Submitted to Special Issue on Application of Concurrency to System Design

A

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

REINHARD VON HANXLEDEN, Kiel University
MICHAEL MENDLER and JOAQUÍN AGUADO, Bamberg University
BJÖRN DUDERSTADT, INSA FUHRMANN, and CHRISTIAN MOTIKA, Kiel University
STEPHEN MERCER and OWEN O'BRIEN, National Instruments
PARTHA ROOP, Auckland University

Synchronous languages ensure determinate concurrency, but at the price of 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 determinate concurrency. The *sequentially constructive* (SC) model of computation (MoC) presented here harnesses the synchronous execution model to achieve determinate concurrency while taking advantage of familiar, convenient programming paradigms from sequential languages.
In essence, the SC MoC extends the classical synchronous MoC by allowing variables to be read and written in any order and multiple times 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 investigate classes of shared variable accesses, define SC-admissible scheduling as a restriction of "free scheduling," derive the concept of sequential constructiveness, and present a priority-based scheduling algorithm for analyzing and compiling SC programs efficiently.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms: Theory, Languages, Algorithms

Additional Key Words and Phrases: Concurrency, constructiveness, determinacy, determinism, embedded systems, Esterel, reactive systems, synchronous languages

ACM Reference Format:
R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien and P. Roop. 2014. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (December 2014), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

This work has been supported in part by the German Science Foundation, as part of the PRETSY project (DFG HA 4407/6-1, ME 1427/6-1), and by National Instruments.
Authors' adresses: R. von Hanxleden, B. Duderstadt, I. Fuhrmann and C. Motika, Dept. of Computer Science, Kiel University, Kiel, Germany; e-mail: {rvh, bdu, ima, cmot}@informatik.uni-kiel.de; M. Mendler and J. Aguado, Dept. of Computer Science, Bamberg University, Bamberg, Germany; e-mail: {michael.mendler, joaquin.aguado}@uni-bamberg.de; S. Mercer and O. O'Brien, National Instruments, Austin, TX, USA; e-mail: {stephen.mercer, owen.o'brien}@ni.com; P. Roop, Dept. of Electrical and Electronic Engineering, Auckland University, Auckland, New Zealand; e-mail: p.roop@auckland.ac.nz.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1539-9087/2014/12-ARTA $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

One of the challenges of embedded system design is the determinate handling of concurrency. As suggested by Milner [1989], we consider a computation as *determinate* if the same sequence of inputs produces the same sequence of outputs, as opposed to *deterministic* computations, which in addition have identical internal behavior/scheduling. The concurrent programming paradigm exemplified by languages such as Java or C with Posix threads 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 determinate behavior [Hansen 1999; Lee 2006], 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. Especially for safety-critical embedded systems, as found in, e. g., medical, avionics, or automotive applications, such non-determinacy is unacceptable, as it hampers predictability and certifyability. However, even for non-safety-critical systems non-determinacy is undesirable as it makes testing and functional validation more difficult. To address this, tools and methods for determinacy analysis have been proposed recently for different concurrent programming domains [Vechev et al. 2010; Leung et al. 2012; Yuki et al. 2013; Kuper et al. 2014].

An established concurrent programming paradigm which has determinacy built into its very core is the *synchronous* model of computation (MoC) [Benveniste et al. 2003], exemplified by languages such as Esterel [Berry 2000], Quartz [Schneider 2002], Lustre [Halbwachs et al. 1991], Signal [Guernic et al. 1991] and SyncCharts [André 1996]. 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 "shared variable" in a generic sense that also encompasses streams and signals. However, race conditions are resolved by a determinate, statically-determined scheduling regime, which ensures that within a tick, (i) concurrent reads occur after writes and (ii) each shared variable is written only once (with certain exceptions, such as combination functions, as discussed later). 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 shared variables can be assigned a unique value. This provides a sufficient condition for a determinate semantics, though, as we argue here, not a necessary condition.

Introducing global synchronization barriers and sequences of reaction cycles is a sound basis for determinate concurrency and applicable also for general programming languages commonly used for embedded systems. Demanding unique variable values per tick is appropriate for, e. g., control theory and circuit behavior, which are two domains that have driven the original development of synchronous languages. However, this requirement limits expressiveness and also runs against the intuition of programmers versed in sequential programming. It also makes the task of producing a program free of "causality errors" more difficult than it needs to be. For example, consider a simple programming pattern such as `if (!done) { ... ; done = true}`, where `done` is a shared variable. This is a common pattern in Programmable Logic Controller (PLC) software, for example. The requirement of unique values per tick would produce a *causality error* because `done` is written to within the cycle *after* it is read, and because `done` would possibly be both `false` and `true` within the same tick. However, in this example, there is no race condition between the read and the write. Thus, there is no reason to reject such a program in the interest of ensuring determinate concurrency.

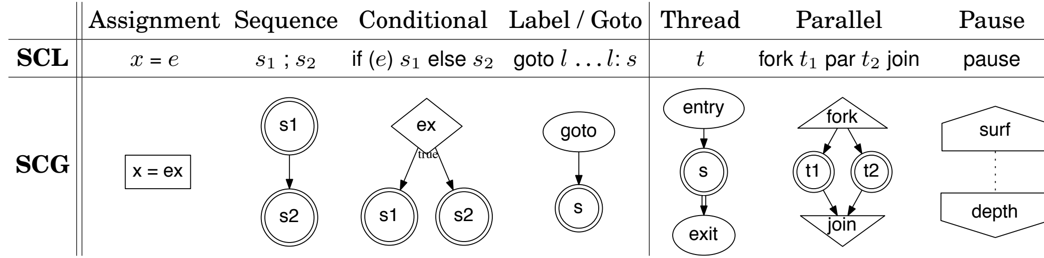


Fig. 1. The mapping between SCL statements and SCG subgraphs. Double circles are place holders for SCG subgraphs, those labeled t1/t2 are subgraphs representing threads. Solid arrows depict *seq* (sequential) edges, double arrows represent an arbitrary number of *seq*-edges. The dotted line indicates a tick edge.

Contributions. We propose the *sequentially constructive* (SC) MoC, a conservative extension to the synchronous MoC that accepts a strictly larger class of programs. Specifically, the SC MoC permits shared variables to have multiple values per tick as long as these values are 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 determinate 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 SC. However, there exist Esterel programs that are SC, but not constructive in the sense of Berry [2002], the standard notion of constructiveness which we will call *B-constructive*. E. g., all programs that do not use the concurrency operator are trivially SC, though they may be not B-constructive.

Outline. The next section presents the SC Language (SCL) and the SC Graph (SCG), which we use as a basis for the concept of sequential constructiveness. Sec. 3 presents the general scheduling problem, on which Sec. 4 builds to define sequential constructiveness. Sec. 5 presents an approach to analyze whether programs are SC and to compute a schedule for them. We discuss related work in Sec. 6, in Sec. 7 we summarize and provide an outlook. For proofs and further aspects of SC not covered here due to space constraints we refer the reader to an extended technical report [von Hanxleden et al. 2013b].

2. 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.

2.1. The SC Language

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*.

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

$$s \ ::= \ x = e \mid s \,;\, s \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid l\colon s \mid \mathsf{goto}\ l \mid \mathsf{fork}\ s\ \mathsf{par}\ s\ \mathsf{join} \mid \mathsf{pause}$$

where x is a *variable*, e 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. We include the primitive goto rather than some structured alternative as this facilitates, for example, the synthesis of state ma-

chines/statecharts. Of course, structured control flow constructs, such as loops, can be easily derived from the goto. As a syntactical detail, the conditional, as is the practice in C-like languages, omits a then keyword, but we will still refer to the two branches as *then* and *else branches*. The pause statement introduces synchrony, by deactivating a thread until the next tick commences. Parallel composition forks off two threads and terminates (joins) when both threads have terminated. In Esterel, parallel composition is denoted $\|$, and we will use this notation also for our formal treatment of concurrency; however, the SCL language uses 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. Jumps are not allowed to cross thread boundaries. The sublanguage of expressions e used in assignments and conditionals is not restricted. All we assume is a function $eval$ to evaluate e in a given memory M and return a value $v = eval(e, M)$ of the appropriate data type. However, we rule out side effects when evaluating e.

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; e. g., the conditional binds stronger 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, and initial values. A concrete program also contains an *interface* that declares inputs (set at the beginning of each tick by the environment), outputs (conveyed to the environment at the end of each tick), and input/outputs. However, as our formal development will be based on the internal representation of SCL programs as SC Graphs, we leave the concrete SCL syntax informal.

