposition 1 (Active continuations have sequential predecessors)**.** *Let $(C, \rho) \twoheadrightarrow_{\mu s} (C', \rho')$ be a micro tick sequence and $c' \in C'$ an active continuation. Then there exists an active $c \in C$ such that $c.node \twoheadrightarrow_{seq} c'.node$.*

Proof. We proceed by induction on the length of the micro tick sequence $(C, \rho) \twoheadrightarrow_{\mu s} (C', \rho')$. If the length is zero, $(C, \rho) = (C', \rho')$, and the claim follows trivially by reflexivity of $\twoheadrightarrow_{seq}$. For the inductive step assume a sequence of length greater than zero, say

$$(C, \rho) \twoheadrightarrow_{\mu s} (C'', \rho'') \overset{c}{\rightarrow}_{\mu s} (C', \rho')$$

and $c' \in C'$ is active, i. e., $c' = c'[\mathsf{active} :: n']$ where $n' = c'.node$.

If $c' \in C''$, then the claim follows by induction hypothesis, directly. If $c' \notin C''$ then continuation c' has entered the pool C' as a result of executing the active continuation c'' from (C'', ρ''), i. e., $c' \in nxt(c'', \rho'')$ with $c'' = c''[\mathsf{active} :: n]$ and $n = c''.node$. A simple case analysis on the statement type $n.st$ and the definition of $\rightarrow_{\mu s}$ shows that $n \rightarrow_{seq} n'$. Informally, since $c' \in nxt(c'', \rho'')$ is active, its node n' cannot be a surf statement, which would be pausing nor a join statement, which would be waiting. All the remaining cases produce nodes n' which are sequential successors of n. Since c'' is active in C'' we have $c'' \in C''$. Applying the induction hypothesis to c'' yields an active continuation $c \in C$ with $c.node \twoheadrightarrow_{seq} c''.node = n$. By transitivity of $\twoheadrightarrow_{\mu s}$ this implies $c.node \twoheadrightarrow_{seq} n' = c'.node$. □

Proposition 2 (Active and pausing continuations are concurrent)**.** *Let (C, ρ) be a reachable (micro or macro tick) configuration for the SCG G. Then, for any two active or pausing continuations $c_1, c_2 \in C$ with $c_1 \neq c_2$ we have $c_1.node \neq c_2.node$ and $th(c_1.node) \mid\mid th(c_2.node)$. In particular, $c_1.node \not\twoheadrightarrow_{seq} c_2.node$ and $c_2.node \not\twoheadrightarrow_{seq} c_1.node$.*

Proof. First note that concurrent nodes cannot be sequentially ordered, so that it suffices to prove $th(c_1.node) \parallel th(c_2.node)$ for all active or pausing nodes. Also, the invariant holds trivially in the initial configuration (C_0, ρ_0) for G in which C_0 only contains the root-level entry node Root.*en* as its only (active) configuration.

Now suppose (C, ρ) is a reachable configuration satisfying the invariant, and one additional micro tick is performed, i. e., $(C, \rho) \xrightarrow{c}_{\mu s} (C', \rho')$. We claim that the successor configuration (C', ρ') satisfies the invariant, too. It suffices to show $th(c_1.node) \parallel th(c_2.node)$ for all distinct active or pausing $c_1, c_2 \in C'$, because no node is concurrent to itself.

So, let $c_1, c_2 \in C'$ be active or pausing nodes. If both $c_1, c_2 \in C$ we have the result by induction hypothesis. If $c_1 \notin C$ and $c_2 \notin C$, so that both configurations have been instantiated by executing c, i. e., $c_1, c_2 \in nxt(c, \rho)$, then $c.node$ must be a fork node and $c_1.node$ and $c_2.node$ the two entry nodes of its concurrent child threads, for which $th(c_1.node) \parallel th(c_2.node)$ holds by definition and $c.node = lcafork(c_1.node, c_2.node)$. Note that neither c_1 nor c_2 can be the join node introduced by executing the fork, because c_1 and c_2 are assumed to be active, whereas the join node would have waiting status.

It remains to deal with the case where exactly one of the two configurations arises from c. Without loss of generality suppose $c_1 \in C$, $c_2 \notin C$ and $c_2 \in nxt(c, \rho)$. We first observe that $c_1 \neq c$. Suppose otherwise, $c = c_1 \in C'$. Then, since $c \neq c_2 \in C'$ and by construction, $C' = C \setminus \{c\} \cup nxt(c, \rho)$, we would have $\{c, c_2\} \subseteq nxt(c, \rho)$. But this means $c.node$ must be a fork node, which is the only node type which generates $nxt(c, \rho)$ with more than one element. But a fork node can never reproduce itself, i. e., we always have $c \notin nxt(c, \rho)$ which is a contradiction. Since $c_1 \neq c$ and both $c, c_1 \in C$, we can use the induction hypothesis on (C, ρ) to infer $th(c_1.node) \parallel th(c.node)$. From this we now show that $th(c_1.node) \parallel th(c_2.node)$ taking into account that c_2 has been generated by executing c. As in the proof of Prop. 1 we proceed by the status $c_2.status$ and statement type $c.node.st$:

- If $c_2.status =$ pausing, then $c.node.st =$ surf then as well as $c.node = c_2.node$. This immediately gives $th(c_1.node) \parallel th(c_2.node)$ from $th(c_1.node) \parallel th(c.node)$.
- If $c_2.status =$ active, then $c_2.node$ cannot be a join node because all join nodes start their life with waiting status. Then, considering all possible cases of a statement c generating an active c_2, we find that $c.node \rightarrow_{seq} c_2.node$. Now, since $th(c_1.node) \parallel th(c.node)$, the definition of thread concurrency and the fact that $c.node \rightarrow_{seq} c_2.node$, where $c_2.node$ is not a join, implies $th(c_1.node) \parallel th(c_2.node)$.

Finally, consider a clock tick $(C, \rho) \rightarrow_{tick} (C', \rho')$, where (C, ρ) must be a quiescent configurations, i. e., contain only pausing or waiting continuations. None of the latter can become active in the clock tick. Instead, the two active continuations $c_1, c_2 \in C'$ must be depth nodes and tick successors of distinct pausing surf nodes $c_1^*, c_2^* \in C$. These must be concurrent, $th(c_1^*.node) \parallel th(c_2^*.node)$, by induction hypothesis. Since $lcafork(c_1^*.node, c_2^*.node) = lcafork(c_1.node, c_2.node)$ we get $th(c_1.node) \parallel th(c_2.node)$. □

Notice that the status waiting is actually superfluous because it is derivable from the subordination relation $\prec$ of the threads. All we really need is the distinction between pausing and {active, waiting}. Nevertheless, we keep the value waiting explicit because it suggests a natural implementation, while the algebraic path order $\prec$ on ancestor paths is more useful for mathematical analysis. It relates the status to the static topological nesting of program threads.

Also, note that in concrete implementations there may be explicit statuses such as {enabled, disabled} associated with each thread to indicate whether it is instantiated or not. These remain

implicit in our modeling. A thread $t \in T$ is *enabled* in C if there is $c \in C$ such that $t = th(c.node)$ and it is *disabled* otherwise.

4.3 Sequentiality vs. concurrency

The key to determinism lies in ruling out any uncertainties due to an unknown scheduling mechanism. Like the synchronous MoC, the SC MoC ensures macro-tick determinism by inducing certain scheduling constraints on variable accesses. Unlike the synchronous MoC, the SC MoC tries to take maximal advantage of the execution order already expressed by the programmer through sequential commands. A scheduler can only affect the order of variable accesses through concurrent threads. As stated in Prop. 2, if variable accesses are already sequentialized by $\rightarrow_{seq}$, they cannot appear simultaneously in the active continuation pool. Hence, there is no way that a thread scheduler can reorder them and thus lead to a non-deterministic outcome. Similarly, a thread is not concurrent with its parent thread. Because of the path ordering $\prec$, a parent thread is always suspended when a child thread is in operation. Thus, it is not up to the scheduler to decide between parent and child thread. There can be no race conditions between variable accesses performed by parent and child threads, and there is no source of non-determinism here.

In every reachable micro configuration (C, ρ) the order of execution of the active continuations is up to the discretion of the scheduler. Hence, non-determinism can occur if the macro tick response, computed during the tick in which (C, ρ) occurs, depends on this ordering. In this case, the program must be rejected. Yet, it is computationally intractable whether a program is deterministic on the macro tick level, even for a given configuration (C, ρ).

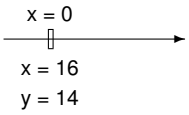
The challenge is to find a suitable restriction on the "free" scheduler which is a) easy to compute, b) leaves sufficient room for concurrent implementations and c) still (predictably) sequentialises any concurrent variable accesses that may conflict and produce unpredictable responses. Note that it is easy to obtain deterministic executions disregarding b): Simply restrict the scheduler to a globally static execution regime, e.g., by assigning each (occurrence) of a program statement a unique execution priority. However, this destroys the natural parallelism of the program.

A simple example for sequential accesses would be s_1: `x = 0`; ...; s_2: `x = 1` (sequential writes). Another example would be `if (x < 0) x = 0` (read followed by write), or `x = f(x)` (another read followed by write). Other cases are more difficult to detect. For instance, in `fork x = 0 par { pause; x = 1 } join`, the accesses to `x` are not concurrent, because `x = 0` would be executed in the first tick, and due to the `pause` statement, `x = 1` would be executed in the second tick. Fig. 8 gives another example of a program `XY` in which the actual run-time concurrency between variable accesses (reading of `y` in L9 and its update in L16) may be prevented by tick separation, depending on the input value `x`.

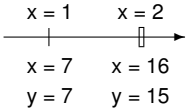
However, it may go beyond the analysis capabilities of a compiler to perform the type of reasoning required in these examples; a conservative approach would be to consider the accesses to be concurrent. For this it is enough to approximate the concurrency and the conflict relations. In Sec. 5.2 we introduce such a restriction, called *SC-admissible* schedules. Before, we need some further terminology, introduced in Sec. 5.1, to characterise potential conflicts in variable assignments.

```
1  module XY
2  input int x;
3  output int y;
4  {
5    fork {
6      // Thread1
7      y = x;
8      y += 2;
9      x = y;
10   }
11   par {
12     // Thread2
13     y += 4;
14     if (y > 6)
15       pause;
16     y += 8;
17   }
18   join;
19   x = 16;
20 }
```

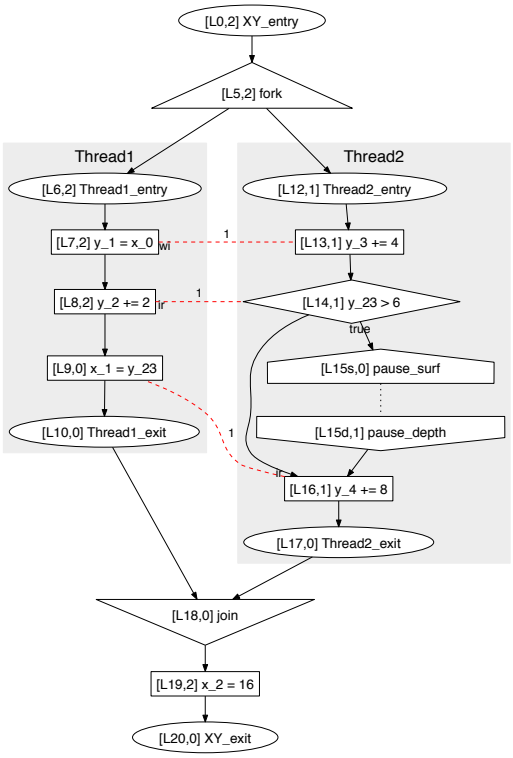
(a) The SCL program



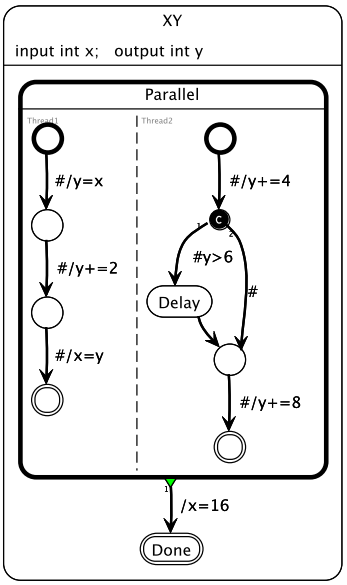
(b) Example trace 1



(c) Example trace 2



(d) The SC Graph



(e) Functionally equivalent SC-Chart

Figure 8: The XY example. In the tick time line, hollow tick markers denote final ticks.

5 Sequential Constructiveness

5.1 Types of writes

In general, concurrent writes to the same variable constitute a race condition that must be avoided. However, there are exceptions to this that we want to permit, again with the goal of not needlessly rejecting sensible, deterministic programs.

For instance, the execution order of any concurrent assignments to *different* variables does not matter. Also, in certain cases, two assignments $x = ex_1$ and $x = ex_2$ to the *same* variable x may be scheduled successively in any order with the same final result. This depends on the semantics of the expressions ex_1 and ex_2 and the memory configurations in which the assignments are evaluated. When the execution order is irrelevant we call such assignments *confluent* in a given configuration, see Def. 7 below. Often, confluence of assignments can be guaranteed globally, i.e., for all reachable configurations. A large class of such assignments are those involving combination functions, defined in the following:

Definition 5 (Combination functions)**.** *A function $f(x, y)$ is a* combination function *(on x) if, for all x and all y_1, y_2, $f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$.*

If f is a combination function, then, by definition, any set of assignments $x = f(x, ex_i)$ in which the expressions ex_i neither produce any side effect nor depend on x, can be executed in arbitrary order yielding a unique final value for x.

