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‡, and Owen O'Brien‡
*Department of Computer Science, Kiel University, Kiel, Germany, E-mail: {rvh, bdu, ima, cmot}@informatik.uni-kiel.de
†Bamberg University, Bamberg, Germany, E-mail: {michael.mendler, joaquin.aguado}@uni-bamberg.de
‡National Instruments, Austin, TX, USA. E-mail: {stephen.mercer, owen.o'brien}@ni.com

***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. 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.

I. 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. This may generate write/write and write/read *race conditions*, which are problematic with regard to ensuring deterministic behavior [1].

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. 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 *macro tick*, or *tick* for short, 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. 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. 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), which 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.

Outline. The next section discusses related work. Sec. III presents the SC language (SCL) and the SC graph (SCG), which are used in Sec. IV to define the SC MoC. Sec. V presents an approach to analyze whether programs are SC and to compute a schedule for them. We summarize in Sec. VI and outline the remaining aspects of SC not covered here but instead presented in an extended report [3].

II. Related Work

Edwards [4] and Potop-Butucaru et al. [5] 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) [6]. 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 [7] provides deterministic concurrency with shared *signals*. Causal Esterel programs on pure signals satisfy a strong scheduling invariant: they can be translated into constructive circuits which are *delay-insensitive* under the non-inertial delay model [8]. The algebraic transformations proposed by Schneider et al. [9] increase the class of programs considered constructive, but do not permit sequential writes within a tick. The notion of sequential constructiveness

This work has been supported in part by the German Science Foundation, as part of the PRETSY project (DFG HA 4407/6-1), and by National Instruments.
978-3-9815370-0-0/DATE13/©2013 EDAA

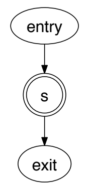
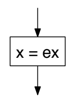
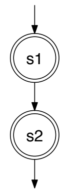
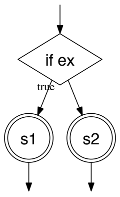
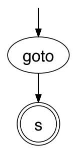
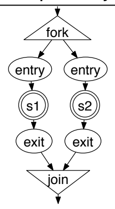
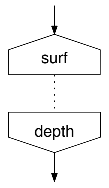
Statement type	Program	Assignment	Sequence	Conditional	Label / Goto	Parallel	Pause
SCL statement	s	$x = ex$	s_1 ; s_2	if (ex) s_1 else s_2	goto l … l : s	fork s_1 par s_2 join	pause
SCG statement nodes							

Fig. 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.

introduced here is weaker regarding schedule insensitivity, but more adequate for the sequential memory models available for imperative languages.

Caspi et al. [10] 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 [11], 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. V-B can be used to automatically synthesize priorities for Synchronous C. PRET-C [12] also provides deterministic reactive control flow, with static thread priorities.

SHIM [13] provides concurrent Kahn process networks with CSP-like rendezvous communication [14] 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.

Various other approaches with their own admissible scheduling schemes have been considered for Statecharts. The three most prominent approaches are due to Pnueli and Shalev [15], Boussinot [16] and Berry and Shiple [17]. None of them considers sequential control flow as SC does.

III. 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.

A. *The Control Example*

Fig. 2 shows the Control example, which 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. 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:

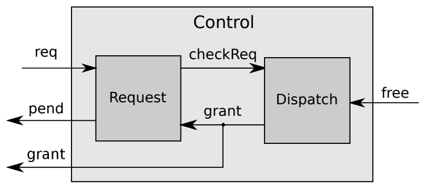
1) read input variables from the environment,
2) execute all active (currently instantiated) threads until they either terminate or reach a pause statement,
3) write output variables to the environment.

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. III-C), is not externally observable. Hence, when reasoning about deterministic behavior, we only consider the outputs emitted at the end of each macro tick.

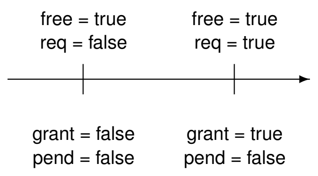
The execution of Control begins with a fork that spawns off Request and Dispatch. These two threads then progress on their own. Were they Java threads, a scheduler of some run time system could now switch back and forth between them arbitrarily, until both of them had finished. Under the SC MoC, their 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.