2.2. The Control Example

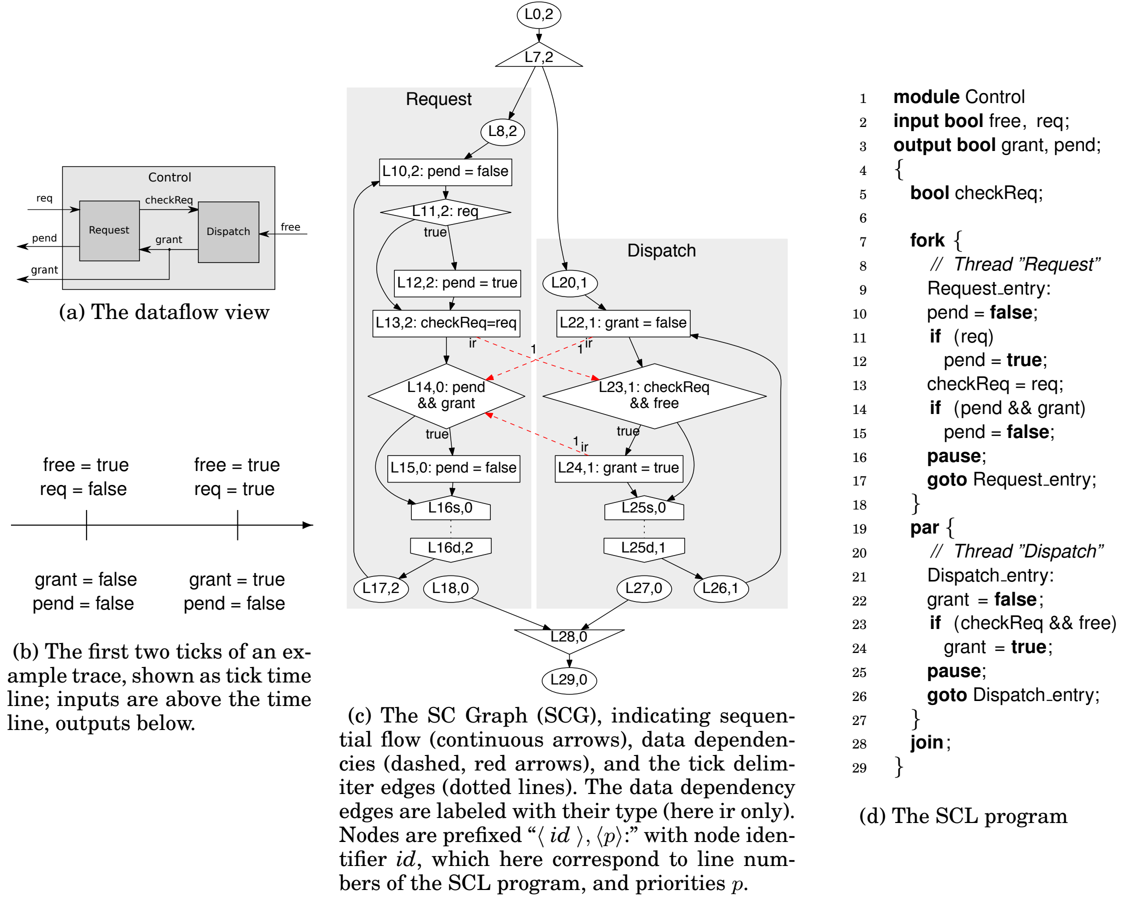
A running SCL program always executes a top-level thread referred to as Root thread. In the Control example shown in Fig. 2, Root spawns two further threads named Request and Dispatch that are concurrent to each other. Thus Control consists of three threads. The functionality of Control is inspired by PLC software used in the railway domain. It processes requests (as indicated by the req input flag) to a resource, which may be free or not. As shown 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,
(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. 2.5), is not externally observable. Hence, when reasoning about determinate behavior, we only consider the outputs emitted at the end of each macro tick (e. g., in Def. 4.8).

The execution of Control begins by launching the Root thread, which executes a fork that spawns off Request and Dispatch. These two threads then progress on their own.



(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 ir only). Nodes are prefixed "⟨ id ⟩, ⟨p⟩:" with node identifier id, which here correspond to line numbers of the SCL program, and priorities p.

(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 vars	free	***t***												t
	req	***f***												f
Output vars	grant	$\perp$								f				***f***
	pend	$\perp$				f								***f***
Local var	checkReq	$\perp$						f						f
Continuations	C_{Root}	L0	L7	[L28]										[L28]
	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	$R^a(i)$	L0	L7	L8	L10	L11	L13	L20	L22	L23	L25s	L14	L16s	

(e) An admissible run, corresponding to the first (macro) tick of the example trace (b).

Fig. 2. The Control example, illustrating the sequential modification of shared variables and the resulting scheduling under the SC MoC. "Ln" refers to the statement at line n of the SCL program. In (e), the values *true* and *false* are abbreviated as t and f. 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. Continuations denote for each thread the statement to be executed next, as further explained in Sec. 3.1. Threads may be disabled (denoted $\perp$), active, waiting (square brackets), or pausing (parentheses).

Were they Java threads, a scheduler of some run-time system could now switch back and forth between them arbitrarily, until both of them have finished. Under the SC MoC, their progression and the context switches between them are instead disciplined by a scheduling regime that prohibits race conditions. Determinacy 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) in Dispatch. Hence L13 in Request must be scheduled before L23 Dispatch.

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.

The concurrent threads in Control 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-determinacy. However, this would not be permitted under the strict synchronous, fixed-point MoC using signals, which requires unique 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. Thus, Control is not B-constructive and would be rejected by compilers for current synchronous languages. 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* (SC). The rest of this paper makes this notion precise and describes a practical strategy to analyze sequential constructiveness and to implement schedules that adhere to the SC MoC. One building block is the graph abstraction presented next.

2.3. The SC Graph

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 $e.tgt \in N$. Fig. 2c shows the SCG for Control.

Nodes and edges are further described by various attributes. A node n is labelled by the *statement type* $n.st$ that it represents, viz. $n.st \in \{$entry, exit, goto, $x = e$, if (e), fork, join, surf, depth$\}$, where x is some variable and e is some expression.[1] Nodes labelled with $x = e$ are referred to as *assignment nodes*, those with if (e) 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. 1, 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; these mainly serve to structure the SCG and could be eliminated without changing the meaning of an SCG. Fig. 1 sketches how SCG elements correspond to an SCL program, the TR [von Hanxleden et al. 2013b] describes this mapping in detail.

For a fork node n_f, we define $join(n_f) =_{\text{def}} n_j$, where n_j is the uniquely associated join node. Similarly, for a surface node n_s, we define $tick(n_s) =_{\text{def}} n_d$, where n_d is the uniquely associated depth node of n_s, also referred to as *tick successor*.

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

Every edge e has a type $e.type$ that specifies the nature of the particular ordering constraint expressed by e.

Definition 2.1 (*Edge types*). We define the following sets of edge types: ***iur*-edges** $\alpha_{iur} =_{\text{def}} \{\textit{\textbf{ww}}, \textit{\textbf{iu}}, \textit{\textbf{ur}}, \textit{\textbf{iu}}\}$, ***instantaneous edges*** $\alpha_{ins} =_{\text{def}} \{\textit{\textbf{seq}}\} \cup \alpha_{iur}$, ***arbitrary edges*** $\alpha_a =_{\text{def}} \{\textit{\textbf{tick}}\} \cup \alpha_{ins}$, and ***flow edges*** $\alpha_{flow} =_{\text{def}} \{\textit{\textbf{seq}}, \textit{\textbf{tick}}\}$.

We write $e.src \rightarrow_\alpha e.tgt$, pronounced "$e.src$ α-precedes $e.tgt$," if $e.type = \alpha$. We also write $\rightarrow_{iur/ins/flow}$ to encompass all $\rightarrow_\alpha$ with $\alpha \in \alpha_{iur/ins/flow}$.

We also write $E_{iur/ins/flow} \subseteq E$ to denote the *iur* / instantaneous / flow edges.

When $n_1 \rightarrow_{seq} n_2$ holds, we say that n_1 and n_2 are *sequential* successors. We analogously define *tick* and *flow* successors. A path consisting exclusively of flow edges is referred to as *flow path*. We use $\twoheadrightarrow_\alpha$ for the reflexive and transitive closure of $\rightarrow_\alpha$.

The *iur*-edges are derived later for the purpose of scheduling analysis (Def. 4.3).

The sequential order $n_1 \rightarrow_{seq} n_2$ expresses the usual sequential control flow. 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. When n_1 and n_2 are enclosed in a loop, an execution of n_2 might be followed by an execution of n_1 in the same tick.