Definition 5 is closely related to resolution functions in VHDL, and is a generalized variant of Esterel's combination function (see below). In Esterel, such combination functions must be commutative and associative and are used to deterministically merge concurrent emissions of valued signals, for example via addition. The SC MoC adopts combination functions not only to encompass that functionality, but also to emulate signals with variables, as discussed in Sec. 7. In practice, combination functions are used as "updates" on a variable for which the final value is accumulated incrementally from concurrent source processes.

One may construct arbitrarily complicated assignments from which a compiler might try to extract some combination function. However, to facilitate the compiler's job and to make increments also obvious to the human reader of a program, we recommend to use, whenever possible, patterns of the form `x` f`=` ex, with f being `+`, `*`, etc., and ex being an expression not involving `x`.

A sufficient condition for a combination function is that f is commutative and associative, as demanded in Esterel's combination function, and that ex neither produces any side effect nor depends on `x`. However, commutativity and associativity are not necessary conditions, as "`–`" (subtraction) demonstrates; assignments `x –=` ex are confluent. Indeed, `–` could be replaced by an equivalent commutative, associative combination function, as `x –=` ex is equivalent with `x +=` -ex.

Our notion of sequential constructiveness is based on the idea that the compiler guarantees a strict "initialise-update-read" execution schedule during each macro tick. The initialisation phase is given by the execution of a class of writes which we call *absolute* writes, while the update phase consists of executing *relative* writes. All the *read* accesses, in particular the conditional statements which influence the control flow, are done last. In this way, the compiler restricts the freedom of the run-time platform for reordering variable accesses and creating non-deterministic macro step responses. Although our definitions of sequential admissibility (Def. 10) and sequentially constructive programs (Def. 11) permit an arbitrary separation of writes into absolute and relative writes, it will be important for ASC schedulability (Def. 13) that we can efficiently verify that any two absolute writes and any two relative writes are confluent in all reachable configurations. To this end it is expedient to restrict the notion of relative writes to be combination functions as done in the following definition.

Definition 6 (Absolute/relative writes and reads)**.** *For a combination function f, an assignment $x = f(x, ex)$ where ex does not reference x is a* relative write of type f. *Other assignments are* absolute writes. *A conditional* `if`(ex) *or an assignment $x = ex$ is a* read *for every variable y referenced by ex, unless the assignment is a relative write to y (which also implies $y \neq x$).*

Relative writes of the same type are also confluent.

As the definition of relative writes requires that ex does not reference x, an assignment such as `x *= x - 1` is not considered a relative write, even though `*` is a valid combination function. However, such an assignment might be broken up using a temporary variable, as in `temp = x - 1; x *= temp`, which would be a read followed by a relative write, which then again might be combined with other relative writes of the same type.

In practice, when writing SCL programs, we expect that most writes are absolute writes, and to understand the basics of the SC MoC it suffices to focus on absolute writes and reads and to understand that the former precede the latter for a particular variable. However, the concept of relative writes adds to the expressiveness of sequential constructiveness, in that, as

already mentioned and discussed further in Sec. 7, they allow a straightforward emulation of Esterel-style signals.

5.2 SC-Admissible Scheduling

We are now ready to define what variable accesses we allow in the SC MoC, and what scheduling requirements the accesses induce. The idea is to formulate the requirements that a given program must fulfill to produce a deterministic result, and to accept all programs for which a schedule can be found that meets these requirements.

First, as explained, we want to take advantage of any sequentiality that is already present in the program. If two variable accesses are sequential, we know that they will be executed in this prescribed sequential order. By construction, due to the linear flow of the program text, it is not possible to express conflicting sequential orderings with sequential program statements. We therefore restrict our attention in the following on concurrent variable accesses.

In a nutshell, the order to be imposed by any valid run is, within all ticks R, for all variables v that are accessed concurrently within R, "any confluent absolute writes on v before any confluent relative writes on v before any reads on v." In this fashion, each variable is subjected to a strict sequence of initialisation, incremental update and finally read accesses. Note that these three groups could be interchanged arbitrarily and we would still achieve deterministic concurrency. For example, if reads were to be done before any writes, the reads would not refer to variable values from the current tick, but would always refer to the variable values from the previous tick, or possibly uninitialized values. Also, we could order relative writes before absolute writes, but then the relative writes would be overwritten by the absolute writes. Therefore, we consider the order prescribed above to be the most sensible and intuitive one, as it offers the programmer the greatest degree of control and expressiveness. This is important since this scheduling regime is part of the behavioral semantics of the SC MoC to be implemented by the compiler and/or the target execution architecture. It must be controllable by the programmer at the source level.

Definition 7 (Confluence of Nodes)**.** *Let (C, ρ) be a valid configuration of the SCG. Two nodes $n_1, n_2 \in N$ are called* conflicting *in (C, ρ), if both are active in C, i.e., there exist $c_1, c_2 \in C$ with $c_i.status =$ active, $n_i = c_i.node$, and $c_1(c_2(C, \rho)) \neq c_2(c_1(C, \rho))$. The nodes n_1, n_2 are called* confluent *with each other in (C, ρ), written $n_1 \sim_{(C,\rho)} n_2$, if there is no sequence of micro steps $(C, \rho) \twoheadrightarrow_{\mu s} (C', \rho')$ such that n_1 and n_2 are conflicting in (C', ρ').*

Def. 7 gives a formal account of the notion of confluence which we introduced informally above. Note that confluence is taken *relative* to valid configurations (C, ρ) and *indirectly* as the absence of conflicts. Instead of requiring that confluent nodes commute with each other for *arbitrary* memories we only consider those configurations (C', ρ') that are *reachable* from (C, ρ). For instance, if it happens for a given program that in all memories ρ' reachable from a configuration (C, ρ) two expressions ex_1 and ex_2 evaluate to the same value, then the assignments x = ex_1 and x = ex_2 are confluent in (C, ρ). Similarly, if the two assignments are never jointly active in any reachable continuation pool C', they are confluent in (C, ρ), too. This means that statements may be confluent for some program relative to some reachable configuration, but not for other configurations or in another program. However, notice that relative writes of the same type, according to Def. 6, are confluent in the absolute sense, i.e., for all valid configurations (C, ρ) of all programs.

This relative view of confluence expressed in Def. 7 is useful in order to keep the scheduling constraints on admissible macro ticks, to be defined below in Def. 9, sufficiently weak. Notice that two nodes which are confluent in some configuration are still confluent in every later configuration

reached through an arbitrary sequence of micro steps. Formally, if $(C, \rho) \twoheadrightarrow_{\mu s} (C', \rho')$ and $n_1 \sim_{(C,\rho)} n_2$ then $n_1 \sim_{(C',\rho')} n_2$. However, there may be more nodes confluent in (C', ρ') as compared to (C, ρ), simply because some conflicting configurations reachable from (C, ρ) are no longer reachable from (C', ρ'). We exploit this in the following definition by making confluence of node instances within a macro tick relative to the index position at which they occur.

We could make confluence in Def. 7 even less constraining by taking into account only those conflicts between nodes which can actually be observed by the environment. Specifically, we could consider active configurations c_1, c_2 in conflict if $c_1(c_2(C, \rho)) \not\approx c_2(c_1(C, \rho))$, where $\approx$ is observational equivalence rather than identity. For instance, if c_1 and c_2 are writes to an external log file, which is never read by the program during execution, we could consider them conflict-free and thus confluent, in this sense. On the other hand, note that confluence $n_1 \sim_{(C,\rho)} n_2$ requires conflict-freeness for all configurations (C', ρ') reachable from (C, ρ) by *arbitrary* micro-sequences under *free scheduling*. We will use this notion of confluence to define the restricted set of *SC-admissible* macro ticks (Def. 10). Since the compiler will ensure SC-admissibility of the execution schedule, one might be tempted to define confluence relative to these SC-admissible schedules. However, this is not possible since this would result in a logical cycle.

Definition 8 (Confluence of Node Instances)**.** *Let R be a macro tick and (C_i, ρ_i), for $0 \leq i \leq len(R)$, the configurations of R. Consider two node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$ in R, i. e., $1 \leq i_1, i_2 \leq len(R)$ and $n_1 = R(i_1)$, $n_2 = R(i_2)$. The node instances are called* confluent *in R, written $ni_1 \sim_R ni_2$ if $n_1 \sim_{(C_i,\rho_i)} n_2$, where $i = min(i_1, i_2) - 1$.*

Definition 8 determines confluence of node instances (n_1, i_1) and (n_2, i_2) in a macro tick R relative to the configuration (C_i, ρ_i) in which the first of the two instances is executed. This is the instance with the minimal index $i = min(i_1, i_2) - 1$. It may thus happen that n_1 and n_2 are confluent relative to this configuration (C_i, ρ_i) although they are not confluent in the initial configuration (C_0, ρ_0) of the macro tick. Since the execution sequence from (C_0, ρ_0) to (C_i, ρ_i) will be done under SC-admissibility constraints, the range of configurations in a tick in which confluence of given node instances becomes critical may be drastically reduced. This is important since whenever two concurrent nodes are not confluent their execution order must be fixed to prevent non-determinism. To this end the concurrency relation $|_R$ is now refined by the following scheduling relations on node instances to characterise potentially conflicting variables accesses:

Definition 9 (Scheduling Relation on Node Instances)**.** *Let R be a macro tick with two node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$, i. e., $1 \leq i_1, i_2 \leq len(R)$ and $n_1 = R(i_1)$, $n_2 = R(i_2)$. Suppose further that ni_1 and ni_2 are concurrent in R, i. e., $ni_1 |_R ni_2$, but not confluent in R, i. e., $ni_1 \not\sim_R ni_2$. Under these conditions, we write*

- *$ni_1 \rightarrow^R_{ww} ni_2$ iff there exists a variable on which ni_1 and ni_2 both perform absolute writes, or both perform relative writes of different type.*
- *$ni_1 \rightarrow^R_{wr} ni_2$ iff ni_1 performs an absolute write to a variable that is read by ni_2.*
- *$ni_1 \rightarrow^R_{ir} ni_2$ iff ni_1 performs a relative write to a variable that is read by ni_2 (the subscript i stands for "increment").*
- *$ni_1 \rightarrow^R_{wi} ni_2$ iff ni_1 performs an absolute write to a variable on which ni_2 performs a relative write.*
- *$ni_1 \rightarrow^R ni_2$ if $i_1 < i_2$, i. e., ni_1 occurs before ni_2 in R.*

Notice that the relation $\rightarrow^R_{ww}$ in Def. 9 is symmetric, i.e., $ni_1 \rightarrow^R_{ww} ni_2$ implies $ni_2 \rightarrow^R_{ww} ni_1$. We abbreviate the conjunction of $ni_1 \rightarrow^R_{ww} ni_2$ and $ni_2 \rightarrow^R_{ww} ni_1$ with $ni_1 \leftrightarrow^R_{ww} ni_2$, and due to the aforementioned symmetry, $\rightarrow^R_{ww}$ implies $\leftrightarrow^R_{ww}$.

By ensuring that execution order respects the ordering constraints $\rightarrow_\alpha$, $\alpha \in \{ww, wr, ir, wi\}$, we can now implement the "initialise-update-read" protocol on variable accesses described above.

Definition 10 (SC-Admissibility)**.** *A macro tick R is* SC admissible *if for all node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$ in R, with $1 \leq i_1, i_2 \leq len(R)$ and $n_{1,2} = R(i_{1,2})$, the following* SC scheduling conditions *are satisfied:*

SC1 $ni_1 \leftrightarrow^R_{ww} ni_2$ *does not hold* [3]*;*

SC2 If $ni_1 \rightarrow^R_{wr} ni_2$ then $ni_1 \rightarrow^R ni_2$, i. e., whenever ni_1 performs an absolute write and ni_2 is a non-confluent concurrent read of the same variable, then the write happens before the read;

SC3 If $ni_1 \rightarrow^R_{ir} ni_2$ then $ni_1 \rightarrow^R ni_2$, i. e., whenever ni_1 performs a relative write and ni_2 is a non-confluent concurrent read of the same variable, then the write happens before the read;

SC4 If $ni_1 \rightarrow^R_{wi} ni_2$ then $ni_1 \rightarrow^R ni_2$, i. e., whenever ni_1 performs an absolute write and ni_2 a non-confluent concurrent relative write on the same variable, then ni_1 happens before ni_2.

A run for an SCG is SC admissible *if all macro ticks R in this run are SC admissible.*

In practice, relative writes to a particular variable tend to be of the same type and thus automatically confluent with each other. Further, typical programs will be such that if there is more than one concurrent absolute write to a variable, then all are identical, i.e., they write the same value. An example is Esterel, where all absolute writes are signal emissions setting a signal variable to true. Hence, these are also confluent in any configuration.

Let us observe that confluence (which expresses a second-order property of nodes relative to a *set* of executions) cannot be replaced by the (first-order) condition that out-of-order write accesses be *ineffective*, i.e., do not change the configurations. For example, consider the program InEffective1 in Fig. 9a which exhibits ineffective out-of-order schedules which produce non-deterministic responses. The first possible macro tick execution is L10 : x=7; L5 : if (x==2); L8 : y=0 which does not contain any out-of-order accesses. The absolute write to variable x in L10 happens before the concurrent read L5. This execution gives the response x=7, y=0. A different response x=7, y=1 is obtained from the macro tick sequence L5 : if (x==2); L6 : y=1; x=7; L10 : x=7. Here, the second absolute write L10 is out-of-order and concurrent with the earlier read L4. However, it is ineffective because x gets assigned the very same value, viz. 7, by statement L6 before. While ineffectiveness would not exclude this sequencing, our notion of confluence does. The write L10 is not confluent with the earlier read L5 since the order of execution matters in the configuration in which L5 and L10 are both active. Hence the second execution is not admissible according to Def. 10.