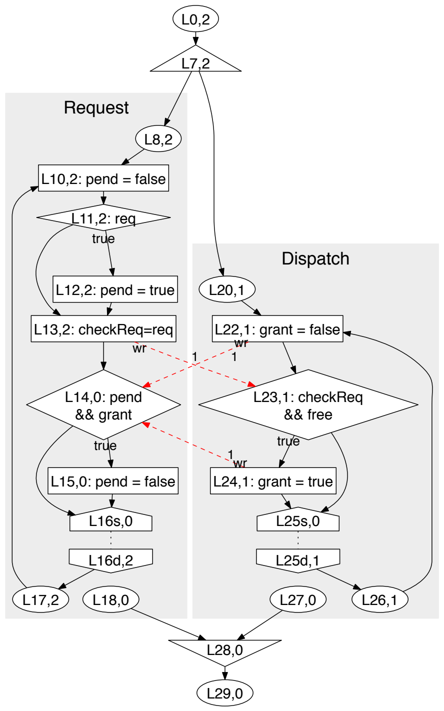
An interesting characteristic of Control is that the concurrent threads not only share variables, but also modify them sequentially. E. g., Dispatch first initializes grant with *false*, and then, in the same tick, might set it to *true*. Similarly, Request might assign to pend the sequence *false*/*true*/*false*. 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



(a) The dataflow view



(b) A tick time line, illustrating the first two ticks of an example trace



(c) The SC Graph (SCG)

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

Macro tick	a	2													2
Micro tick	i	1	2	3	4	5	6	7	8	9	10	11	12	13	13
Input	free	$\boldsymbol{t}$													t
vars	req	$\boldsymbol{t}$													t
Output	grant	f							f		t				$\boldsymbol{t}$
vars	pend	f		f		t								f	$\boldsymbol{f}$
Local var	checkReq	f					t								t
	C_{Root}	[L28]													[L28]
Continuations	C_{Request}	L16d	L10	L11	L12	L13	L14	L14	L14	L14	L14	L14	L15	L16s	(L16s)
	C_{Dispatch}	L25d	L25d	L25d	L25d	L25d	L25d	L22	L23	L24	L25s	(L25s)	(L25s)	(L25s)	(L25s)
Scheduled nodes R_i^a		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 shown in (b). We see for each micro tick the current variable values, where $\perp$ 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.

Fig. 2. The Control example. The SCG indicates 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. The SCG 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. V-B.

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.

B. 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. III-D), 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.

The set of threads of G is denoted T. Each thread $t \in T$, including the top-level Root thread, is associated with unique entry and exit nodes $t.en$, $t.ex \in N$.

Each $n \in N$ belongs to a thread $th(n)$, defined as the immediately enclosing thread $t \in T$ such that there is a control flow path to n that originates in $t.en$ and that does 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 consists of the threads $T = \{$Root, Request, Dispatch$\}$, and the Root thread consists of the statement nodes sts(Root) = {L0, L7, L28, L29}.

We define $fork(t)$ to be the fork node that immediately precedes $t.en$. Every $t \neq$ Root has an immediate *parent thread* $p(t)$, defined as $th(fork(t))$. In the example, p(Request) = p(Dispatch) = Root. We also recursively define the set of *ancestor threads* $p^*(t)$ to consist of t, $p(t)$, $p(p(t))$, …, Root.

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

Definition 1 (Concurrent Threads): Two threads $t_1, t_2 \in T$ are *concurrent*, denoted $t_1 \mid\mid t_2$, iff there exist $t_1' \in p^*(t_1)$, $t_2' \in p^*(t_2)$, with $t_1' \neq t_2'$, which share a common fork node $\mathit{fork}(t_1') = \mathit{fork}(t_2')$. We then refer to this fork node as the *least common ancestor fork*, $\mathit{lcafork}(t_1, t_2)$.

In Control, it is Request $\mid\mid$ Dispatch, whereas Root is not concurrent with any thread. Note that concurrency on threads can be checked with a simple, syntactic analysis of the program structure.

C. Macro Ticks and Micro Ticks

Definition 2 (Ticks): For an SCG $G = (N, E)$, a *(macro) tick* R, of length $\mathit{len}(R) \in \mathbb{N}$, is a mapping from micro tick indices $1 \leq j \leq \mathit{len}(R)$, to nodes $R(j) \in N$. A *run* of G is a sequence of macro ticks R^a, indexed by $a \in \mathbb{N}_{\geq 1}$.