The tick order $n_1 \rightarrow_{tick} n_2$ 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.

2.4. Thread Terminology

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

We denote the set of threads of G by T, which includes Root. Each thread $t \in T$ 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. 2.3) 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. We define ***fork***(t) as the fork node that immediately precedes $t.en$, and for each thread $t \neq$ Root define its immediate *parent thread* $p(t) =_{\text{def}} th(fork(t))$. In Control, it is p(Request) = p(Dispatch) = Root. Conversely, we define the *child threads* $ch(t) =_{\text{def}} \{t' \mid t = p(t')\}$.

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

Definition 2.2 (*Thread relations*). Let t, t_1, t_2 be threads in T.

(1) The set of *ancestor* threads of t is defined as the transitive closure of the parent relationship $p^*(t) =_{\text{def}} \{t, p(t), p(p(t)), \ldots, \text{Root}\}$.
(2) The set of *descendant* threads of t, the inverse ancestor relation, is defined as $ch^*(t) =_{\text{def}} \{t' \mid t \in p^*(t')\}$.
(3) t_1 is *subordinate* to t_2, written $t_1 \prec t_2$, if $t_1 \neq t_2$ and $t_1 \in p^*(t_2)$.
(4) 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 $t'_1, t'_2 \in T$ with $t'_1 \neq t'_2$, $fork(t'_1) = fork(t'_2)$, $t_1 \in ch^*(t'_1)$, and $t_2 \in ch^*(t'_2)$. We then refer to this fork node as the *least common ancestor (lca) fork*, denoted ***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 Root $\prec$ Request and Root $\prec$ Dispatch, meaning that Root can only terminate after Request and Dispatch have terminated (which they never do). It is also Request

|| Dispatch, whereas Root is not concurrent with any thread. Note that concurrency on threads, in contrast to concurrency on node instances (Def. 2.6), is purely static and can be checked with a simple, syntactic analysis of the program structure.

2.5. 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 possibly for a concrete implementation.

Definition 2.3 (*Ticks, runs*). 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 $a \in \mathbb{N}_{\geq 1}$.

Definition 2.4 (*Node instances*). A *node instance* is a pair $ni = (n, i)$ consisting of a statement node $n \in N$ and a micro tick count $i \in \mathbb{N}$.

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 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. 3, 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.

2.6. Concurrency of Node Instances

The function $last(n, i)$, defined next, is instrumental to define concurrency of node instances.

Definition 2.5 (*Last occurrences*). For a macro tick R, an index $1 \leq i \leq len(R)$, and a node $n \in N$, $last_R(n, i) = max\{j \mid j \leq i, R(j) = n\}$ retrieves the last occurrence of n in R at or before index i. If it does not exist, $last_R(n, i) = 0$.

Definition 2.6 (*Concurrent node instances*). For a macro tick R, $i_{1,2} \in \mathbb{N}_{\leq len(R)}$, and $n_{1,2} \in N$, two node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$ are *(run-time) concurrent* in R, denoted $ni_1 \mid_R ni_2$, iff

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

3. 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 flow edges $\rightarrow_{seq}$ and $\rightarrow_{tick}$. Our notion of sequential constructiveness will then arise from a further restriction of the free schedules guided by additional *iur* precedences.

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

3.1. 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, references 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, namely (i) the currently running node in the SCG and (ii) a scheduling status.

Definition 3.1 (*Continuations, continuation pools*). A *continuation* c has a node $c.node \in N$ and a status $c.status \in \{\text{active}, \text{waiting}, \text{pausing}\}$. A *continuation pool* C is a finite set of continuations.

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.

We also associate each thread with a possible *thread state*. 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*. An enabled thread t has a continuation c, we thus associate the state of c with t as well. The resulting possible thread states and the associated notation is illustrated in Fig. 3; see also Fig. 2e. There, the thread pool C_i^a in micro tick i of macro tick a consists 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 parentheses (n) are pausing and all others are active.

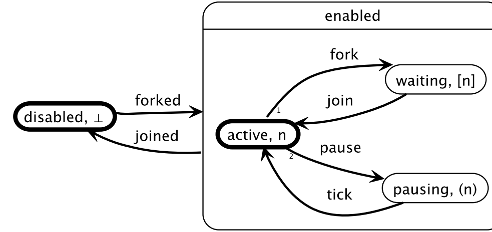


Fig. 3. Execution states of a thread, shown with a Statechart notation; initial states have a bold outline.

At every micro tick of an execution run, the scheduler picks an active continuation from a continuation pool C and executes it. Initially, C only contains the main program Root activated at its entry node Root.en. Then, every time an active fork node n_f is executed, the corresponding join node $n_j = join(n)$ is installed in the same thread as the fork. This thread is subordinate to the threads of the children spawned by the n_f, which thus block the execution of the continuation in the parent thread. In this fashion, n_j starts with status waiting until the children are terminated, at which point $n_j.status$ becomes active. The switching between active and pausing happens in the execution of the pause statement. When an active surf node n_s 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 $tick(n_s)$.

Continuation pools must satisfy some coherence properties.

Definition 3.2 (*Valid continuation pools*). A continuation pool C is called *valid* iff the following conditions are satisfied.

— The waiting continuations $c[\mathsf{waiting} :: n] \in C$ are those continuations in C that are not $\prec$-maximal in C, meaning they are subordinate to some other thread 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., $n.st = \mathsf{surf}$.
— 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 $p(th(c.node)) = th(c'.node)$. Furthermore, the parent continuation always corresponds to a waiting join statement; i. e., $c'.node.st = \mathsf{join}$ and $c'.status = \mathsf{waiting}$.

3.2. Configurations, Micro Step and Macro Step Scheduling

Definition 3.3 (*Configurations*). A *configuration* is a pair (C, M) where C is a pool of continuations and M is a memory assigning values to the variables accessed by G. A configuration is called *valid* if C is valid.

A *scheduling step* moves from the current configuration (C_{cur}, M_{cur}) to the next configuration (C_{nxt}, M_{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 $c.status =$ active and execute it in the current memory M_{cur}. This yields a new memory $M_{nxt} = \mu M(c, M_{cur})$ and a (possibly empty) set of new continuations $\mu C(c, M_{cur})$ by which c is replaced; i. e., $C_{nxt} = C_{cur} \setminus \{c\} \cup \mu C(c, M_{cur})$. 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 $\mu C(c, M_{cur}) = \emptyset$.
The actions μM and μC depend on the statement $c.node.st$ to be executed and will be defined shortly. A transition between two micro ticks is referred to as a *micro step*, written $(C_{cur}, M_{cur}) \xrightarrow{c}_{\mu s} (C_{nxt}, M_{nxt})$, where c is the continuation that is selected for execution. Since (C_{nxt}, M_{nxt}) is uniquely determined by the executed continuation c we may write it as $(C_{nxt}, M_{nxt}) = c(C_{cur}, M_{cur})$.

(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,

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

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 $V_I = [x_1 = v_1, \ldots, x_n = v_n]$ for the next macro tick. All other memory locations persist unchanged into the next macro tick. Formally,

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

A clock step is denoted $V_I : (C_{cur}, M_{cur}) \rightarrow_{tick} (C_{nxt}, M_{nxt})$, where V_I is the external input. Observe that since the set of inputs I is assumed to be fixed globally, both V_I

and M_{nxt} can be derived from each other and from M_{cur}. Hence the label V_I may be dropped.

During a macro tick (Def. 2.3), we perform a maximal sequence of micro steps of G. More concretely, the scheduler runs through a sequence

$$(C_0^a, M_0^a) \stackrel{c_1^a}{\rightarrow}_{\mu s} (C_1^a, M_1^a) \stackrel{c_2^a}{\rightarrow}_{\mu s} \cdots \stackrel{c_{k(a)}^a}{\rightarrow}_{\mu s} (C_{k(a)}^a, M_{k(a)}^a) \tag{1}$$

of micro steps obtained from the interleaving of active continuations, to reach a final quiescent configuration $(C_{k(a)}^a, M_{k(a)}^a)$, in which all continuations are pausing or waiting. We write $(C_0^a, M_0^a) \twoheadrightarrow_{\mu s} (C_i^a, M_i^a)$ to express that there exists a sequence of micro steps, not necessarily maximal, from configuration (C_0^a, M_0^a) to (C_i^a, M_i^a), dropping the information on continuations. The complete sequence (1) from start to end encompasses the macro tick, abbreviated

$$(R^a, V_I^a) : (C_0^a, M_0^a) \Longrightarrow (C_{k(a)}^a, M_{k(a)}^a). \tag{2}$$

The label V_I^a projects the initial memory, i. e., $V_I^a(x) = M_0^a(x)$ for $x \in I$. The final memory state $M_{k(a)}^a$ 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. 2.3. More precisely, $len(R^a) = k(a)$ is the length of the macro tick and R^a the function mapping each micro tick index $1 \leq i \leq k(a)$ to the node $R^a(i) = c_i^a.node$ executed at index i.