The notion of "effectiveness" does not become stronger if we determine effectiveness of L10 relative to the read L5. Consider the modification InEffective2 of the program seen in Fig. 9b

[3] Alternatively, SC1 could be formulated "$ni_1 \rightarrow^R_{ww} ni_2$ must imply $ni_1 \rightarrow^R ni_2$," which would be more in line with the formulations of SC2 – SC3. However, due to symmetry of $\rightarrow_{ww}$, that implication can never be satisfied. Thus the shortened, more direct formulation of SC1, which corresponds to the intuitive requirement that there must not be any concurrent write-write conflicts.

```
1  module InEffective1
2  output int x = 2; int y;
3  {
4    fork
5      if (x == 2)
6        { y = 1; x = 7 }
7      else
8          y = 0
9    par
10     x = 7
11   join
12 }
```

(a) An ineffective absolute write x = 7 at write point.

```
1  module InEffective2
2  output bool x = false; int y;
3  {
4   fork
5     if (!x)
6       { y = 1; x = x xor true }
7     else
8         y = 0
9   par
10    x = x xor true;
11  join
12 }
```

(b) An ineffective relative write x = x xor true wrt to the read if (!x).

Figure 9: The InEffective examples illustrating non-deterministic responses despite ineffective out-of-order write accesses

where the statements x = x xor 1, equivalent to x = !x, are considered relative writes. Now, in the execution sequence L5; L6; L10 the out-of-order write L10 is ineffective relative to the configuration of the read L5. The write L10 : x = x xor true after L6 assigns x=false which is the very same value that x had at the point of the read L5. This sequence would still be admissible and we would have the two different program responses x=true, y=0 and x=false, y=1. In contrast, under Def. 10, the tick producing x=false, y=1 is not admissible because L10 is not confluent with the read L5.

The notion of SC-admissibility (Def. 10) restricts the "free" scheduling defined in Sec. 4.2 to those executions which respect an "initialise-update-read" regime for concurrent variable accesses unless these variable accesses are confluent, in which case the order is immaterial. We assume that this regime is enforced by the compiler and/or the run-time system on the target architecture. A program is considered *sequentially constructive* if it exhibits a deterministic behavior under such SC-admissible scheduling.

Definition 11 (Sequential Constructiveness)**.** *A program is* sequentially constructive *(SC) if (i) there exists an SC-admissible run for it, and moreover (ii) every SC-admissible run generates the same, deterministic trace of macro ticks.*

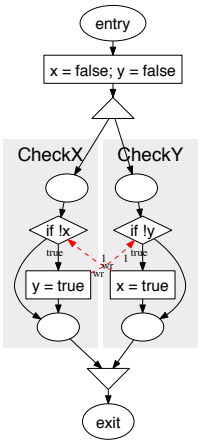
The mere existence of an SC-admissible run does not yet guarantee determinism. A counter example is the program NonDet presented in Fig. 10, which has two sequentially admissible runs in which the threads CheckX and CheckY (conditionals in L5 and L7) are executed atomically. Depending on which thread is scheduled first, we end up with the memory $[\mathsf{x} = \mathsf{true}, \mathsf{y} = \mathsf{false}]$ or $[\mathsf{x} = \mathsf{false}, \mathsf{y} = \mathsf{true}]$. These SC-admissible runs are non-deterministic and thus NonDet is not SC.

Another way in which sequential constructiveness can be violated is when there are no SC-admissible schedules at all. The simplest SC-blocking program is Fail seen in Fig. 11. No execution of Fail under the "free" scheduling is SC-admissible. One of the assignments L6 or L9 will happen after variable z has been read as false by the conditional L8 or L5 in the other thread. Notice that all macro tick runs of Fail produce the same result, i.e., the program is deterministic under non-SC scheduling. Such programs have rather theoretical interest, as they certainly don't represent good coding style, and fall outside of the class considered here.

As a positive example, the program Control, shown in Fig. 2, is sequentially constructive. As the concurrency relation on variable accesses is not transitive, an access may belong to different (maximal) sets of mutually concurrent accesses. For example, in Control, L14 belongs to two

```
1  module NonDet
2  output bool x = false, y = false;
3  {
4    fork    // Thread "CheckX"
5      if (!x)
6        y = true;
7    par     // Thread "CheckY"
8      if (!y)
9        x = true;
10   join;
11 }
```

(a) The SCL program



(b) The SC Graph

Figure 10: The NonDet example, illustrating multiple admissible runs and non-deterministic outcome

```
1  module Fail
2  output bool z = false;
3  {
4   fork
5       if (!z)
6       then z = true
7   par
8       if (z)
9       then z = true
10  join
11 }
```

Figure 11: The SCL program Fail, which has a deterministic outcome, but no SC-admissible schedule

maximal sets of concurrent accesses, namely $\{\mathsf{L14}, \mathsf{L22}\}$ and $\{\mathsf{L14}, \mathsf{L24}\}$. The SC scheduling rule (Def. 10), applied to each of these sets, demands that L22 and L24 must both be scheduled before L14. This, together with the natural sequential ordering of the Dispatch thread, results in a deterministic outcome. Formally, this follows from the fact that Control is ASC-schedulable and every ASC-schedulable program sequentially constructive (see Thm. 1 below).

6 Analyzing Sequential Constructiveness

Practical analyses must approximate the notion of sequential constructiveness which is computationally intractable due to its dependence on run-time properties of macro ticks and node *instances*. To this end we abstract the concurrency and scheduling relations from node instances to static relations on nodes.

Definition 12 (Valid SC-Schedule.)**.** *Let* $\Sigma = (\rightarrow_\alpha \mid \alpha \in \{\mathrm{ww}, \mathrm{wr}, \mathrm{wi}, \mathrm{ir}\})$ *be a system of binary relations* $\rightarrow_\alpha$ *on the statement nodes of* G*.* Σ *is called a* valid SC-schedule *if it (i) is compatible with the static ordering constraints imposed by the "initialise-update-read protocol" and (ii) conservatively over-approximates the dynamic ordering relations necessary for SC-admissible*

runs. Formally, for all $\alpha \in \{\mathrm{ww}, \mathrm{wr}, \mathrm{wi}, \mathrm{ir}\}$ *the following holds:*

- *(Soundness) If* $n_1 \to_\alpha n_2$*, then* n_1 *and* n_2 *are statically concurrent, i.e.,* $th(n_1) \mid\mid th(n_2)$*. Furthermore, if* $\alpha = \mathrm{ww}$*, then* n_1*,* n_2 *are absolute writes to the same variable; if* $\alpha = \mathrm{wr}$*, then* n_1 *is an absolute write and* n_2 *a read to the same variable; if* $\alpha = \mathrm{wi}$*, then* n_1 *is an absolute write and* n_2 *a relative write to the same variable; if* $\alpha = \mathrm{ir}$*, then* n_1 *is a relative write and* n_2 *a read to the same variable.*
- *(Completeness) For every macro tick* R *of* G *which is reached and executed under the SC-admissibility rules, if* $(n_1, i_1) \to_\alpha^R (n_2, i_2)$ *for node instances* $n_1 = R(i_1)$*,* $n_2 = R(i_2)$ *with* $i_1, i_2 \leq len(R)$*, then* $n_1 \to_\alpha n_2$*.*

For a given SC-schedule, we also use the following abbreviations:

- $n_1 \to_{wir} n_2$ *iff* $n_1 \to_{ww} n_2$ *or* $n_1 \to_{wr} n_2$ *or* $n_1 \to_{wi} n_2$ *or* $n_1 \to_{ir} n_2$*. This summarizes the constraints induced by concurrent write/increment/read accesses.*
- $n_1 \to n_2$ *iff* $n_1 \to_{seq} n_2$ *or* $n_1 \to_{wir} n_2$*, that is, if there is any control-flow or concurrent-access-induced ordering constraint.*

The second condition of Def. 12 on SC-schedules Σ is the crucial validity requirement. It guarantees that the static node relations $\to_\alpha$ of Σ are a sound over-approximation of the dynamic relations $\to_\alpha^R$ on node instances under the assumption that G is executed in an SC-admissible fashion. In contrast, the first condition of Def. 12 imposes an upper bound on the scheduling relations to prevent the schedule from sequentialising random nodes of the program. Without this restriction Σ could force, e.g., the sequentialisation of concurrent accesses to *different* variables, or of reads before writes. This is not what we want. The purpose of Def. 12 is to provide a convenient criterion to decide SC-constructiveness of programs. Of course, this does not prevent a compiler from adding further ordering constraints between arbitrary program nodes on top of a given valid SC-schedule Σ to achieve a particular deterministic implementation on a particular execution platform.

Definition 13 (ASC Schedulability)**.** *A program is* acyclic SC (ASC) schedulable *if there exists a valid SC schedule such that there is no* $\to$ *cycle that contains edges induced by* $\to_{wir}$*.*

Theorem 1 (Sequential Constructiveness)**.** *Every ASC schedulable program is sequentially constructive.*

Proof. Very roughly, the result follows from examining the constraints required by ASC schedulability and the SC-admissibility rules underlying the definition of SC, plus the observation that under SC scheduling, all *potential* writes are scheduled before any reads, thus ensuring determinism. The formal proof is somewhat more involved.

For simplicity, we identify continuations with the active nodes they carry, i. e., we write $n \in C$, if there is $c \in C$ such that $c.status = \mathsf{active}$ and $c.node = n$. This is justified because in the execution of SCGs no node can be instantiated more than once in different continuations of the same continuation pool.

First, observe that the final configuration reached through a sequence of micro steps is uniquely determined by the sequence of nodes executed. Formally, if

$$(C, \rho) \overset{R}{\twoheadrightarrow}_{\mu s} (C', \rho') \text{ and } (C, \rho) \overset{R}{\twoheadrightarrow}_{\mu s} (C'', \rho'')$$

then $C' = C''$ and $\rho' = \rho''$. This is because the successor configuration is uniquely determined by the choice of the continuation executed in each micro step, and because no node can be instantiated more than once in different continuations of the same continuation pool. This functional dependency permits us to write

$$(C', \rho') = R(C, \rho) \text{ instead of } (C, \rho) \overset{R}{\twoheadrightarrow}_{\mu s} (C', \rho').$$

It implies that any non-determinism in macro steps must result entirely from the order in which the active configurations are selected in the sequence of micro steps.

Let $SC(C_0, \rho_0)$ be the set of all SC-admissible macro ticks (maximal sequences of SC-admissible micro steps) starting in (C_0, ρ_0) and $\rightarrow_\alpha$ for $\alpha \in \{ww, wr, wi, ir\}$) be a valid and acyclic SC-schedule of G according to Def. 12.

We first argue SC-schedulability, i. e., that $SC(C_0, \rho_0)$ is non-empty for every reachable configuration (C_0, ρ_0). Since there is no $\rightarrow$ cycle on nodes containing $\rightarrow_{wir}$ edges we can sequentially order the nodes in the SCG consistently with $\rightarrow$ and associate with every node a priority $n.pr \in \mathbb{N}$ such that if $n_1 \rightarrow_{wir} n_2$ then $n_1.pr > n_2.pr$, and if $n_1 \rightarrow_{seq} n_2$ then $n_1.pr \geq n_2.pr$ (see also Sec. 6.2). It is straightforward to show that executing nodes according to their priority enforces SC-admissibility, i. e., that every macro tick R starting from (C_0, ρ_0) such that $R(i).pr \geq R(j).pr$ for all $1 \leq i < j \leq k$ is SC-admissible.

Firstly, R cannot contain concurrent node instances $(R(i_1), i_1) \rightarrow^R_{ww} (R(i_2), i_2)$ that perform non-confluent absolute or non-confluent relative writes to the same variable. Otherwise, we would have a cycle $R(i_1) \leftrightarrow_{ww} R(i_2)$ by symmetry of $\rightarrow^R_{ww}$ and validity of the schedule. But this is excluded by ASC schedulability. This proves requirement SC1 of SC-admissibility. For requirement SC2 and SC3 suppose R contains concurrent node instances $(R(i_2), i_2) \rightarrow^R_{ir} (R(i_1), i_1)$ or $(R(i_2), i_2) \rightarrow^R_{wr} (R(i_1), i_1)$, with $i_1 < i_2$, i. e., $R(i_1)$ is a read and $R(i_2)$ a subsequent concurrent, non-confluent write to the same variable. Then, $R(i_2) \rightarrow_{ir} R(i_1)$ or $R(i_2) \rightarrow_{wr} R(i_1)$ by validity, which means $R(i_2) \rightarrow_{wir} R(i_1)$. But then $R(i_2).pr > R(i_1).pr$ which contradicts the construction of R which enforces $R(i_1).pr \geq R(i_2).pr$. The condition SC4 of SC-admissibility is argued analogously. Its violation would give nodes $R(i_2) \rightarrow_{wi} R(i_1)$ for $i_1 < i_2$ which contradicts the execution priorities.