One possible run of the Control example is illustrated in Fig. 2e. The *continuations* 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 whether a thread is currently *active*, i. e., eligible for execution, or *inactive*. Parenthesized node labels indicate that a thread is inactive because it has finished its current tick, whereas bracketed node labels indicate that a thread is inactive because it waits for its child threads to terminate. A $\perp$ denotes that there currently is no continuation associated with a thread, as it has either not been started yet or has terminated since the last start; we then also say that a thread is *disabled*, as opposed to *enabled*.

D. Concurrency of Node Instances

For a tick R, an index $1 \leq i \leq \mathit{len}(R)$, and a node $n \in N$, $\mathit{last}(n, i) = \mathit{max}\{j \mid j \leq i, R(j) = n\}$ retrieves the last occurrence of n in R at or before index i. If it does not exist, $\mathit{last}(n, i) = 0$. A *node instance* is a pair $ni = (n, i)$ consisting of a statement node $n \in N$ and a micro tick count $i \in \mathbb{N}$.

Definition 3 (Concurrent Node Instances): Two node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$ are *concurrent* in a macro tick R, denoted $ni_1 \mid_R ni_2$, iff

1) they appear in the micro ticks of R, i.e., $i_1, i_2 \leq \mathit{len}(R)$ and $n_1 = R(i_1)$, $n_2 = R(i_2)$,
2) they belong to statically concurrent threads, i.e., $\mathit{th}(n_1) \mid\mid \mathit{th}(n_2)$, and
3) their threads have been instantiated by the same fork, i.e., $\mathit{last}(n, i_1) = \mathit{last}(n, i_2)$ where $n = \mathit{lcafork}(n_1, n_2)$.

IV. Sequential Constructiveness

A. 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.

Definition 4 (Confluent/identical/effective writes): Two assignments $x = ex_1$ and $x = ex_2$ to the same variable x are *confluent* if the order in which they are executed does not matter (otherwise *non-confluent*). We call these assignments *identical* if ex_1 and ex_2 evaluate to the same value (otherwise *non-identical*). An assignment is *effective* if it changes the value of a variable (otherwise *ineffective*).[1]

Given two identical writes, the second write is ineffective. Obviously, identical writes are confluent. However, they are not the only type of confluent writes.

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

This is a generalized variant of Esterel's combination function that allows to merge concurrent emissions of valued signals, for example via addition; this also permits to emulate Esterel-style signals [3].

Definition 6 (Absolute/relative writes): 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*.

Relative writes of the same type are also confluent.

B. 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. In a nutshell, the order to be imposed by any valid run is, within all ticks t, for all variables v that are accessed concurrently within t, "any identical absolute writes on v before any confluent relative writes on v before any reads on v."

Definition 7 (SC-Admissibility, Constructiveness): A run for an SCG is *SC admissible* if, for all ticks R in this run, and for all concurrent node instances $ni_{1,2} = (R(i_{1,2}), i_{1,2})$, with $1 \leq i_1 < i_2 \leq \mathit{len}(R)$ and $ni_1 \mid_R ni_2$, none of the following *SC scheduling violations* occurs:

V1 $R(i_1)$ and $R(i_2)$ perform non-confluent writes on the same variable;

V2 $R(i_1)$ reads a variable, on which $R(i_2)$ then performs an effective write;

V3 $R(i_1)$ performs a relative write to a variable, on which $R(i_2)$ then performs an absolute write.

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.

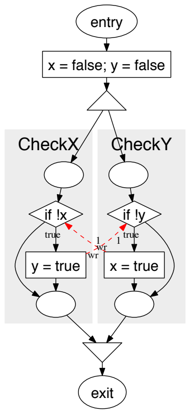
Note that an SC-admissible order cannot be found if there are concurrent, non-confluent writes.

The mere existence of an SC-admissible run does not yet guarantee determinism. A counter example is the program

[1]To be exact, these definitions imply a reference to all reachable memory configurations of all possible runs, which come into play when formally defining the semantics of SCL/SCG [3].

```
module NonDet
output bool x = false, y = false;
{
  fork    // Thread "CheckX"
    if (!x) y = true;
  par     // Thread "CheckY"
    if (!y) x = true;
  join;
}
```

(a) The SCL program



(b) The SC Graph

Fig. 3. The NonDet example, illustrating multiple admissible runs and non-deterministic outcome.