For example, in the execution run shown in Fig. 2e, the (doubly indexed) node sequence $R^a(i)$ is given by the last row "Scheduled nodes."

Note that in (2) the input label V_I^a may be dropped since it can be derived from I and M_0. When the memory state and scheduling information is not needed, it is convenient to identify a macro tick (2) with R^a as its abstraction.

We call the end points of a macro tick (2) *macro (tick) configuration*, while intermediate configurations (C_i^a, M_i^a), $0 < i < k(a)$, seen in (1) are *micro (tick) configurations*.

Definition 3.4. A *run* of an SCG G is a sequence of macro ticks R^a and external inputs V_I^a of the form (2) such that (i) the initial continuation pool $C_0^0 = \{c_0\}$ activates the entry node of the G's Root thread, i.e., $c_0.node = \mathsf{Root}.en$ and $c_0.status = \mathsf{active}$, and (ii) all macro tick configurations are connected by clock steps, i.e., $(C_{k(a)}^a, M_{k(a)}^a) \rightarrow_{tick} (C_0^{a+1}, M_0^{a+1})$.

It remains to define the actions μM and μC exercized by active continuations on memory M 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 $x = e$. In this case variable x is updated by the value of e. Formally, $\mu M(c, M)(x) = eval(e, M)$ and $\mu M(c, M)(y) = M(y)$ for $y \neq x$. In all other cases, if $c.node.st$ is not an assignment, we have $\mu M(c, M) = M$. The action of a continuation on the continuation pool is only slightly more involved. For $c[\mathsf{active} :: n] \in C$ the set $\mu C(c, M)$ is given thus:

— For $n.st \in \{\mathsf{entry}, \mathsf{goto}, x = e, \mathsf{depth}, \mathsf{join}\}$ the continuation c passes on control to its immediate sequential successor, i. e., $\mu C(c, M) = \{c[\mathsf{active} :: n']\}$, where n' with $n \rightarrow_{seq} n'$ 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., $\mu C(c, M) = \emptyset$.
— When $n.st = \mathsf{surf}$, then we set the continuation pausing to wait at this node for the next synchronous tick. i. e., $\mu C(c, M) = \{c[\mathsf{pausing} :: n]\}$.

— Consider a conditional statement $n.st = \text{if } (e)$ 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 e, so that $\mu C(c, M) = \{c[\text{active} :: n_i]\}$, where $i = 1$ if $eval(e, M) = \text{true}$ and $i = 2$ if $eval(e, M) = \text{false}$. Note that in each case $n \rightarrow_{seq} n_i$.
— Finally, suppose c instantiates a fork statement with edges $n \rightarrow_{seq} t_1.en$ and $n \rightarrow_{seq} t_2.en$ leading to the two entry nodes of its concurrent child threads t_1, t_2. Let $n_j = join(n) \in N$ be the corresponding join node. Then, $\mu C(c, M) = \{c[\text{active} :: t_1.en], c[\text{active} :: t_2.en], c[\text{waiting} :: n_j]\}$. Hence the free scheduler may execute $t_1.en$ or $t_2.en$ in any order, but both have to terminate before the join statement n_j can resume.

4. THE SEQUENTIALLY CONSTRUCTIVE (SC) MODEL OF COMPUTATION

The key to determinacy lies in ruling out any uncertainties due to an unknown scheduling mechanism. Like the synchronous MoC, the SC MoC ensures macro-tick determinacy 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. For example, 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-determinacy here.

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

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) sequentializes any concurrent variable accesses that may conflict and produce unpredictable responses. Note that it is easy to obtain determinate executions disregarding b): Simply restrict the scheduler to a globally static execution regime, e. g., by assigning each (occurrence) of a program statement some arbitrary, unique execution priority. However, this would not be transparent to the programmer and would destroy the natural parallelism of the program.

In the previous section, we defined our "playing field" for reactive control flow; sequential flow is expressed directly in the structure of the program, concurrent control flow is subject to run-time scheduling. Unlike Java etc., the SC MoC does not leave this run-time scheduling to chance, but follows certain rules, set down in this section. We start by defining different types and properties of accesses to shared variables, proceed in Sec. 4.2 by defining when we consider schedules admissible, and then define when a program is sequentially constructive (SC) in Sec. 4.3.

4.1. Types of variable accesses

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, determinate programs. For instance, it may be the case that two assignments $x = e_1$ and $x = e_2$ can be scheduled successively in any order with the same final result; this depends on the expressions e_1 and e_2 and

possibly the memory configurations in which the assignments are evaluated. When the execution order is irrelevant we call such assignments *confluent* in a given configuration, formalized later in Def. 4.4. 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 4.1 (*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 $x = f(x, e_i)$, in which the expressions e_i neither produce any side effect nor depend on x, can be executed in arbitrary order yielding a unique final value for x.

Definition 4.1 is closely related to resolution functions in VHDL, or similar operations used in parallel computing [Schwartz 1980]. Also, Esterel has a slightly restricted variant of our combination function (Esterel requires commutativity, SC does not; consider, e. g., subtraction). In Esterel, such combination functions are used to merge concurrent emissions of valued signals in a determinate way, for example via addition. Combination functions can be used as "updates" on a variable for which the final value is accumulated incrementally from concurrent source processes.

Our notion of sequential constructiveness is based on the idea that the compiler guarantees a strict "initialize-update-read" (*iur*) execution schedule during each macro tick. The *initialize* 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 avoids possibly non-determinate macro step responses.

Definition 4.2 (*Absolute/relative writes and reads*). For a combination function f, an assignment $x = f(x, e)$ where e does not reference x is a *relative write*, or an *update*, *of type* f. Other assignments are *absolute writes*, or *initializations*. Initializations $x = e$, updates $x = f(x, e)$ and conditionals if (e) are all *reads* for every variable y referenced by e.

Relative writes of the same type are confluent. As the definition of relative writes requires that e does not reference x, an assignment such as x += x − 1 is not considered a relative write, even though + is a valid combination function.

Based on this purely syntactic, structural classification of variable accesses, we define the following relations on nodes.

Definition 4.3 (*iur relations*). Given two statically concurrent accesses $n_1 \parallel n_2$ on some variable x, we define the *iur relations*

— $n_1 \rightarrow_{ww} n_2$ iff n_1 and n_2 both initialize x or both perform updates of different type. We call this a *ww conflict*.
— $n_1 \rightarrow_{iu} n_2$ iff n_1 initializes x and n_2 updates x.
— $n_1 \rightarrow_{ur} n_2$ iff n_1 updates x and n_2 reads x.
— $n_1 \rightarrow_{ir} n_2$ iff n_1 initializes x and n_2 reads x.

Since $n_1 \rightarrow_{ww} n_2$ implies $n_2 \rightarrow_{ww} n_1$, we abbreviate the conjunction of $n_1 \rightarrow_{ww} n_2$ and $n_2 \rightarrow_{ww} n_1$ with $n_1 \leftrightarrow_{ww} n_2$, and by symmetry $\rightarrow_{ww}$ implies $\leftrightarrow_{ww}$.

4.2. 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 require-

ments that a given program must fulfill to produce a determinate result, and to accept all programs for which a schedule can be found that meets these requirements. First, we formalize the notion of confluence.

Definition 4.4 (*Confluence of nodes*). Let (C, M) be a valid configuration of the SCG. Two nodes $n_{1,2} \in N$ are called *conflicting* in (C, M), if both are active in C, i. e., there exist $c_{1,2} \in C$ with $c_i.status =$ active, $n_i = c_i.node$, for $i \in \{1, 2\}$ and $c_1(c_2(C, M)) \neq c_2(c_1(C, M))$. The nodes $n_{1,2}$ are called *confluent* with each other in (C, M), written $n_1 \sim_{(C,M)} n_2$, if there is no sequence of micro steps $(C, M) \twoheadrightarrow_{\mu s} (C', M')$ such that $n_{1,2}$ are conflicting in (C', M').

Note that confluence is taken *relative* to valid configurations (C, M) 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', M') that are *reachable* from (C, M). For instance, if it happens for a given program that in all memories M' reachable from a configuration (C, M) two expressions $e_{1,2}$ evaluate to the same value, then the assignments x = e_1 and x = e_2 are confluent in (C, M). Similarly, if the two assignments are never jointly active in any reachable continuation pool C', they are confluent in (C, M), 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. 4.2, are confluent in the absolute sense, i. e., for all valid configurations (C, M) of all programs.