To sum up, we can generate a SC-admissible macro step from initial configuration (C_0, ρ_0) simply by executing the nodes in order of non-increasing priorities. This, in turn, can be achieved by always selecting among all the active nodes one with maximal priority. Consider two consecutive steps in a micro step sequence

$$\cdots (C_i, \rho_i) \overset{n_{i+1}}{\rightarrow}_{\mu s} (C_{i+1}, \rho_{i+1}) \overset{n_{i+2}}{\rightarrow}_{\mu s} (C_{i+2}, \rho_{i+2}) \cdots$$

such that n_{i+1} and n_{i+2} each have maximal priority among all active configurations in C_i and C_{i+1}, respectively. Then, if n_{i+2} was already contained in C_i we have $n_{i+1}.pr \geq n_{i+2}.pr$ by construction. If n_{i+2} is not in C_i but has entered C_{i+1} as a result of executing n_{i+1}, then the statement node of n_{i+2} must be a sequential successor of that of n_{i+1}, i. e., $n_{i+1} \rightarrow_{seq} n_{i+2}$. But this means $n_{i+1}.pr \geq n_{i+2}.pr$ by the assignment of priorities.

Since in every configuration (with finite continuation pool) there must be a continuation with maximal node priority, we have shown that $SC(C_0, \rho_0)$ is non-empty for any reachable configuration (C_0, ρ_0).

Next we deal with determinism. We prove that if $SC(C_0, \rho_0)$ contains at least one finite execution, then all executions in $SC(C_0, \rho_0)$ are finite and result in the same quiescent configuration. Let $len(C_0, \rho_0) \geq 0$ be the minimal length of any finite and maximal micro step sequence

R starting in (C_0, ρ_0). If this number does not exist, $len(C_0, \rho_0) = \infty$, all micro sequences from (C_0, ρ_0) are infinite and we are done. We prove by induction on $len(C_0, \rho_0) < \infty$ that any two SC-admissible macro ticks $R, R' \in SC(C_0, \rho_0)$ are finite

$$(C_0, \rho_0) \stackrel{R}{\Longrightarrow} (C_k, \rho_k) \text{ and } (C_0, \rho_0) \stackrel{R'}{\Longrightarrow} (C'_{k'}, \rho'_{k'})$$

and we necessarily have $C_k = C'_{k'}$ and $\rho_k = \rho'_{k'}$.

If $len(C_0, \rho_0) = 0$, then no micro step is needed to complete a macro tick from C_0, i. e., C_0 contains no active continuation and thus is quiescent. Hence, no micro step is possible at all and thus trivially $C_k = C_0 = C'_{k'}$ as well as $\rho_k = \rho_0 = \rho'_{k'}$.

For the inductive case, $len(C_0, \rho_0) \geq 1$, which means C_0 contains at least one active continuation and every $R \in SC(C_0, \rho_0)$ must necessarily be of length $len(R) \geq 1$. Note that SC-admissibility is closed under suffixes: Whenever

$$(C_0, \rho_0) \stackrel{n}{\rightarrow}_{\mu s} (C_1, \rho_1) \stackrel{R}{\twoheadrightarrow}_{\mu s} (C', \rho')$$

is SC-admissible, then R is SC-admissible, too. This follows from the fact that any pair of concurrent or conflicting node (instances) in R are also concurrent or conflicting in n, R.

For every $n \in C_0$, let $Dep(n) \subseteq C_0$ be the set of (active) nodes $n^* \in C_0$ such that there is a macro tick $R' \in SC(C_0, \rho_0)$ containing n^* and a node m' such that $n^* \twoheadrightarrow_{seq} m'$ and $m' \rightarrow_{wir} n$. This includes the special case that $n^* = m'$, i. e., $n^* \rightarrow_{wir} n$. In general, whenever $n^* \in Dep(n)$ then $n^* \rightarrow n$ includes exactly one $\rightarrow_{wir}$ dependency. By ASC-schedulability (Def. 13), there must be at least one $n \in C_0$ such that $Dep(n) = \emptyset$, for otherwise finiteness of C_0 would imply the existence of a $\rightarrow$ cycle that traverses at least one $\rightarrow_{wir}$ edge.

As argued above, every node $n \in C_0$ is the start of an SC-admissible sequence, i. e., $SC(n(C_0, \rho_0)) \neq \emptyset$. Therefore, there must exist an SC-admissible macro tick $R = n, R_1$ with

$$(C_0, \rho_0) \stackrel{n}{\rightarrow}_{\mu s} (C_1, \rho_1) \stackrel{R_1}{\Longrightarrow} \cdots \tag{3}$$

so that $Dep(n) = \emptyset$. We claim that every quiescent configuration reachable from (C_0, ρ_0) through a finite SC-admissible macro tick in d micro steps is also reachable from (C_1, ρ_1) in an SC-admissible sequence of $d - 1$ micro steps. By induction from (C_1, ρ_1) the result follows easily since then $len(C_1, \rho_1) = len(C_0, \rho_0) - 1 < \infty$ and any finitely non-determinism reachable from configuration (C_0, ρ_0) would generate the same non-deterministic responses from (C_1, ρ_1).

To prove the reduction claim, let $R' = n', R'_1$

$$(C_0, \rho_0) \stackrel{n'}{\rightarrow}_{\mu s} (C'_1, \rho'_1) \stackrel{R'_1}{\Longrightarrow} (C'_{k'}, \rho'_{k'}) \tag{4}$$

be any *finite* SC-admissible macro tick of length $d = len(R')$ in $SC(C_0, \rho_0)$ ending in a quiescent configuration $(C'_{k'}, \rho'_{k'})$. If $n = n'$ then $(C_1, \rho_1) = (C'_1, \rho'_1)$ and the claim follows directly. Hence, we may assume $n' \neq n$ in (3) and (4).

Since active nodes remain active as long as they are not executed, n must be active in C'_1. Further, n must eventually be scheduled in R'_1 because the final quiescent configuration $(C'_{k'}, \rho'_{k'})$ does not contain any more active continuations. Hence, $R'_1 = R'_{11}, n, R'_{12}$. Letting $j' = len(R'_{11})$, the macro tick (4) must break up like this:

$$\begin{aligned}
&(C_0, \rho_0) \stackrel{n'}{\rightarrow}_{\mu s} (C'_1, \rho'_1) \stackrel{R'_{11}}{\twoheadrightarrow}_{\mu s} (C'_{j'+1}, \rho'_{j'+1}) \\
&(C'_{j'+1}, \rho'_{j'+1}) \stackrel{n}{\rightarrow}_{\mu s} (C'_{j'+2}, \rho'_{j'+2}) \stackrel{R'_{12}}{\Longrightarrow} (C'_{k'}, \rho'_{k'}),
\end{aligned}$$

where n does not occur in R'_{11} but $n \in C'_i$ for all $1 \leq i \leq j' + 1$.

Moreover, observe that since both nodes n, n' are simultaneously active in C_0, they must be concurrent node instances in R', i. e., $(n', 1) \mid_{R'} (n, j' + 2)$. More generally, $(n, j' + 2)$ is concurrent to every node instance m' in n', R'_{11}. This follows from Prop. 2. This means that the scheduling constraints SC1–SC4 expressed in Def. 10 apply, which restricts the possible statement types appearing in n', R'_{11}. A consequence of this restriction is, as we shall show, that the initial sequence n', R'_{11} must commute with n in R. This implies that

$$(C_1, \rho_1) \overset{n', R'_{11}, R'_{12}}{\Longrightarrow} (C'_{k'}, \rho'_{k'}) \tag{5}$$

is a macro tick (maximal micro step sequence) from initial configuration (C_1, ρ_1). One can show, moreover, that (5) is SC-admissible: Any pair of node instances concurrent in n', R'_{11}, R'_{12} must have been concurrent in the SC-admissible macro tick $R' = n', R'_{11}, n, R'_{12}$ (considering that n is concurrent with all n', R'_{11}), and every pair of nodes from n', R'_{11}, R'_{12} which are non-confluent in (C_1, ρ_1) are also non-confluent in (C_0, ρ_0) by (3). Therefore, since $len(n', R'_{11}, R'_{12}) = len(R'_1) = len(R') - 1 = d - 1$ the claim follows from (5).

The core of our argument for (5) is the confluence of node executions. Recall (Def. 7) that two nodes $n_1, n_2 \in D$ are confluent in a configuration (D, σ), written $n_1 \sim_{(D,\sigma)} n_2$, if in any configuration reachable from (D, σ), in which both nodes are active, the order of executing n_1 or n_2 has no influence on the resulting configuration. We generalise this slightly as follows. Let $R = m_1, m_2, \ldots, m_k$ be a sequence of micro ticks executable from initial configuration (D, σ) and $R_i = m_1, m_2, \ldots, m_{i-1}$, for $1 \leq i \leq k$, its prefix sequences. This includes the degenerated case R_1 which is the empty sequence. A node n is *confluent* with respect to R, written $n \sim_{(D,\sigma)} R$ if $n \sim_{R_i(D_0,\sigma_0)} m_i$ for all $1 \leq i \leq k$. For $i = 1$ this is the same as $n \sim_{(D_0,\sigma_0)} m_1$. One can show that if $n(D, \sigma) = (D_1, \sigma_1)$ and $R(D, \sigma) = (D_2, \sigma_2)$, then $n \sim_{(D,\sigma)} R$ implies there is (D', ρ') with $R(D_1, \sigma_1) = (D', \sigma') = n(D_2, \sigma_2)$. Finally, we write $n \sim R$ if $n \sim_{(D,\sigma)} R$ for all valid configurations (D, σ).

We claim that $n \sim_{(C_0,\rho_0)} n', R'_{11}$, which gives us (5). First, observe that if the node n is of one of the statement types exit, entry, goto, fork, join, surf or depth, then it is confluent with any other active node, in particular those of the sequence n', R'_{11}. This holds trivially since these statements neither depend on nor change the memory, and their effect is solely determined by the static structure of the program graph.

It remains to consider the cases where the node n is an assignment or a conditional:

- Suppose n is a conditional if (ex) and no variable referred in ex is written by any node in n', R'_{11}. Since executing a conditional does not change the memory, this means $n \sim m$ for all nodes m in n', R'_{11}, and thus $n \sim n', R'_{11}$, regardless the specific configurations (C'_{i-1}, ρ'_{i-1}) generated during the execution of R'.

  The only interesting sub-case is when a variable referred in ex (and thus read by n) is written by any node instance (m', i) in n', R'_{11} with $1 \leq i \leq len(n', R'_{11}) = j' + 1$. If confluence is violated, i.e., $n \not\sim_{(C'_{i-1},\rho'_{i-1})} m'$, then we have $m' \rightarrow^{R'}_{wr} n$ or $m' \rightarrow^{R'}_{ir} n$ by Def. 9 which in turn implies $m' \rightarrow_{wir} n$ by validity of the SC-schedule Def. 12. We will show below that this can be excluded, whence $n \not\sim_{(C'_{i-1},\rho'_{i-1})} m'$ and thus $n \sim n', R'_{11}$ in all the sub-cases.

- Suppose n is an assignment x = ex and no variable referred in ex is written by any node m' in n', R'_{11}, unless this variable is x and m' is a relative write to x or n an absolute write

to x. We show from the confluence conditions of SC-admissibility of R' that $n \sim_{(C_0,\rho_0)} m'$ for each node m' in n', R'_{11}. This implies $n \sim_{(C_0,\rho_0)} n', R'_{11}$ as desired.

Let us first look at the possibilities of a node instance (m', i), for $1 \leq i \leq len(n', R'_{11}) = j' + 1$, in n', R'_{11} that refers to x:

- If m' is a read on x, then the claim follows directly from SC-admissibility of R' (Def. 10): Considering that $(m', i) \mid_{R'} (n, j' + 2)$ we cannot have $n \not\sim_{(C'_{i-1},\rho'_{i-1})} m'$ because this would imply $(n, j' + 2) \rightarrow^{R'}_{wr} (m', i)$ or $(n, j' + 2) \rightarrow^{R'}_{ir} (m', i)$ by Defs. 8 and 9 depending on whether n is an absolute write or a relative write of x. Yet, the former is excluded by SC2 and the latter by SC3.
- If m' is a relative write of x, for the same reason, we have $n \sim_{(C'_{i-1},\rho'_{i-1})} m'$ because of conditions SC1 or SC4 of SC-admissibility, which enforce confluence of all concurrent writes after any relative write in R'.

Every other node m' in n', R'_{11} which is neither a read of x nor a relative write on x, cannot refer to (evaluate) variable x at all, by Def. 6. Thus, every node instance (m', i) of n', R'_{11} which refers to x is confluent to n in (C'_{i-1}, ρ'_{i-1}) by the analysis above. Any node instance (m', i) which does not refer to x is trivially confluent with n in any configuration unless it is an assignment which changes a variable referred to by ex. However, by assumption, m' is a relative write to x or both m' and n are absolute writes to x. In all cases, we infer $n \sim_{(C'_{i-1},\rho'_{i-1})} m'$ by the properties SC1 or SC4 for SC-admissible R'.

What about the sub-cases of nodes m' in n', R'_{11} not covered? It can be shown that these must also satisfy $n \sim_{(C'_{i-1},\rho'_{i-1})} m'$. Otherwise, if a node m' writes to a variable occurring in ex, but different from x, *and* $n \not\sim_{(C'_{i-1},\rho'_{i-1})} m'$, then $m' \rightarrow^{R'}_{ir} n$ or $m' \rightarrow^{R'}_{wr} n$; If m' is an absolute write to x and n a relative write to x, *and* $n \not\sim_{(C'_{i-1},\rho'_{i-1})} m'$, then $m' \rightarrow^{R'}_{wi} n$. Validity of the SC-scheduling relations implies $m' \rightarrow_{wir} n$ in all these sub-cases, which is excluded by the argument given below.

Finally, as indicated, let us show that the case analysis above is complete, i. e., there cannot exist a node m' in n', R'_{11} with $m' \rightarrow_{wir} n$. By Prop. 1 such m' must be the sequential successor $m^* \twoheadrightarrow_{seq} m'$ of some node $m^* \in C_0$. But then $m^* \in Dep(n)$ which contradicts our assumption that $Dep(n) = \emptyset$. This completes the proof of (5) and thus of Thm. 1.