NonDet presented in Fig. 3, which has two sequentially admissible runs. Depending on which conditional 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.

V. Analyzing Sequential Constructiveness

A. Concurrent variable accesses

Definition 8 (Concurrent Nodes): Two nodes $n_1, n_2 \in N$ are *concurrent*, denoted $n_1 \mid n_2$, iff they may both execute in concurrent node instances within the same tick, i.e., there is a reachable macro tick R and $i_1, i_2 \leq len(R)$ such that $n_1 = R(i_1)$, $n_2 = R(i_2)$ and $(n_1, i_1) \mid_R (n_2, i_2)$.

We introduce the following relations on nodes in N:

- $n_1 \rightarrow_{seq} n_2$ if there is a sequential control flow edge from n_1 to n_2.
- $n_1 \leftrightarrow_{ww} n_2$ iff $n_1 \mid n_2$ and there exists a variable on which n_1 and n_2 perform non-confluent writes.
- $n_1 \rightarrow_{wr} n_2$ iff $n_1 \mid n_2$ and n_1 performs an absolute write to a variable that is read by n_2.
- $n_1 \rightarrow_{wi} n_2$ iff $n_1 \mid n_2$ and n_1 performs an absolute write to a variable on which n_2 performs a relative write (the subscript i stands for "increment").
- $n_1 \rightarrow_{ir} n_2$ iff $n_1 \mid n_2$ and n_1 performs a relative write to a variable that is read by n_2.
- $n_1 \rightarrow_{wir} n_2$ iff $n_1 \rightarrow_{wr} n_2$ or $n_1 \rightarrow_{wi} n_2$ or $n_1 \rightarrow_{ir} n_2$. This summarizes the constraints induced by concurrent write/increment/read accesses.
- $n_1 \rightarrow n_2$ iff $n_1 \rightarrow_{seq} n_2$ or $n_1 \rightarrow_{wir} n_2$, that is, if there is any ordering constraint.

Definition 9: A program is *acyclic SC (ASC) schedulable* if in its SCG (i) there are no nodes n_1, n_2 with $n_1 \leftrightarrow_{ww} n_2$, and (ii) there is no $\rightarrow$ cycle containing $\rightarrow_{wir}$ edges.

Theorem 10 (Sequential Constructiveness): Every ASC schedulable program is sequentially constructive.

Proof sketch: This follows directly from examining the constraints required by ASC schedulability and the SC-admissibility rules underlying the definition of SC, plus the observation that under ASC scheduling, all *potential* writes are scheduled before any reads, thus ensuring determinism. ■

The Control example is ASC schedulable; NonDet, however, contains a cycle that involves *wr* edges, and therefore is not ASC schedulable.

B. 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 11 (Priorities): 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.

The priority concept can also serve to determine sequential constructiveness, based on Thm. 10 and the following theorem:

Theorem 12 (Finite Priorities): A program is ASC schedulable iff (i) there are no statements n_1, n_2 with $n_1 \leftrightarrow_{ww} n_2$, and (ii) all statement priorities are finite.

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. ■

C. Computing priorities

The calculation of priorities (Def. 11) 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, we can exclude all graphs with a positive weight cycle, 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), e.g., by Tarjan's algorithm [18], 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. 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.

VI. 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.

This seemingly simple idea has turned out to be a fairly rich topic, of which only the fundamentals could be covered in the limited space available. More on how the SC MoC relates to other MoCs can be found in the full version of this paper [3]. There we also cover related work in more depth, formally define the mapping from SCL to SCG, present a formal semantics for SCL/SCG, discuss conservative approximations of SC-admissibility, examine how ASC schedulability is conservative wrt. SC, explain how Esterel/SyncChart-style pure and valued signals can be emulated with shared variables in the SC MoC, illustrate how hierarchical abortions can be mapped to SCL/SCG, discuss further synchronous language issues such as statement reincarnation and instantaneous loops, and present further examples including the combined use of absolute and relative writes.

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, Murali Parthasarathy, Marc Pouzet, and Partha Roop, who helped to improve an earlier draft of this manuscript.

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] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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, June 2009.

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] 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.

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