This relative view of confluence expressed in Def. 4.4 is useful to keep the scheduling constraints on admissible macro ticks, defined later in Def. 4.6, sufficiently weak. Notice that two nodes that are confluent in some configuration are still confluent in every later configuration reached through an arbitrary sequence of micro steps. Formally, if $(C, M) \twoheadrightarrow_{\mu s} (C', M')$ and $n_1 \sim_{(C,M)} n_2$ then $n_1 \sim_{(C',M')} n_2$. However, there may be more nodes confluent in (C', M') as compared to (C, M), simply because some conflicting configurations reachable from (C, M) are no longer reachable from (C', M'). 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.

One could make confluence in Def. 4.4 even less constraining by taking into account only those conflicts between nodes which can actually be observed by the environment. Specifically, one could consider active continuations c_1, c_2 in conflict if $c_1(c_2(C, M)) \not\approx c_2(c_1(C, M))$, 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.

Note that confluence $n_1 \sim_{(C,M)} n_2$ requires conflict-freeness for all configurations (C', M') reachable from (C, M) 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. 4.7). 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 would result in a definitional cycle.

Definition 4.5 (*Confluence of node instances*). Let R be a macro tick and (C_i, M_i), for $0 \leq i \leq len(R)$, the configurations of R. Consider two node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$ in R, i. e., $1 \leq i_{1,2} \leq len(R)$ and $n_{1,2} = R(i_{1,2})$. The node instances are called *confluent* in R, written $ni_1 \sim_R ni_2$ if $n_1 \sim_{(C_i,M_i)} n_2$, where $i = min(i_1, i_2) - 1$.

Definition 4.5 determines confluence of node instances $(n_{1,2}, i_{1,2})$ in a macro tick R relative to the configuration (C_i, M_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, M_i) although they are not confluent in the initial configuration (C_0, M_0) of the macro tick. Since the execution sequence from (C_0, M_0) to (C_i, M_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-determinacy. To this end the concurrency relation $|_R$ is now refined by the following scheduling relations on node instances to characterize potentially conflicting variables accesses:

Definition 4.6 (*Scheduling relation on node instances*). For a macro tick R with two node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$, i. e., $1 \leq i_{1,2} \leq len(R)$ and $n_{1,2} = R(i_{1,2})$, that are concurrent in R, i. e., $ni_1 \;|_R\; ni_2$, but not confluent in R, i. e., $ni_1 \not\sim_R ni_2$, we write $ni_1 \rightarrow^R_\alpha ni_2$ iff $n_1 \rightarrow_\alpha n_2$ for some $\alpha \in \alpha_{iur}$, and $ni_1 \rightarrow^R ni_2$ iff $i_1 < i_2$; i. e., ni_1 happens before ni_2 in R.

By ensuring that the execution order of concurrent statements respects the ordering constraints $\rightarrow_{iur}$, we can now implement the *iur* protocol on concurrent accesses to shared variables.

Definition 4.7 (*SC-admissibility, scheduling conditions SC$_{iur}$*). A macro tick R is *SC-admissible* iff for all node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$ in R, with $1 \leq i_{1,2} \leq len(R)$ and $n_{1,2} = R(i_{1,2})$, and for all $\alpha \in \alpha_{iur}$, it fulfills *scheduling condition* SC_α: if $ni_1 \rightarrow^R_\alpha ni_2$ then $ni_1 \rightarrow^R ni_2$.

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

Note that if there is a *ww* conflict, then SC_{ww} cannot hold, due to symmetry of $\rightarrow^R_{ww}$ and anti-symmetry of $\rightarrow^R$, and thus the run cannot be SC-admissible.

Note also that ordering absolute writes before relative writes before reads is not the only possible order to achieve determinate 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 (due to persistence, the relative writes would be relative to the last value from the previous tick), but then the relative writes would be overwritten by the absolute writes. Therefore, we consider the *iur* order prescribed above to be the most sensible and intuitive one, as it offers the programmer the greatest degree of control and expressiveness.

The run shown in Fig. 2e for the Control example 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).

4.3. Sequential Constructiveness

The notion of SC-admissibility (Def. 4.7) restricts the free scheduling defined in Sec. 3.2 to those executions which respect an *iur* regime for concurrent variable accesses unless these variable accesses are confluent. 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 determinate behavior under such SC-admissible scheduling.

Definition 4.8 (*Sequential constructiveness, reactivity, determinacy*). A program is *sequentially constructive* (SC) if (i) it is *reactive*, i. e., there exists an SC-admissible run for it, and (ii) it is *determinate*, i. e., every SC-admissible run generates the same, determinate sequence of macro ticks. More precisely, for any two SC-admissible runs

$(R^a, V_I^a) : (C_0^a, M_0^a) \implies (C_{k(a)}^a, M_{k(a)}^a)$ and $(R'^a, V_I'^a) : (C_0'^a, M_0'^a) \implies (C_{k'(a)}'^a, M_{k'(a)}'^a)$ such that $M_0^0 = M_0'^0$ are the same initial memory, and $V_I^a = V_I'^a$ are the same input sequences, the sequence of responses is identical, too, i.e., $M_{k(a)}^a = M_{k'(a)}'^a$ for all a.

The term "reactive" is chosen in analogy to "logically reactive" [Berry 2002]. Sequential constructiveness is violated when a program is not reactive, as is the case for NonReact seen in Fig. 4. Since the assignments in NonReact are ordered L7 $\rightarrow_{ir}$ L9 and L9 $\rightarrow_{ir}$ L7 in both directions, the scheduler has no chance to select one of them without violating SC_{ir} of Def. 4.7. Note that all macro tick runs of NonReact produce the same result; i. e., the program is determinate under non-SC (free) scheduling; however, programs like NonReact certainly do not represent good coding style, thus we do not mind rejecting them.

```
1  module NonReact
2  output bool u, v;
3  {
4    u = false;
5    v = true;
6    fork
7      u = v
8    par
9      v = u
10   join;
11   u = true;
12   v = true;
13 }
```

Fig. 4. NonReact is determinate, but not reactive.

```
1  module NonDet
2  output bool x, y;
3  {
4    x = false;
5    y = false;
6    fork // "CheckX"
7      if (!x)
8        y = true;
9    par // "CheckY"
10     if (!y)
11       x = true;
12   join;
13 }
```

Fig. 5. NonDet is reactive, but not determinate.

Reactivity does not imply determinacy, as illustrated by NonDet in Fig. 5. There are two SC-admissible runs in which the threads CheckX and CheckY (conditionals in L7 and L11) 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 runs are non-determinate and thus NonDet is not SC.

As a positive example, the program Control, shown in Fig. 2, is SC. 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 maximal sets of concurrent accesses, namely $\{\mathsf{L14}, \mathsf{L22}\}$ and $\{\mathsf{L14}, \mathsf{L24}\}$. The SC scheduling rule (Def. 4.7), 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 determinate outcome.

5. ANALYZING SEQUENTIAL CONSTRUCTIVENESS IN PRACTICE

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 now discuss how to abstract the concurrency and scheduling relations from node instances to static relations on nodes.

5.1. Acyclic Sequential Constructiveness (ASC)

Definition 5.1 (*SC-schedules*). For an SCG $G = (N, E)$, an *SC-schedule* Σ is a subset of G's instantaneous edges: $\Sigma \subseteq E_{ins}$. We refer to E_{ins} itself, which is derived solely by analysis of the program structure, as *structural SC-schedule*.

An SC-schedule Σ is *valid* if for every macro tick R of G which can be reached and executed under the SC-admissibility rules, if $(n_1, i_1) \rightarrow_\alpha^R (n_2, i_2)$ for some node instances $(n_{1,2}, i_{1,2})$ in R and some $\alpha \in \alpha_{ins}$ (Def. 4.6), then $(n_1 \rightarrow_\alpha n_2) \in \Sigma$.

The validity requirement of Def. 5.1 on SC-schedules Σ guarantees that the static node relations $\rightarrow_\alpha$ of Σ are a conservative over-approximation of the dynamic relations $\rightarrow_\alpha^R$ on node instances under the assumption that G is executed in an SC-admissible fashion. In contrast, the inclusion condition $\Sigma \subseteq E_{ins}$ of Def. 5.1 excludes superfluous scheduling constraints that the program does not justify.

So far, we defined a schedule as a subset of E. The schedule order, defined next, provides a concrete rule set for a scheduler that has to choose among instructions (SCG nodes) from concurrent threads.