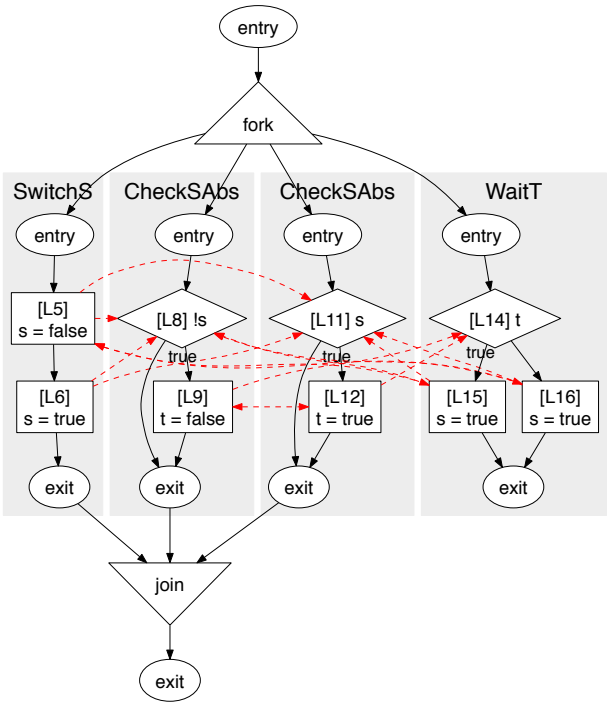
□

The Control example of Fig. 2 is ASC schedulable and thus sequentially constructive. Specifically, it is easy to see that the only pairs of nodes which are concurrent and in conflict relative to any initial configuration of Control are L24 $\not\sim$ L14 and L22 $\not\sim$ L14 which are read-write conflicts on variable grant, and L23 $\not\sim$ L13 as a read-write conflict on checkReq. Observe that we treat the assignments L22, L23 and L24 as absolute writes. Hence, by forcing the execution to respect the orderings L22 $\rightarrow_{wr}$ L14, L13 $\rightarrow_{wr}$ L23 and L24 $\rightarrow_{wr}$ L14, specified by the red arrows in Fig. 2, we avoid the only possible scheduling violation (SC1) expressed in Def. 10. Since these constraints, when added to the program order, do not introduce a causality cycle, we have a valid acyclic SC schedule in the sense of Def. 13. This not only ensures SC-admissible execution but also, by Thm. 1, deterministic macro step responses.

Slightly more subtle is the analysis of the sequentially constructive program GuardedA depicted in Fig. 12a. It illustrates the dynamic nature of our SC scheduling relations of Def. 9

```
1  module GuardedA
2  output bool s, t;
3  {
4    fork
   // Thread "SwitchS"
5      s = false;
6      s = true;
7    par
   // Thread "CheckSAbs"
8      if (!s)
9        t = false;
10   par
   // Thread "CheckSPres"
11     if (s)
12       t = true;
13   par
   // Thread "WaitT"
14     if (t)
15       then s = true;
16       else s = true;
17   join;
18 }
```

(a) The SCL program



(b) The SC Graph

Figure 12: The GuardedA example, illustrating the data-dependent nature of SC-constructiveness

which need to be over-approximated to find a valid SC schedule, according to Def. 12. We begin by classifying writes into relative and absolute writes. This is not necessarily unique. The most conservative stand is to consider all the assignments L5, L6, L9, L12, L15, L16 as absolute writes. Then, looking at the program from an arbitrary initial configuration and under free scheduling we could treat all of the potential conflicts by the following valid SC schedule Σ_0 (Def. 12):

C1 $w \rightarrow_{wr} r$, for $w \in \{\mathsf{L5}, \mathsf{L6}, \mathsf{L15}, \mathsf{L16}\}$ and $r \in \{\mathsf{L8}, \mathsf{L11}\}$, which resolves read-write conflicts on variable s;

C2 L12 $\rightarrow_{wr}$ L14 and L9 $\rightarrow_{wr}$ L14 to deal with read-write conflicts on variable t;

C3 L9 $\leftrightarrow_{ww}$ L12 for the write-write conflict on variable t;

C4 L5 $\leftrightarrow_{ww}$ L15 and L5 $\leftrightarrow_{ww}$ L16 to treat write-write conflicts on variable s.

Observe that the pairs $\{\mathsf{L6}, \mathsf{L15}\}$ and $\{\mathsf{L6}, \mathsf{L16}\}$ are not conflicting because they are identical writes, and thus confluent in all configurations. The assignments L5 and L6 are not in conflict because they are sequentially ordered and hence not concurrent. The same applies to L15 and L16.

All the conflicts C1–C4 can indeed be generated, in the sense of Def. 7, by free scheduling of GuardedA from a suitable initial configuration. For instance, we could execute the sequence of statements L4, L5, L8, L6, L11 to reach a configuration in which both assignments L9 and L12 are jointly active. Similarly, in an initial configuration in which s = 1 we can run through L4, L11, L12, L14 and reach a configuration in which both L15 and L5 are active.

SC-admissible schedules must make sure that program nodes are never executed against these orderings, unless they are confluent in the local configuration. Running GuardedA under

the static SC-schedule Σ_0 given by C1–C4 would achieve that. However, this schedule is of no use since it creates a causality cycle. In fact, every run that adheres to C1 and C2 will block: C1 forces L15 and L16 to be executed before both L8 and L11. By program order this can only happen after L14 is taken, which in turn must wait for L9 and L12, by C2. Since both L9 and L12 depend on prior execution of L8 and L11, by sequential ordering, we are locked up in a cycle. This is in contrast to the program Control, where the static orderings valid for unconstrained free executions already generate an acyclic SC-schedule in the sense of Def. 13.

For GuardedA the conservative "worst-case" over-approximation C1–C4 is too tight a belt. Instead, to see that GuardedA is ASC-schedulable, we can exploit the fact that validity of SC-schedules in Def. 12 is not required for arbitrary free executions but only those that are SC-admissible. Because of this, so it turns out, a subset of the constraints C1–C4 is already sufficient to ensure SC-admissibility. For instance, since L6 is sequentially after L5 in program order, $\mathsf{L6} \rightarrow_{wr} r$ suffices to enforce $\mathsf{L5} \rightarrow_{wr} r$ for $r \in \{\mathsf{L8}, \mathsf{L11}\}$. A similar transitivity argument yields C4 from C1 and C2. It can be shown that imposing the orderings

C1* $\mathsf{L6} \rightarrow_{wr} r$, for $r \in \{\mathsf{L8}, \mathsf{L11}\}$

C2 $w \rightarrow_{wr} \mathsf{L14}$, for $w \in \{\mathsf{L9}, \mathsf{L12}\}$

alone, as seen in Fig 12b, yields a valid SC schedule Σ_1. Specifically, under SC-admissibility there is no reachable configuration in which L9 and L12 conflict, eliminating the need for C3. To show this we argue as follows: If any of the conditionals L8 or L11 were executed before the assignment L6, then violation of condition SC2 becomes unavoidable, hence the run cannot be completed to an SC-admissible macro step. Since the assignment L6 must be executed strictly earlier, both the conditionals L8 or L11 must consistently see the *same* memory state $\mathsf{s} = \mathsf{true}$. This has two consequences. Firstly, only L12 becomes active, while L9 is skipped. In other words, SC-admissibility eliminates the free schedule L4, L5, L8, L6, L11, so that the conflicting absolute writes L12 and L9 are never jointly active. Secondly, starting with the configuration in which any of the conditionals L8 or L11 is executed, we have $\mathsf{s} = \mathsf{true}$, which is never subsequently changed. This means that the writes L15 and L16 are confluent with the reads L8 and L11 because, in line with Def. 8, they are never in conflict. Thus, the constraints $w \rightarrow_{wr} r$, for $w \in \{\mathsf{L15}, \mathsf{L16}\}$ and $r \in \{\mathsf{L8}, \mathsf{L11}\}$ are redundant. We may safely remove them from Σ_0 without losing completeness of the schedule as expressed in Def. 12.

Since the remaining constraints C1* and C2 form a valid and acyclic SC schedule Σ_1, GuardedA is ASC-schedulable by Def. 13 and hence sequentially constructive by Thm. 1. Note that to find the schedule C1* and C2 and prove its validity involves reasoning about execution orders and data dependencies. In practice, different compilers will be distinguished by how much such analysis capability they provide. Our notion of ACS-schedulability leaves a lot of room for semantical abstraction or refinement, as discussed below in Sec 6.1.

An example of a program for which no compiler, however clever, can find a valid and acyclic SC-schedule is NonDet from Fig. 10. Every valid SC-schedule must contain a cycle that involves $\rightarrow_{wr}$ edges, and therefore NonDet is not ASC schedulable. As explained above, NonDet indeed exhibits non-determinism under SC-admissible scheduling.

To force determinism we would have to eliminate one of the two SC-admissible runs without blocking off the other. This is seen in the program SCDet in Fig. 13, which is almost like NonDet, but employs Fail to force one of the two parallel threads to fail. Consequently, there is only one SC-admissible schedule left over, viz. first to execute the test L5 immediately followed by the assignment L6. This sets y = true and prevents the second thread to run into Fail. This means that we no longer need to consider the read-write conflict on variable x. It suffices to include the single constraint $\mathsf{L6} \rightarrow_{wr} \mathsf{L8}$ on variable y to obtain a valid acyclic SC-schedule.

```
1  module SCDet
2  output bool x = false; y = false;
3  {
4    fork // must be executed atomically
5      if (!x)
6        y = true;
7    par // failing thread
8      if (!y)
9        { x = true; Fail }
10   join
11 }
```

Figure 13: SCL program SCDet

It is important to observe that dropping the constraint L9 $\rightarrow_{wr}$ L5 from the schedule, or alternatively, keeping L9 $\rightarrow_{wr}$ L5 and dropping L6 $\rightarrow_{wr}$ L8, is not an option for program NonDet in Fig. 10. Although each would remove the deadlock from the execution and resolve non-determinism, none of these schedules is valid for NonDet. This is quite sensible since each of these "fabricated" schedules yields a different response. We would simply trade non-determinism resolved by the run-time for non-determinism resolved by the compiler. Both is equally bad from the programmer's point of view.

6.1 Conservative approximations

The concepts underlying the definition of ASC schedulability (Def. 13), namely when statement nodes are concurrent and how they access a shared variable (type of write), may involve run time information that a compiler cannot infer. E. g., it is undecidable in the general case of whether the evaluation of two syntactically different expressions results in the same value. In such cases, a compiler must work with conservative approximations, as further detailed in this section. This does not inflict on the validity of the approach presented here, and in particular does not introduce any non-determinism. However, such approximations may suggest data dependencies where in fact there are none in any concrete program execution, and thus it may lead a compiler to reject a program that another compiler with better analysis capabilities would consider ASC schedulable and therefore would accept. Then again, this situation is not uncommon in, e. g., hardware synthesis or Esterel compilation, where different compilers with different analysis capabilities may accept different programs as being free of data races.

Strictly, Def. 13 is not defining a fixed set of relations $\rightarrow_\alpha$ but a *class* of relations for SC-scheduling, depending on what information we may care to infer, statically, about which nodes are relative and which are absolute writes, which nodes are non-confluent with each other and may be executed concurrently in the same tick. Specifically, the latter will always be a conservative approximation, in the sense that "nodes may execute in the same tick" means that we cannot disprove it. But note that even with an exact analysis on which nodes can appear together in the same tick, concurrency will still only be an approximation. The reason is that two concurrent nodes may well be executable in the same tick but still sequentially ordered by data dependencies and synchronization.

For example, assume a pair of writes that are assumed to be non-confluent, in concurrent threads. These writes may be guarded by expressions that are mutually exclusive, in which

case, according to our definition, these writes would not be concurrent. However, a compiler may or may not recognize mutual exclusivity and thus may or may not consider the writes to be concurrent. Possible levels of analysis are:

1. No analysis: don't recognize mutual exclusivity.
2. Simple negation: recognize textually-identical-except-for-a-not-operator guards as being mutually exclusive.
3. Logical negation: recognize guards that can be proven to be logically inverse for all truth table values as being mutually exclusive.
4. Logical-over-time negation: recognize guards that can be proven to be logically inverse for all values that can actually occur in the course of running the program as being mutually exclusive.

Similarly, a compiler may perform different levels of analysis on whether two statements are in the same tick, for example:

1. No analysis: consider all statements to be executed in the same tick.
2. Simple linear tick count analysis: detect when statements are executed a different number of ticks since the start of their respective threads. This might involve traversing different branches, and other techniques such as dead code elimination to rule out ineffective pause statements.
3. Loop tick count analysis: detect when statements in loops are aligned such that they are always executed in different ticks since the start of their respective threads.

Similarly, there are several aspects of the data dependency definitions that a compiler must approximate conservatively, and which may lead rejection of programs that are in fact SC:

- A compiler may not recognize that writes are confluent, thus introducing superfluous ww, wr or wi edges.
- A compiler may not recognize that a write is a relative write, falsely classifying it as an absolute write. This may replace ir edges by wr edges (which is harmless), or may introduce superfluous wr edges (which may introduce superfluous cycles), or may introduce superfluous ww edges.

These approximations involve the analysis of values, which might be, for example:

1. No analysis: consider all absolute writes as non-confluent.
2. Text equivalence: accept absolute writes as confluent that are textually equivalent (recall that we rule out side effects).
3. Mathematical equivalence: recognize absolute writes as confluent that involve logically equivalent expressions, i. e., "$a + b$" and "$b + a$".

6.2 Determining SC schedules

For a sequentially constructive program, a *valid schedule* is one which executes concurrent statements in the order induced by $\rightarrow$. Such a schedule may be implemented by associating a *priority* $n.pr$ with each statement node n.

Definition 14 (Priorities)**.** *Given a SC-schedule $\rightarrow_{wir}$, the* priority *$n.pr$ of a statement n is the maximal number of $\rightarrow_{wir}$ edges traversed by any path originating in n in the SCG.*

A scheduler that always gives control to the thread with highest priority, chosen from the set of threads that are still active in the current tick, never allows a statement with higher priority to wait on one with lower priority. Such a scheduler implements a valid schedule, as can be verified from the SCG construction. For example $n_1 \rightarrow_{wi} n_2$ implies $n_1 \rightarrow_{wir} n_2$, which implies, by definition of priorities, $n_1.pr > n_2.pr$, which in turn implies that n_1 gets scheduled before n_2. Thus absolute writes are performed before concurrent relative writes to the same variable. Similarly $\rightarrow_{wr}$ ensures that absolute writes are performed before concurrent reads of the same variable, and $\rightarrow_{ir}$ ensures that relative writes are performed before concurrent reads of the same variable.

The priority concept can also serve to determine sequential constructiveness, based on Thm. 1 and the following theorem:

Theorem 2 (Finite Priorities)**.** *A program is ASC schedulable iff there exists a SC-schedule such that all statement priorities are finite.*

Note that the existence of finite priorities imply there is no $\leftrightarrow_{ww}$ cycle, which means there is no $\rightarrow_{ww}$, dependency edge in the schedule.

Proof sketch: This follows from the observation that whenever ASC schedulability requires that n_1 must be scheduled before n_2, then n_1 gets assigned a higher priority than n_2. □

The statements executed within a tick always execute in decreasing priority order, and a thread may never fork off threads with higher priority than its current priority. Therefore a thread t currently executing with some priority pr, meaning that its priority is higher than that of all other threads that still are eligible for scheduling in the current tick, cannot be preempted by another thread u unless t just lowered its own priority below u's priority. Thus, the only points when a scheduler is called for are 1) when threads are forked and the scheduler has to schedule one of the forked threads, or 2) when a thread lowers its own priority, or 3) when a thread finishes for the current tick, that is, it reaches a pause statement or terminates.

Cycles induced solely by $\rightarrow_{seq}$, which correspond to instantaneous loops within a thread, do not impede on sequential constructiveness, as they do not entail race conditions, as already discussed in the context of the InstLoop example in Sec. 3.8. In practice, one might want to rule out *unbounded* instantaneous loops, which could lead to infinite statement sequences within a tick. However, this question is orthogonal to sequential constructiveness, and is more related to timing predictability and program correctness in general.

6.3 Computing priorities

The calculation of priorities (Def. 14) can be formulated as a longest weighted path problem. We assign to each edge $e \in E$ a weight $e.w$, with $e.w = 0$ iff $e.src \rightarrow_{seq} e.tgt$, and $e.w = 1$ iff $e.src \rightarrow_{wir} e.tgt$. Note that the relations $\rightarrow_{wir}$ and $\rightarrow_{seq}$ exclude each other, as statements cannot be sequential *and* concurrent to each other, so the weight of each edge is uniquely determined.

With this assignment of weights, $n.pr$ becomes the maximal weight of any path originating in n.

A non-trivial aspect in calculating priorities is that we want to handle (sequential) loops, i. e., cyclic SCGs. In the usual synchronous MoC, loops are prohibited when they can occur within a tick; this simplifies the scheduling problem, but is again more restrictive than necessary to ensure determinism. For arbitrary (i. e., possibly cyclic) weighted graphs, the computation of the longest weighted path is an NP-hard problem, as it can be reduced to the Hamiltonian path problem. However, according to our definition of SC, we can exclude all graphs that contain a cycle with a positive weight, as these cycles would contain a $\rightarrow_{wir}$ edge, which would mean that the program is not ASC schedulable. Thus we can compute priorities efficiently as follows:

1. Detect whether any positive weight cycles exist. We can do so by computing the Strongly Connected Components (SCCs), for example using Tarjan's algorithm [30], and checking if any SCC contains a node that is connected to another node within the same SCC by a $\rightarrow_{wir}$ edge.

2. If a positive weight cycle exists, the program is not ASC schedulable; we then **reject** the program and are done. Otherwise, we **accept** the program, and continue. Now nodes in the same SCC can reach each other, but only through paths with weight 0, and therefore must have the same priority.

3. From the SCCs, construct the directed acyclic graph $G_{SCC} = (N_{SCC}, E_{SCC})$, where $N_{SCC} \subset N$ contains a representative node from each SCC of G (using e. g. the SCC roots computed by Tarjan's algorithm), and E_{SCC} contains an edge from one SCC representative to another iff the corresponding SCCs are connected in G. Here we assign an edge in E_{SCC} the maximum weight of the corresponding edges in E.

4. Compute for each $n_{SCC} \in N_{SCC}$ the maximum weighted length (priority) $n_{SCC}.pr$ of any path originating in n_{SCC}. This can be done with a depth-first recursive traversal of all edges in the acyclic G_{SCC}.

5. Assign each statement $n \in N$ the priority computed for its SCC.

Note that we can perform all these steps in time linear to the number of nodes and edges of the graph. For the Control example, the resulting priorities are indicated in Fig. 2c.

7 Variables vs. Signals

The SC MoC does not mandate directly an Esterel/SyncChart-like signal mechanism, as described in Sec. 2. However, with the SC scheduling regime described in Sec. 5.2, signals can be emulated with ordinary boolean variables, as are present in Java and can be represented with integers in C.

7.1 Emulating pure signals

To make a boolean s behave like a signal, declared at some scope (which can be the whole program), we can use s as follows:

1. Initialize s to be absent at the beginning of each tick by adding a thread while (true) { s = false; pause } that runs concurrently to the scope of s. This constitutes an absolute write of s.

2. To emit s, i. e., to make it present, perform s = s || true. This constitutes a relative write, which can be combined with other, concurrent relative writes.

3. Interpret s == true as s being present, s == false as absent. These constitute reads of s.

7.2 Emulating valued signals

For valued signals without a combination function, we must statically check that they cannot be emitted concurrently. If this is the case, the emission of the valued signal becomes a simple absolute write to an ordinary variable, plus the emission of a pure signal that indicates the presence status.

The more interesting case, considered in the following, are valued signals that are equipped with a combination function and that can be emitted concurrently. A slight complication is that the value of a signal is persistent across ticks. That is, if the signal is not emitted in the current tick, we must remember its value from the previous tick. However, if it is emitted, we must consider it to be initialized with the neutral element of the combination function.

We can emulate a valued signal of some type, say integer, and some combination function, say +, as follows. A boolean s indicates signal presence/absence, as with pure signals. An integer sval carries the value of the signal, an auxiliary integer scur is used to collect emissions of the current tick. These variables are used as follows:

1. Initialize s with false at every tick in the scope of the signal, as for pure signals. Also at every tick, initialize scur with 0, the neutral element for +.

2. To emit the signal with some value ex, execute scur = scur + ex; s = s || true.

3. To set s to its current value if it has been emitted in the current tick, perform while (true) { if (s) sval = scur; pause } concurrently to the scope of the signal.

4. Refer to sval to retrieve the value of the signal, and interpret s == true as the signal being present, as for pure signals.

Note how the use of the auxiliary variable scur allows to maintain schedulability of concurrent emissions. If we would operate directly on sval, we would possibly have to perform an absolute write—for initialization with the neutral element—at any signal emission. This would destroy schedulability if we would try to emit different values at the same tick.

7.3 The 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, as introduced in Sec. 7.1. 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 _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.

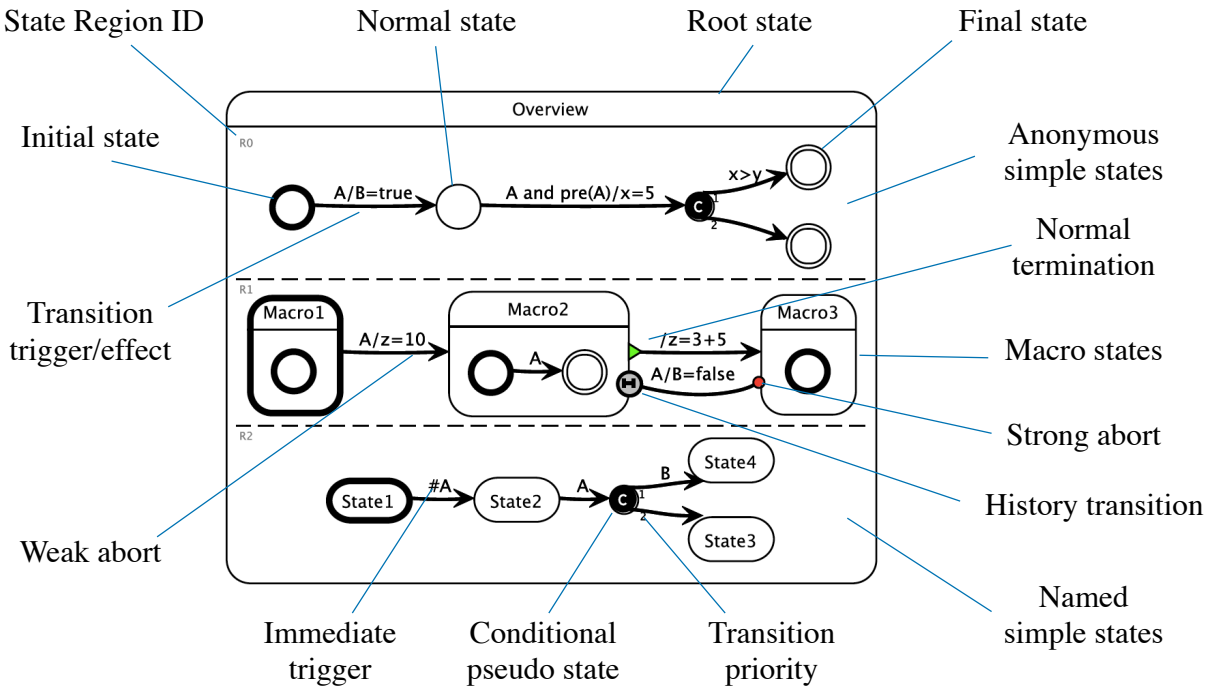


Figure 14: Overview of SCCharts

However, taking a broader look at the issue, we consider the need to use the pre operator at all to be much lower than in the current synchronous MoC. After all, a very typical use of the pre operator in the synchronous MoC is to break causality cycles that arise if we disregard any sequential order. The SC MoC already makes these cycles disappear by restricting the write-before-read requirement to truly concurrent variable accesses.

8 SCCharts — Sequentially Constructive Statecharts

As indicated earlier, it is natural to apply the concept of sequential constructiveness not only to a textual C/Java-like language, but also to a graphical formalism such as Statecharts [4]. In fact, the development of a semantically sound, yet flexible and intuitive Statechart dialect was the original motivation for developing the SC concept. We have developed such a Statechart dialect, named Sequentially Constructive Statecharts, or SCCharts in short, to be used for the development of safety-critical embedded systems in an industrial setting. We here do not present SCCharts in detail, as this would go beyond the scope of this paper, but outline its key characteristics. In particular, we discuss how it relates to other Statecharts dialects with respect to what Statecharts are considered admissible.

Compared to the SCL language introduced in Sec. 3.1, which already contains concurrency (the par statement) and state (pause), the main semantic addition of Statecharts is preemption. SCCharts provide two types of preemption, *strong* preemption and *weak* preemption, analogous to SyncCharts. Adding the concept of preemption allows to write more compact diagrams, but does not really add to the expressive power. Therefore, there are no particular challenges involved in mapping preemption to the SC model of computation. An overview of the elements of SCCharts is shown in Fig. 14.

An SCChart that corresponds to the XY example (see Fig. 8) is shown in Fig. 8e.

An SCChart that illustrates preemption, called ABSWO, is shown in Fig. 15a. As noted, preemption can be emulated with auxiliary signals and normal termination. The ABSWO SC-Chart shown in Fig. 15b is the correspondingly expanded variant of ABSWO. This in turn can be expressed as textual SCL program, see Fig. 15c, with the corresponding SCG in Fig. 15e. The SCG nodes show the statement priorities computed by our algorithm in square brackets.

9 Alternative Notions of Constructiveness

As noted in the beginning, SC-scheduling is not the only way to interpret the Synchrony Hypothesis. In the following let us mention the three most prominent approaches discussed in the literature. The induced notions of P-, L-, and B-constructiveness are due to Pnueli & Shalev, Boussinot and Berry, respectively. A survey and detailed mathematical analysis of these schemes can be found in [31].

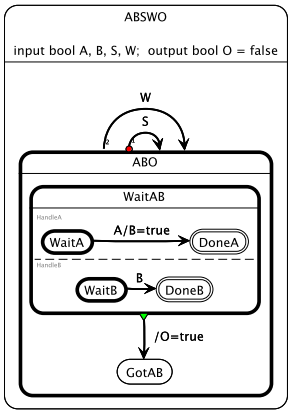
The Statecharts dialects we consider here use broadcast communication on (pure) signals. We assume these are translated into our imperative language, with shared boolean variables, as described in Sec. 7.1. Hence, we assume signals are initialised to false at the beginning of each macro tick and the only assignments to a signal s are relative writes s = s || true setting s true. Since there is no risk of confusion with absolute writes these may simply be represented as constant assignments s = true. Moreover, for compactness of the code we use numbers 0, 1 for the boolean values instead of false, true.