Definition 5.2 (*Schedule order*). For a valid SC-schedule Σ, the *schedule order* $\twoheadrightarrow_{ins}^{\Sigma}$ is defined such that $n_1 \twoheadrightarrow_{ins}^{\Sigma} n_2$ iff (i) $n_1 \parallel n_2$ and (ii) Σ contains a path from n_1 to n_2 that includes an *iur*-edge.

Condition (ii) captures the case that there is an *iur* scheduling constraint that requires that n_1—or a sequential successor of n_1—must be scheduled before n_2—or a sequential predecessor of n_2. Thus, to enforce the *iur* protocol among concurrent threads, it suffices to always execute $\twoheadrightarrow_{ins}^{\Sigma}$-minimal nodes; i. e., if $n_{1,2}$ are eligible for execution and $n_1 \twoheadrightarrow_{ins}^{\Sigma} n_2$ holds, then n_1 must be scheduled before n_2.

Note that (ii) is conservative in that it may also impose a scheduling order between nodes if they are not run-time concurrent. We choose this conservative definition to be compatible with the priority-based scheduling scheme introduced in Sec. 5.2.

A less conservative, *thread-instance aware* definition of schedule order would for example not consider paths that include $lcafork(n_1, n_2)$, since at run time, executing $lcafork(n_1, n_2)$ would preclude that the node instances corresponding to $n_{1,2}$ could be run-time concurrent (consider (3) in Def. 2.6).

LEMMA 5.3 (STRUCTURAL SC-SCHEDULE IS VALID). *E_{ins} is a valid SC-schedule.*

This lemma, which follows directly from Def. 4.6, gives us a means to infer a valid SC-schedule with simple, structural program analysis. However, a valid schedule may still contain conflicting orderings that cannot be satisfied or where it depends on the capabilities of the compiler or the run-time system whether it can be implemented. Thus, we also introduce the following classifications.

Definition 5.4 (*Schedule / program classes*). An SC-schedule Σ is *acyclic* if it does not contain any cycle; Σ is *iur-acyclic* if it does not contain any cycle that contains edges induced by $\rightarrow_{iur}$. A program for which a valid (*iur*-) acyclic SC-schedule exists is *acyclic* (*iur-acyclic*) *SC*, abbreviated *ASC* (*IASC*).

A program for which the structural SC-schedule E_{ins} is (iur-)acyclic is *structurally (iur-)acyclic SC*, abbreviated *SASC* (*SIASC*).[2]

THEOREM 5.5 (SEQUENTIAL CONSTRUCTIVENESS). *Every IASC program (and thus every ASC program) is sequentially constructive.*

The proof, provided in the TR [von Hanxleden et al. 2013b], adapts the argument by Keller [1975] to prove global confluence from local confluence by parameterizing it to the set of admissible schedules specified by the *iur* protocol.

We have the implications SASC $\implies$ SIASC $\implies$ IASC $\implies$ SC and SASC $\implies$ ASC $\implies$ IASC. Thus, SASC is the strongest condition and most conservative approximation of SC. To determine whether a program is SASC or not requires minimal, only structural analysis. We therefore propose that any compilation technique developed for the SC MoC should at least be able to accept all SASC programs; conversely, writing programs such that they are SASC ensures maximal portability. However, this precludes some features like instantaneous loops, as permitted by SIASC, even if the loops are provably finite, thus there is a trade-off to make. The priority-based analysis

[2]This terminology has evolved slightly. In [von Hanxleden et al. 2013a; von Hanxleden et al. 2013b], "ASC schedulable" denoted IASC. Furthermore, "wir" (write-increment-read) was used there instead of "iur." In [von Hanxleden et al. 2014], where E_{ins} is the only considered schedule, the term "ASC schedulable" was used to denote SASC.

and execution scheme proposed in Sec. 5.2 is indeed not restricted to SASC programs, but can handle all SIASC programs.

One may also relax the sequential order to only order non-confluent statements. This leads to the class of *data-flow acyclic* programs, which is not further elaborated on here.

For the Control example of Fig. 2, 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 write-read precedences on variable grant, and L23 $\not\sim$ L13 as a write-read precedence on checkReq. Hence, by forcing the execution to respect the orderings L22 $\rightarrow_{ir}$ L14, L13 $\rightarrow_{ir}$ L23 and L24 $\rightarrow_{ir}$ L14, specified by the dashed arrows in Fig. 2, we avoid the only possible scheduling violation (SC_{ir}) expressed in Def. 4.7. Since these constraints, when added to the program order, do not introduce a causality cycle, we have a valid *iur*-acyclic SC-schedule Σ in the sense of Def. 5.4. Thus Control is ASC. This not only ensures SC-admissible execution but also, by Thm. 5.5, determinate macro step responses. Control is in fact SASC since Σ is identical to the instantaneous edges E_{ins}, which is acyclic.

5.2. Determining SC-schedules with priorities

As explained in Sec. 5.1, the schedule order $\twoheadrightarrow^{\Sigma}_{ins}$ for a valid schedule Σ provides a (conservative) means to execute SC programs according to the SC MoC. One means to implement $\twoheadrightarrow^{\Sigma}_{ins}$ are *priorities*, defined next. Conceptually, priorities are similar to (inverse) logical time stamps, where $\rightarrow_{iur}$ corresponds to "happens before" [Lamport 1978]. We here use the term "priority" as it shall be used by a *priority-based scheduler*, which we define such that always gives control to the thread with highest priority, chosen from the set of threads that are still active in the current tick.

Definition 5.6 (*Priorities*). Given a valid SC-schedule Σ, the *priority* $n.pr$ of a statement $n \in N$ is the maximal number of $\rightarrow_{iur}$ edges traversed by any path in Σ that originates in n.

The following lemma follows from the observation that in $\twoheadrightarrow^{\Sigma}_{ins}$, only *iur*-edges can involve concurrent threads.

LEMMA 5.7 (PRIORITIES IMPLEMENT THE SCHEDULE ORDER). *For a priority assignment according to some SC-schedule Σ, for any two run-time (and hence also statically) concurrent statements $n_{1,2} \in N$, $n_1 \twoheadrightarrow^{\Sigma}_{ins} n_2$ implies $n_1.pr > n_2.pr$.*

A priority-based scheduler never allows a statement that is ready for execution to wait on another statement with lower priority. Such a scheduler implements a valid schedule, as can be verified from the SCG construction. For example $n_1 \rightarrow_{iu} n_2$ implies $n_1 \rightarrow_{iur} n_2$, 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 initializations are performed before concurrent updates to the same variable. Similarly $\rightarrow_{ir}$ ensures that initializations are performed before concurrent reads of the same variable, and $\rightarrow_{ur}$ ensures that updates are performed before concurrent reads of the same variable. The priority concept can also serve to determine sequential constructiveness, based on Thm. 5.5 and the following theorem:

THEOREM 5.8 (FINITE PRIORITIES). *A program is IASC iff there exists a valid SC-schedule such that all statement priorities are finite.*

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

In principle, we might invoke the priority-based scheduler before every single statement execution. However, for a practical implementation, it is worthwhile to consider when a scheduler must actually be invoked and when an executing thread might just keep executing without invoking the scheduler. The way we (conservatively) defined our priorities, the statements executed within a tick always execute in non-increasing priority order. (A more permissive priority assignment might, under some circumstances, permit the priority of a fork to be lower than its children, corresponding to the thread-instance aware schedule order discussed along Def. 5.2.) Therefore a thread t currently executing with some priority pr, meaning that its priority is at least as high as any other thread currently eligible for execution, cannot be preempted by another thread u with higher priority unless t just yields by lowering its own priority below u's priority. Thus, the only points when a scheduler is called for are 1) when a thread lowers its own priority, or 2) when threads are forked and the scheduler has to schedule one of the forked threads, or 3) when a thread finishes for the current tick, that is, it reaches a pause statement or terminates.

5.3. Determining IASC and computing priorities

Given a valid SC-schedule Σ, which can be either E_{ins} or a subset thereof depending on the analysis capabilities of the compiler, the calculation of priorities (Def. 5.6) can be formulated as a longest weighted path problem. We assign to each edge $e \in \Sigma$ a weight $e.w$, with $e.w = 0$ iff $e.src \rightarrow_{seq} e.tgt$, and $e.w = 1$ iff $e.src \rightarrow_{iur} e.tgt$. As the relations $\rightarrow_{iur}$ and $\rightarrow_{seq}$ exclude each other, 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 determinacy. 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 $\rightarrow_{iur}$ edge, which would mean that the program is not IASC. Thus we can compute priorities efficiently as follows:

(1) Detect whether Σ has a positive weight cycle. We can do so by computing the Strongly Connected Components (SCCs), for example using the algorithm of Tarjan [1972], and checking if any SCC contains a node that is connected to another node within the same SCC by a $\rightarrow_{iur}$ edge.
(2) If a positive weight cycle exists, then Σ is not *iur*-acyclic (Def. 5.4); 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 Σ.
(4) Compute for each $n_{SCC} \in N_{SCC}$ the maximum weighted length (priority) $n_{SCC}.pr$ of any path originating in n_{SCC}, e. g., 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.