An overview of how sequentially constructive Statecharts (called *S-constructive*, for short, in the sequel) relate to these other classes of Statecharts is given in Fig. 16.

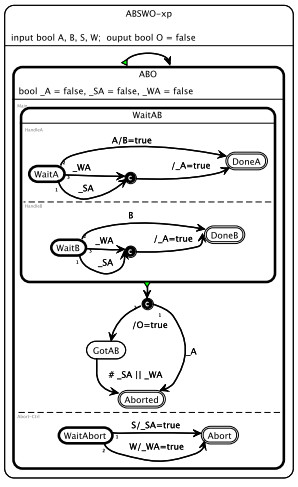
P-Constructiveness (PC) As one of the first semantics for Statecharts, Pnueli & Shalev [6] have introduced a (non-deterministic) execution model (P) which is distinguished by the fact that it permits speculation on the absence of a signal. As long as no thread has emitted a signal, say y, the P-scheduler may execute a command guarded by the absence of y. In doing so, the guard signal y is fixed to have absent status for the whole synchronous instant. P-admissible executions must be *globally consistent* and the status of a signal not be overridden by any thread. Otherwise, the scheduler detects a conflict and backtracks in search of a viable schedule. As a consequence, *P-admissible* schedules lead to non-deterministic response behavior, like our SC-admissible schedules do, e.g., as exhibited by the program P_{NonDet} in Fig. 10 of Sec. 5.2. In line with Def. 11 a program then is *P-constructive* if there exists a conflict-free P-admissible schedule, and all P-admissible schedules induce the same response. For example, consider the program

$$\begin{aligned} P_{LP} \quad = \quad & \mathsf{fork}\ s_1 : \mathsf{if}\ (!\mathsf{y})\ s_{11} : \mathsf{x} = 1 \\ & \mathsf{par}\ s_2 : \mathsf{if}\ (!\mathsf{x})\ s_{21} : \{\mathsf{x} = 1; \mathsf{y} = 1\}\ \mathsf{join} \end{aligned}$$

which executes under P-admissible scheduling to generate the unique response y = 0, x = 1. Initially, both x and y are absent (by default), so both parallel statements s_1, s_2 are executable. If s_2 is chosen, then the guard x is frozen up to be absent, and the statement s_{21} becomes ready to be executed. Yet, this will eventually emit x, which contradicts the default speculation on the absence of x. Thus, this schedule is abandoned. If s_1 is executed instead, then the statement s_{11} sets x = 1. Now, the guard of s_2 is false, whence s_2 can enter its (implicit) else branch and terminate. This is the only P-admissible schedule, whence the response is deterministic and P_{LP} is P-constructive according to our terminology.



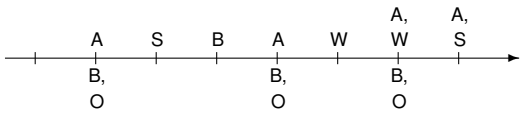
(a) SCChart ABSWO, with strong and weak preemption of ABO, triggered by S and W, respectively



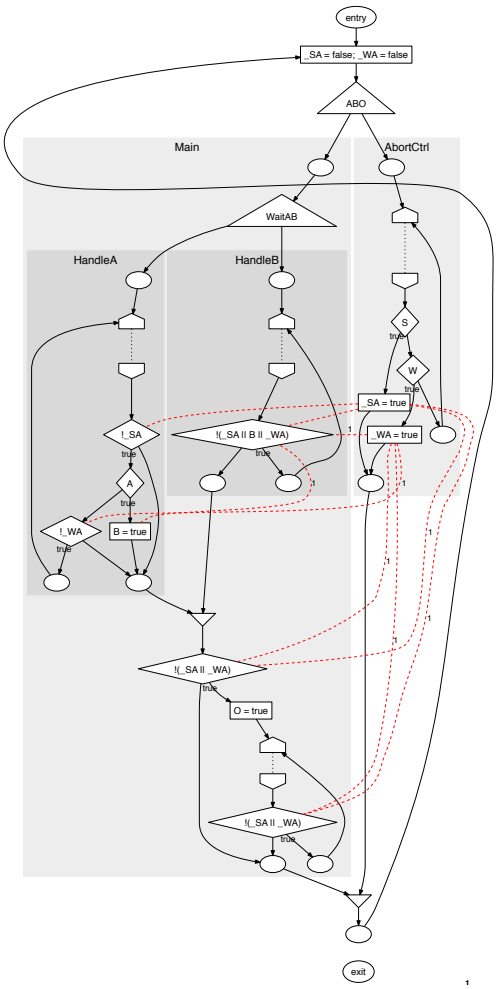
(b) Functionally equivalent SCChart ABSWO-xp, which emulates preemption

```
1 module ABSWO−xp
2 input bool A, B, S, W;
3 output bool O = false;
4 {
5   ABO: { // Local vars of ABO
6     bool _SA = false, _WA = false;
7
8     fork { // Macro State ABO
9       // Thread Main
10      fork { // Macro State WaitAB
11        // Thread HandleA
12        WaitA: pause;
13        if (!_SA) {
14          if (A) {
15            B = true;
16            goto DoneA;
17          } else if (!_WA)
18            goto WaitA;
19        } _A = true;
20        DoneA:
21      } par {
22        // Thread HandleB
23        WaitB: pause;
24        if (!_SA) {
25          if (B)
26            goto DoneB;
27          else if (!_WA)
28            goto WaitB;
29        } _A = true;
30        DoneB:
31      } join;
32
33      if (!_A) {
34        O = true;
35        GotAB:
36        if (!(_SA || _WA)) {
37          pause;
38          goto GotAB;
39        }
40      }
41      // State Aborted
42    } par {
43      // Thread Abort−Ctrl
44      WaitAbort: pause;
45      if (S)
46        _SA = true;
47      else if (W)
48        _WA = true;
49      else
50        goto WaitAbort;
51      // State Abort
52    } join;
53  }; // Leave scope of local vars
54  goto ABO;
55 }
```

(c) SCL program ABSWO-xp



(d) Possible execution trace, with inputs above the tick time line and outputs below



(e) The SC Graph; fork nodes are labelled with corresponding macro states. All data dependencies are of wr type and hence their labels are omitted.

Figure 15: The ABSWO example, illustrating strong and weak preemption, and the functionally equivalent ABSWO-xp ("expanded") example, which emulates preemption with an Abort-Ctrl handler and auxiliary control signals _A, _SA and _WA.

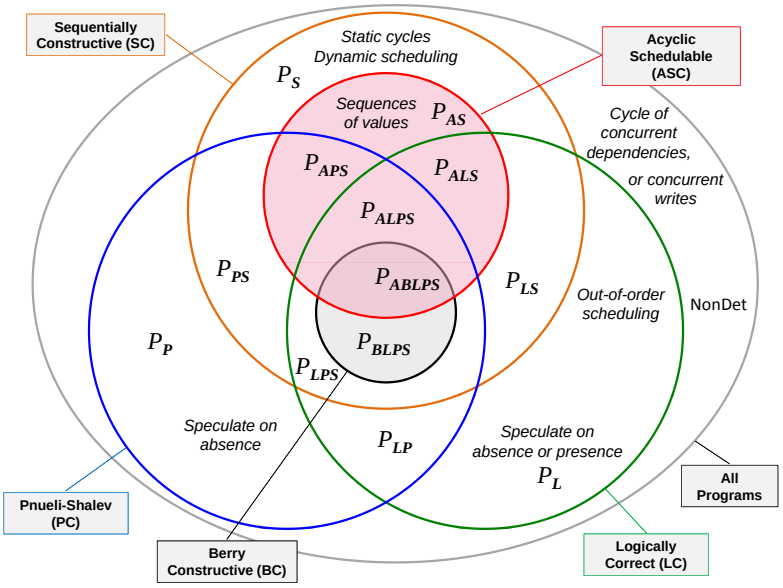


Figure 16: Relationships of Statecharts classes

Since the SC-schedule also runs P_{LP} from the initialisation x = 0; y = 0 to model the signals, it might seem we should get the same result. However, as P-schedules are dynamically self-correcting, they may be deterministic where SC-schedules are not. Concretely, P_{LP} has two SC-admissible schedules. One executes s_1 and then s_2 atomically and produces the result y = 0, x = 1, just like in the P-schedule above. The other is to execute s_2 first and then s_1. This is viable under SC-rules since the update of x = 1 after reading x = 0 in the guard of s_2 is permitted and the update done in the same thread. Thus, we have two distinct responses whence P_{LP} is not S-constructive. Observe that the program

$$\begin{aligned} P_S \quad = \quad & \mathsf{fork}\ s_1\mathsf{:}\ \mathsf{if}\ (\mathsf{!x})\ \{\mathsf{y = true; x = true}\} \\ & \mathsf{par}\ s_2\mathsf{:}\ \mathsf{if}\ (\mathsf{!y})\ \{\mathsf{y = true; x = true; Fail}\}\ \mathsf{join} \end{aligned}$$

which is S-constructive (for the same reason as SCDet from Fig. 13) is not P-constructive. No matter which of the guards if (!x) or if (!y) is scheduled first, testing the absence of the signal, the very signal will be emitted later, which is not P-admissible. Still there are programs such as

$$\begin{aligned} P_{PS} \quad = \quad & \mathsf{fork}\ s_1\mathsf{:}\ \mathsf{if}\ (\mathsf{!x})\ \{\mathsf{y = true}\} \\ & \mathsf{par}\ s_2\mathsf{:}\ \mathsf{if}\ (\mathsf{!y})\ \{\mathsf{x = true; Fail}\}\ \mathsf{join} \end{aligned}$$

which are both P and S-constructive. Thus, the properties of being P or S-constructive are independent from each other.

L-Constructiveness (LC) A different, and in some sense more permissive, scheduling scheme underlies the notion of *logical coherence* (L) defined by Boussinot [28] and also Berry [3]. L-schedules may speculated on both absence and presence of signals subject to the constraint that the response be *logically coherent* in the sense that a signal is present throughout a tick iff it is emitted in that tick. If no signal emission statement is executed, the signal must be consistently interpreted as absent. In addition, like for all Statecharts dialects, signals cannot be unemitted.

A program is called *L-constructive* if it has exactly one logically coherent response in each macro tick. As an example consider the program

$$\begin{aligned} P_L \quad = \quad & \textsf{fork } s_1 : \textsf{if (y) } s_{11} : \textsf{x = 1} \\ & \textsf{par } s_2 : \textsf{if (x) } \{\textsf{x = 1; } s_{21} : \textsf{y=1}\} \textsf{ else y = 1 join.} \end{aligned}$$

The L-schedule may guess the (emit) statements s_{11} and s_{21} to be executed (out of order) which makes both signals x, y become present. This justifies the execution of the then (true) branch of s_1 and the then branch of s_2, which confirms the initial speculation that s_{11} and s_{21} are executed in the macro tick. One can show that this is the only logically coherent schedule: Since the signal emission y = 1 appears in both branches of s_2, it will happen irrespective the status of x. Now, if y is present, the then branch of s_1 is taken, which emits x. Hence, x = 1, y = 1 is the unique L-response of P_L, so it classifies as being *L-constructive*.

On the other hand, P_L is not S-constructive. Let us see why. From the initialisation x = 0; y = 0 we can schedule either the (implicit) else branch of s_1 or the else branch of s_2. However, in the ordering s_1, s_2 the second thread executing s_2 would write y = 1 after s_1 has tested y for absence. This is a concurrent-write-after-read conflict. If we schedule s_2, s_1, then the thread for s_2 sets y = 1 first, which forces s_1 to run s_{11} writing x = 1. Again, this is a concurrent-write-after-read conflict since s_2 tested x before. So, we find that P_L does not admit any SC-schedules, and thus is not S-constructive. Note that P_L not P-constructive either. No matter which of the conditional tests in s_1 and s_2 are executed first, we run into a conflict. Signal x will necessarily be emitted after it has been tested absent by s_2.

One can show that L-constructiveness is incomparable with both the P and S classes of constructiveness. The program P_S (see p. 43), which is S-constructive, but neither P-constructive nor L-constructive: It is not L-constructive because it does not admit any coherent solutions: x = 0 is excluded since then s_1 would force x = 1 to be executed, and y = 0 is impossible because this would result in the execution of y = 1 in s_2. But if x = 1 and y = 1, then both of the guards if (!x) and if (!y) are switched off and neither x = 1 nor y = 1 is executed. At the same time there are programs such as

$$\begin{aligned} P_P \quad = \quad & \textsf{fork } s_1 : \textsf{if (x) } s_{11} : \textsf{y = 1} \\ & \textsf{par } s_2 : \textsf{if (y) } s_{21} : \textsf{x = 1} \\ & \textsf{par } s_3 : \textsf{x = 1} \\ & \textsf{par } s_4 : \textsf{if (z) } s_{41} : \textsf{z = 1 join} \end{aligned}$$

which is P-constructive but neither S-constructive nor L-constructive. Any admissible P-schedule may execute nodes from $\{s_1, s_2, s_3\}$ and from s_4 in arbitrary interleaving since they do not depend on each other. Due to the initialization z is found absent, whence s_4 exits immediately. Regarding the scheduling of nodes within $\{s_1, s_2, s_3\}$ we are not free: The P-schedule must first execute s_3, setting x present, then s_1, atomically, followed by s_2. Otherwise, if the test if (x) in s_1 is evaluated before the assignment s_3, or if (y) in s_2 is evaluated before the assignment y = 1 from s_1, we would override a signal tested absent by emitting it. The final P-response is x = 1, y = 1, z = 0.

P_P is not *S*-constructive since all schedules must necessarily execute the writes s_3 or s_{21} after the test if (x) in s_1, or s_{11} after the test if (y) in s_2, which all violates condition SC2 of SC-admissibility. P_P is not *L*-constructive since it admits the two logically coherent solutions x = 1, y = 1, z = 0 and x = 1, y = 1, z = 1.

Any two of the properties of S, P and L-constructiveness can be combined without the third: The program P_{LP} given above, which is not S-constructive, is not only P-constructive but also

L-constructive. Its only P-admissible response y = 0, x = 1 is also the only logically coherent speculative execution for P_{LP} in which a signal is present if and only if it is emitted. Program P_{PS} is both P and S-constructive but not L-constructive, because of its two coherent solutions x = 0, y = 1, z = 1 and x = 1, y = 0, z = 1. Finally,

$$\begin{aligned} P_{LS} \quad = \quad & \textsf{fork } s_1 : \textsf{if (!x) } s_{11} : \textsf{x = 1} \\ & \textsf{par } s_2 : \textsf{if (x \&\& y) } s_{21} : \{\textsf{x = 1; y = 1}\} \textsf{ join} \end{aligned}$$

is L and S-constructive but not P-constructive. All executions start with an evaluation of x in the guards of s_1 or s_2 and then eventually emit x with s_{11} which is not P-admissible. However, there is one SC-admissible schedule which first atomically executes s_1 (emitting x) and then s_2 in which the concurrent write s_{21} to x is not executed. The write s_{11} takes place sequentially after the test if (!x) at the beginning of s_1 and is SC-admissible. Finally, P_{LS} admits of only one coherent logical assignment, viz. x = 1, y = 1 as one shows without difficulty.

B-Constructiveness (BC) Finally, the most well-known interpretation of the Synchrony Hypothesis can be found in the semantics propagated by Berry [29, 3] for the language Esterel, which has also been adopted by SyncCharts [32]. It is known as the "constructive" semantics and motivated from the theory of boolean circuits which have been an important target for Esterel compilers. Every Esterel program compiles naturally into a circuit [3] such that the resulting circuit is delay-insensitive [12] iff the program is constructive [13]. This means that the response behaviour of constructive Esterel programs is time-bounded and deterministic under arbitrary node and wire delays on a fully concurrent execution architecture (such as circuits). This fully concurrent, asynchronous circuit scheduling, is more admissible than any of the SC-, P- or L-scheduling schemes. Not surprisingly, the notion of constructiveness deriving from it, which universally quantifies over all admissible schedules ("all concurrent asynchronous schedules produce the same deterministic response"), is rather stringent. Fewer programs satisfy this condition, which we propose to call *B-constructive*, for the purposes of this discussion.

To understand B-constructiveness in terms of our shared variable model, we can use its characterisation through ternary analysis [14, 13]. Ternary, or "must/cannot" analysis on the source Esterel program [3] is an abstract simulation that tries to justify the presence and absence status of all signals without any form of speculation. Ternary simulation works like a simultaneous breadth-first exploration of all threads to propagate signal absence and present statuses. A conditional can only be executed if all signals in the guard have been fully resolved. This means that, based on the accumulated signal statuses, it can be ruled out that any write ever changes the value of the guard, later. In particular, signal absence is no longer a default value but must be justified. A ternary schedule will get stuck, i. e., deadlock ("waiting for stabilisation"), in case not all signal statuses are uniquely resolved to be absent or present for the tick. Note that this lock-up in ternary simulation indicates non-constructiveness by the non-existence of a terminating admissible schedule. It exploits the existential quantification in the definition of constructiveness ("there exists a terminating ternary schedule"). Ternary simulation and asynchronous circuit execution are equivalent notions of B-admissible scheduling to characterise B-constructiveness from different ends of the stick.

Consider the example P_{LP}, which is not B-constructive: The B-schedule cannot initially resolve the status of either x or y as absent, because both can be emitted by the branches s_{11} and s_{21} and because their execution cannot be excluded as the guards of s_1 and s_2 have not yet been resolved. We are locked up in a causal cycle. Note that a P-schedule can run s_1 by itself, based on the default absence of y, which blocks s_2 and prevents the deadlock.

The L-constructive example P_L is not B-constructive either: Initially, x cannot be decided as absent, since s_1 might still produce an emission for it. To exclude that, the guard variable y of s_1 would have to be resolved absent. But this is not guaranteed because statement s_{21} could set y=1. This cannot be excluded either since we cannot resolve the status of the guard signal x of s_2. Again, the schedule is stuck in a justification cycle without finite grounding.

The same is true for P_P. Starting the ternary simulation from x = $\bot$, y = $\bot$ in which none of the conditional guards of s_1 or s_2 is resolved, we cannot conclude that either is executed or that it isn't. Hence, we do not know if x (y) must be emitted or cannot be emitted, viz through execution or blocking of the "then" branch of s_2 (s_1).

Similarly, one shows that none of the other programs P_P, P_S, P_{PS} or P_{LS} discussed so far, are B-constructive. If we translated these programs into gate-level circuits we would find that they are not delay-insensitive, showing non-deterministic or oscillatory behavior under the non-inertial delay model [13].

From the results in [31] it follows that every B-constructive Esterel program is also P-constructive and L-constructive. One can show that B-constructive programs are also S-constructive, so that B-constructiveness properly lies inside the intersection of our classification as seen in Fig. 16. In particular, S-constructiveness is strictly larger than B-constructiveness. Concretely, all programs without the parallel fork-par-join operator are trivially S-constructive, but many fail to be B-constructive. The simplest examples are the programs

$$\begin{aligned} P_{AS} &= s_1 : \text{if (!x) } s_{11} : \text{x = 1} \\ P_{APS} &= s_1 : \text{if (x) } s_{11} : \text{x = 1} \\ P_{ALS} &= s_1 : \text{if (!x) } s_{11} : \text{x = 1 else x = 1} \\ P_{ALPS} &= s_1 : \text{if (!x \&\& y) } s_{11} : \{\text{x = 1; y = 1}\} \end{aligned}$$

which are all trivially ASC schedulable and therefore S-constructive by Thm. 1. After all, without the parallel operator there cannot be any dependency cycle through $\rightarrow_{wir}$ edges. However, one shows that none of them is B-constructive. In each case the signals x and y tested in the conditionals are potentially emitted by the if statement. So, in order to decide (by ternary simulation) the status of x and y, we need to decide whether or not the if branch is executed. But this means we need to know the status of the signals tested, in the first place. This is a logical cycle. The above programs have SC-admissible schedules because of the default initialisation of signals (absolute writes) and the fact that a variable may be freely overwritten within a thread. This gains substantial extra ground for programming synchronous interactions in SCL compared to the existing imperative synchronous code which is based on B-constructiveness. At the same time, so we believe, S-constructive programs are more practical, than either P- and L-constructive programs.

To complete this section let us mention that the class of ASC schedulable programs, although more restricted, is independent of the PC, LC and BC classes of constructiveness, just like S-constructiveness:

- For instance, none of P_{AS} or P_{APS} is L-constructive. The former because is has no logically coherent (speculative) execution, and the latter because it has two different ones. P_{ALS} is L-constructive with the unique coherent response x = 1.

- Among the programs P_{AS}, P_{APS} and P_{ALS} only only P_{APS} is P-constructive. Assuming signal variables are initialised to 0, executing s_1 of P_{APS} exits into the implicit 'else' branch of the conditional and terminates. This is the only P-admissible schedule of P_{APS}.

Executing s_1 in P_{AS} or P_{ALS}, however, activates the "then" branch in which s_{11} emits x. This raises a failure since x has just been tested absent in the evaluation of the conditional. Since there is no other conflict-free P-schedule, program P_{AS} is not P-constructive.

- P_{ALPS} lies in all three classes LC, PC, and SC.

In the other direction, there are example programs

$$\begin{aligned} P_{LPS} &= \text{fork } s_1 : \text{if (!x \&\& z) } s_{11} : \{\text{y = 1; z = 1}\} \\ &\quad\ \text{par } s_2 : \text{if (y) } s_{21} : \text{x = 1 join} \\ P_{BLPS} &= \text{fork } s_1 : \text{if (!x \&\& z) } s_{11} : \text{y = 1} \\ &\quad\ \text{par } s_2 : \text{if (y) } s_{21} : \text{x = 1 join} \\ P_{ABLPS} &= s_1 : \text{if (x) } s_{11} : \text{y = 1} \end{aligned}$$

to separate BC from ASC and the intersection of LC, PC and SC. To sum up, we find that besides the inclusions $BC \subset SC \cap PC \cap LC$ and $ASC \subset SC$ all the classes are independent as illustrated in Fig. 16.

10 Summary and Outlook

Relying on a scheduler that is blind to shared variable accesses, such as a Java thread scheduler, makes concurrent programming a difficult endeavor with generally unpredictable outcome. The SC MoC presented in this paper harnesses the synchronous MoC where it truly matters, namely to ensure determinism when shared variables are accessed concurrently, and combines this with the flexibility and familiarity of sequential programming.

The SC MoC domain and associated algorithm, introduced in this paper, for static analysis and scheduling sequentially constructive programs, provide for a novel regime to achieve deterministic concurrency for C/Java-style imperative programming. The scheme borrows ideas from synchronous programming such as the fork…par…join construct and global clock synchronisation through the pause statement. By exploiting the inherent sequential program order we can compile more programs than existing synchronous programming languages without losing determinism. This not only adds expressive power compared to established synchronous programming, but allows programmers versed in sequential languages to harness the SC MoC and the deterministic concurrency it provides without giving up familiar, safe programming patterns that are merely sequential. By synchronising hierarchically instantiated threads on global tick barriers, potential non-determinism is restricted, which considerably simplifies static scheduling analysis.

Current and future work entails gathering practical experience with full-scale applications, as well as a further theoretical investigation as how the SC MoC compares to other concurrent MoCs.

Acknowledgment

The material presented here has benefited greatly in substance and presentation from discussions with Hugo Andrade, Stephen Edwards—who rightfully pointed to relationships with static single assignment techniques—, Jeff Jensen, Stephen Mercer, Murali Parthasarathy, and Marc Pouzet.

References

[1] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," in *Proceedings of the IEEE, Special Issue on Embedded Systems*, vol. 91, Jan. 2003, pp. 64–83.

[3] G. Berry, *The Constructive Semantics of Pure Esterel*. Draft Book, 1999, ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[5] C. André, "SyncCharts: A visual representation of reactive behaviors," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–52, rev. RR 96–56, Rev. April 1996.

[6] 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)*. London, UK: Springer, 1991, pp. 244–264.

[7] S. A. Edwards, "Tutorial: Compiling concurrent languages for sequential processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 2, pp. 141–187, Apr. 2003.

[8] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, May 2007.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[10] G. Berry, "The foundations of Esterel," *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000, editors: G. Plotkin, C. Stirling and M. Tofte.

[11] Esterel Technologies, *The Esterel v7 Reference Manual Version v7_30—initial IEEE standardization proposal*, Nov. 2005, http://www.esterel-technologies.com/files/Esterel-Language-v7-Ref-Man.pdf.

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

[13] M. Mendler, T. Shiple, and G. Berry, "Constructive boolean circuits and the exactness of timed ternary simulation." *Formal Methods in System Design*, vol. 40, no. 3, pp. 283–329, 2012.

[14] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 950–956, Jul. 1994.

[15] K. Schneider, J. Brandt, T. Schüle, and T. Türk, "Improving constructiveness in code generators," in *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, UK, 2005.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[17] P. L. Guernic, T. Goutier, M. L. Borgne, and C. L. Maire, "Programming real time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, Sep. 1991.

[18] M. Pouzet and P. Raymond, "Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation," *Design Autom. for Emb. Sys.*, vol. 14, no. 3, pp. 165–192, 2010.

[19] D. Biernacki, J.-L. Colaco, G. Hamon, and M. Pouzet, "Clock-directed Modular Code Generation of Synchronous Data-flow Languages," in *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, AZ, USA, Jun. 2008.

[20] 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)*, Dublin, Jun. 2009.

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

[22] S. Andalam, P. Roop, A. Girault, and C. Traulsen, "PRET-C: A new language for programming precision timed architectures," in *Procedings of the Workshop on Reconciling Performace with Predictability (RePP), Embedded Systems Week*, Grenoble, France, Oct. 2009.

[23] O. Tardieu and S. A. Edwards, "Scheduling-independent threads and exceptions in SHIM," in *Proceedings of the Proceedings of the International Conference on Embedded Software (EMSOFT'06)*, Seoul, Korea, Oct. 2006.

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

[25] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop, "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation," Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report, ISSN 2192-6247, to appear.

[26] 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)*, Jersey City, NJ, USA, Sep. 2005.

[27] 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)*, Seoul, South Korea, Oct. 2006.

[28] F. Boussinot, "SugarCubes implementation of causality," INRIA, Research Report RR-3487, Sep. 1998.

[29] T. R. Shiple, G. Berry, and H. Touati, "Constructive Analysis of Cyclic Circuits," in *Proc. Int. Design and Test Conference (ITDC'96), Paris, France*, Mar. 1996.

[30] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal of Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[31] J. Aguado and M. Mendler, "Constructive semantics for instantaneous reactions," *Theoretical Computer Science*, vol. 241, pp. 931–961, 2011.

[32] C. André, "Computing SyncCharts reactions," *Electronic Notes in Theoretical Computer Science*, vol. 88, pp. 3–19, Oct. 2004.