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

6. RELATED WORK

Edwards [2003] and Potop-Butucaru et al. [2007] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages and classical work such as the Program Dependence Graph (PDG) [Ferrante et al. 1987]. 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, like Quartz, provides determinate 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* [Brzozowski and Seger 1995] under the non-inertial delay model, which can be fully decided using ternary Kleene algebra [Mendler et al. 2012]. This makes the work on causality analysis of cyclic circuits by Malik [1994] applicable to constructiveness analysis of (instantaneous) Esterel program. This has been extended by Shiple et al. [1996] to state-based systems, as induced by Esterel's pause operator, thus handling non-instantaneous programs as well. The algebraic transformations proposed by Schneider et al. [2005] increase the class of programs considered constructive by permitting different levels of partial evaluation. However, none of these approaches separates initialisations and updates or permits sequential writes to a shared variable within a tick, as we do here. The notion of sequential constructiveness introduced here is weaker regarding schedule insensitivity, but more adequate for the sequential memory models available for imperative languages. Various other approaches with their own admissible scheduling schemes have been considered. 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 [1991], Boussinot [1998] and Berry [2002], which we refer to as P, L, and B-constructiveness, respectively. We find that {S, P, L}-constructiveness are incomparable with each other, but all three include B-constructiveness. None of {P, L, B}-constructiveness considers sequential control flow as SC does. Thus we believe that in particular for software-based reactive system, S-constructiveness is more practical than the alternatives.

The standard synchronus MoC distinguishes between (local) variables and (shared) signals. 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*. The SC MoC adopts and slightly generalizes this concept of a combination function, and considers such assignments via a combination function as a *relative write*. Esterel signals can be coded in SCL using variable accesses (described further in the TR [von Hanxleden et al. 2013b]). Finally, Esterel also has the concept of *variables* that can be modified sequentially within a tick. However, they cannot be shared among concurrent threads. 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 determinacy.

Lustre, like Signal, 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 [Pouzet and Raymond 2010]. Caspi et al. [2009] 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, like the original Lustre, they adhere to the current synchronous model of execution in that they do not harness sequentiality to permit and order multiple writes.

Our work on SC exploits, and thus also depends on, the implementation of sequential composition as expressed by ";" at the target platform. This deviates from existing synchronous languages such as Esterel and Quartz which do not assume that statements separated by ";" are executed in strict sequence. In fact, in causal (pure) Esterel and Quartz programs sequential composition ";" can be replaced by parallel composition $\|$ without affecting the macro tick response. Recently, Mandel et al. [2013] and Gemünde et al. [2013] applied clock refinement to provide micro-level sequencing within an outer macro-clock. In this way, sequences of accesses to the same variable but from concurrent threads can be bundled within a single tick. This also increases sequential expressiveness but in an orthogonal fashion compared to our approach, which does not use an explicit clock to achieve sequential updating but instead takes it for granted just like a C/Java programmer does.

There have been several proposals that extend C or Java with concurrency constructs. However, these typically induce race conditions for shared variables, as addressed for example by Yu et al. [2012] for OpenMP. Some of these proposed concurrency extensions avoid race conditions by building on synchronous programming principles. Reactive C by Boussinot [1991] is an extension of C that employs the concepts of ticks and preemptions, but does not provide true concurrency and has an interpreted implementation. ECL extends C as well with Esterel-like reactive constructs [Lavagno and Sentovich 1999]; ECL programs are compiled into a reactive part (Esterel) and a data-part (C), plus some "glue logic". FairThreads [Boussinot 2006] are an extension introducing concurrency via native threads. Synchronous C, a.k.a. SyncCharts in C [von Hanxleden 2009], augments C with synchronous, determinate concurrency and preemption. It provides a coroutine-like thread scheduling mechanism, with thread priorities that have to be explicitly set by the programmer. Synchronous C is a possible synthesis target for SC programs, where the algorithm presented in Sec. 5.2 can be used to automatically synthesize thread priorities. PRET-C [Andalam et al. 2009] also provides determinate reactive control flow; however, PRET-C assumes fixed priorities per thread, thus could not execute SC programs that require back-and-forth context switching between threads. Even more restrictive is the synchronous approach ForeC [Yip et al. 2013] for multi-core execution which does not permit any communication during a tick at all. SHIM [Tardieu and Edwards 2006] provides concurrent Kahn process networks with CSP-like rendezvous communication [Hoare 1985] and exception handling. Céu [Sant'Anna et al. 2013] schedules threads determinstically, based on their textual order, and provides shared variables, with compile-time warnings when they may be modified concurrently. None of these language proposals embeds the concept of Esterel-style constructiveness into shared variables as we do here. As far as these language proposals include signals, they come as "closed packages" that do not, for example, allow to separate initialisations from updates.

The concept of sequential constructiveness can be applied not only to software and not only to textual C/Java-like languages, but also to hardware and to graphical formalisms such as Statecharts [Harel 1987]. In fact, the development of a semantically sound, yet flexible and intuitive Statechart dialect for FPGA synthesis was the original motivation for developing the SC MoC. We have developed such a Statechart dialect, named Sequentially Constructive Statecharts (SCCharts) [von Hanxleden et al.

2014], to be used for the development of safety-critical embedded systems in an industrial setting. The core semantic concepts of SCCharts are analogous to SyncCharts of André [1996], which can be viewed as a graphical variant of Esterel. 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 [Colaço et al. 2005; 2006]. The main difference between SCCharts and SyncCharts (including those present in SCADE) is that SCCharts are not restricted to constructiveness in the sense of Berry [2002], but relax this requirement to sequential constructiveness. SCCharts, and with them SCL/SCG as intermediate representations, have been implemented as part of the KIELER[3] environment.

Regarding the compilation of SCL programs (and SCCharts), two alternative code generation strategies have been proposed [von Hanxleden et al. 2014]. The first approach, termed *data-flow approach*, transforms SCL programs first into static single assignment form [Appel 1998] to handle multiple assignments within a tick and then produces a netlist, which can be either mapped to hardware, or can be simulated in software. This approach can compile all data-flow acyclic programs (see Sec. 5.1). The second, *priority-based* approach permits instantaneous loops and is primarily suitable for software synthesis; it can compile all IASC programs.

An extended abstract of this work was presented at the DATE conference [von Hanxleden et al. 2013a]. Since that presentation, we have refined the concept of admissibility, and in particular have sharpened the notion of ineffective writes to confluent writes. We also improved the definition of acyclic schedulability which was informal in [von Hanxleden et al. 2013a] and based it on the structural SC scheduling relation rather than the more general notion of valid SC-schedules.

7. 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 determinacy when shared variables are accessed concurrently, and combines this with the flexibility and familiarity of sequential programming. The SC MoC builds on ideas from synchronous programming such as the fork. . . par. . . join construct and global clock synchronization through the pause statement. By exploiting the inherent sequential program order we can compile more programs than existing synchronous programming languages without losing determinacy. 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 determinate concurrency it provides without giving up familiar, safe programming patterns that are merely sequential.

This seemingly simple idea has turned out to be a fairly rich topic, of which only the fundamentals are covered here. More on how the SC MoC relates to other MoCs can be found in the TR [von Hanxleden et al. 2013b]. There we also explain how Esterel/SyncChart-style pure and valued signals can be emulated with shared variables in the SC MoC, illustrate how hierarchical aborts can be mapped to SCL/SCG, and present further examples including the combined use of absolute and relative writes. Another interesting question is how the SC MoC domain relates to other synchronous models of computation with respect to what class of programs are considered admissible. We have proven elsewhere that B-constructiveness implies SC [Aguado et al. 2014], using a functional/algebraic formalization of SC, rather than the oper-

[3] http://www.informatik.uni-kiel.de/rtsys/kieler/

ational formalization presented here. However, there are other models of computation that we plan a comparison with in the future. For instance, the execution model of PRET-C [Andalam et al. 2010] with its statically scheduled execution of parallel threads can be accommodated in the SC model by constraining the free schedules by an additional thread sequentialization order.

Considering Esterel-like languages with signal-based communication and synchronization, SC empowers shared variables to directly model signals. This helps to close the gap between programming language and implementation language. Thus SC not only enlarges the class of constructive synchronous programs, but should also help in the compilation of Esterel-like languages, which is one area we plan to investigate in the future. Conversely, the compilation of SC programs can also benefit from existing work on Esterel compilation that, e. g., also permits the compilation of cyclic programs as long as they are constructive [Potop-Butucaru et al. 2007]. A related area of practical interest is how to extend the static analysis capabilities of the compiler to enlarge the class of accepted programs, and how best to give feedback to the user in case a compiler rejects a program because it cannot schedule it.

So far, we have mainly considered how to express Esterel concepts, notably signals, into SCL. This is natural in that SCL, with its explicit, programmer-accessible initialization and updates of variables, provides more elementary concepts than Esterel does. However, one might also investigate the other direction, i. e., how to transform an SCL program into an equivalent Esterel program. One approach to do so would be to develop an SSA transformation that is aware of sequential program orderings as well as the *iur* protocol. This approach might not only allow to harness existing Esterel compilation work, but could also lead to an alternative semantic grounding of SC.

We here presented the semantic principles of SC based on an interleaved, sequential scheduling of concurrent threads. However, this sequential model still offers room for parallel execution. This is already illustrated by the aforementioned mapping to hardware. Another natural application area would be multi-core software execution. To that end, we plan to further explore scheduling strategies that leave additional room for parallelism, for example based on PDGs [Ferrante et al. 1987]. For example, one might permit to revert the execution order of sequential statements if they are confluent, as is routinely done by out-of-order execution platforms. We also consider alternative definitions of the SC semantics, for example, as tree rewriting rules, thus reflecting the tree structure of the continuation pools.

ACKNOWLEDGMENT

The material presented here has benefited greatly from discussions with Hugo Andrade, Gérard Berry, Stephen Edwards, Jeff Jensen, Louis Mandel, Murali Parthasarathy, and Marc Pouzet. We are also grateful for the valuable suggestions made by our anonymous reviewers.

REFERENCES

Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. 2014. Grounding Synchronous Deterministic Concurrency in Sequential Programming. In *Proceedings of the 23rd European Symposium on Programming (ESOP'14), LNCS 8410*. Springer, Grenoble, France, 229–248.

Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. 2009. PRET-C: A new language for programming precision timed architectures. *Workshop on Reconciling Performance with Predictability (RePP'09), Embedded Systems Week*, Grenoble, France.

Sidharta Andalam, Partha S. Roop, and Alain Girault. 2010. Deterministic, predictable and light-weight multithreading using PRET-C. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*. Dresden, Germany, 1653–1656.

Charles André. 1996. *SyncCharts: A Visual Representation of Reactive Behaviors*. Technical Report RR 95–52, rev. RR 96–56. I3S, Sophia-Antipolis, France.

Andrew W. Appel. 1998. SSA is functional programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, Vol. 91. IEEE, Piscataway, NJ, USA, 64–83.

Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press, Cambridge, MA, USA, 425–454.

Gérard Berry. 2002. *The Constructive Semantics of Pure Esterel*. Draft Book, Version 3.0, Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France.

Frédéric Boussinot. 1991. Reactive C: An Extension of C to Program Reactive Systems. *Software Prac. Experience* 21, 4 (1991), 401–428.

Frédéric Boussinot. 1998. *SugarCubes Implementation of Causality*. Research Report RR-3487. INRIA.

Frédéric Boussinot. 2006. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience* 18, 5 (April 2006), 445–469.

Janusz A. Brzozowski and Carl-Johan H. Seger. 1995. *Asynchronous Circuits*. Springer-Verlag, New York.

Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. 2009. Synchronous Objects with Scheduling Policies: Introducing Safe Shared Memory in Lustre. In *ACM Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. ACM, Dublin, Ireland, 11–20.

Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*. ACM, Seoul, South Korea, 73–82.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2005. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*. ACM, New York, NY, USA, 173–182.

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

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

Mike Gemünde, Jens Brandt, and Klaus Schneider. 2013. Clock refinement in imperative synchronous languages. *EURASIP J. Emb. Sys.* 2013 (2013), 3.

Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. 1991. Programming real time applications with SIGNAL. *Proc. IEEE* 79, 9 (Sept. 1991), 1321–1336.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data-flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320.

Per Brinch Hansen. 1999. Java's insecure parallelism. *SIGPLAN Not.* 34, 4 (April 1999), 38–45.

David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274.

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

Robert M. Keller. 1975. A fundamental theorem of asynchronous parallel computation. In *Parallel Processing*, Tse-yun Feng (Ed.). Lecture Notes in Computer Science, Vol. 24. Springer Berlin, Heidelberg, 102–112.

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Principles of Programming Languages (POPL'14)*. ACM, New York, USA, 257–270.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.

Luciano Lavagno and Ellen Sentovich. 1999. ECL: a specification environment for system-level design. In *Proc. 36th ACM/IEEE Conf. on Design Automation (DAC'99)*. ACM Press, New York, NY, USA, 511–516.

Edward A. Lee. 2006. The Problem with Threads. *IEEE Computer* 39, 5 (2006), 33–42.

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *Programming Language Design and Implementation PLDI 2012*. ACM, New York, USA, 383–394.

Sharad Malik. 1994. Analysis of Cyclic Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 7 (July 1994), 950–956.

Louis Mandel, Cédric Pasteur, and Marc Pouzet. 2013. Time refinement in a functional synchronous language. In *ACM SIGPLAN Int. Symp. on Principles and Practice of Declarative Programming (PPDP'13)*. ACM, New York, NY, USA, 169–180.

Michael Mendler, Thomas R. Shiple, and Gérard Berry. 2012. Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design* 40, 3 (2012), 283–329.

R. Milner. 1989. *Communication and Concurrency*. Prentice Hall.

Amir Pnueli and M. Shalev. 1991. What is in a Step: On the Semantics of Statecharts. In *Proc. Int. Conf. on Theoretical Aspects of Computer Software (TACS'91)*. Springer, London, UK, 244–264.

Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel*. Springer.

Marc Pouzet and Pascal Raymond. 2010. Modular static scheduling of synchronous data-flow networks - An efficient symbolic representation. *Design Autom. for Emb. Sys.* 14, 3 (2010), 165–192.

Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe system-level concurrency on resource-constrained nodes. In *Proc. 11th ACM Conf. on Embedded Networked Sensor Systems (SenSys '13)*. ACM, New York, NY, USA, Article 11, 14 pages.

Klaus Schneider. 2002. Proving the Equivalence of Microstep and Macrostep Semantics. In *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*. Springer-Verlag, London, UK, 314–331.

Klaus Schneider, Jens Brandt, Tobias Schüle, and Thomas Türk. 2005. Improving Constructiveness in Code Generators. In *Int'l Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, Florence Maraninchi, Marc Pouzet, and Valérie Roy (Eds.). ENTCS, Edinburgh, Scotland, UK, 1–19.

Jacob T. Schwartz. 1980. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 484–521.

Thomas R. Shiple, Gérard Berry, and Hervé Touati. 1996. Constructive Analysis of Cyclic Circuits. In *Proc. European Design and Test Conference (ED&TC'96), Paris, France*. IEEE Computer Society Press, Los Alamitos, California, USA, 328–333.

Olivier Tardieu and Stephen A. Edwards. 2006. Scheduling-Independent Threads and Exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (EMSOFT'06)*. ACM, Seoul, South Korea, 142–151.

Robert E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1, 2 (1972), 146–160.

Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. 2010. Automatic Verification of Determinism for Structured Parallel Programs. In *Static Analysis (SAS 2010 (LNCS)*, R. Cousot and M. Martel (Eds.), Vol. 6337. Springer, 455–471.

Reinhard von Hanxleden. 2009. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT'09)*. ACM, Grenoble, France, 225–234.

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*. ACM, Edinburgh, UK.

Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. 2013a. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*. IEEE, Grenoble, France, 581–586.

Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. 2013b. *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*. Technical Report 1308. Christian-Albrechts-Universität zu Kiel, Department of Computer Science. ISSN 2192-6247.

Eugene Yip, Partha S Roop, Morteza Biglari-Abhari, and Alain Girault. 2013. Programming and Timing Analysis of Parallel Programs on Multicores. In *13th International Conference on Application of Concurrency to System Design (ACSD)*. 167–176.

Fang Yu, Shun-Ching Yang, Farn Wang, Guan-Cheng Chen, and Che-Chang Chan. 2012. Symbolic Consistency Checking of OpenMp Parallel Programs. In *Proceedings of the 13th ACM Int'l Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*. ACM, 139–148.

Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *Principles and Practice of Parallel Programming (PPoPP 2013)*. ACM, New York, USA, 23–34.

Received Month Year; revised Month Year; accepted Month Year