2018 18th International Conference on Application of Concurrency to System Design

# On Reconciling Concurrency, Sequentiality and Determinacy for Reactive Systems—A Sequentially Constructive Circuit Semantics for Esterel

Alexander Schulz-Rosengarten,
Steven Smyth and Reinhard von Hanxleden
*Kiel University*
*Department of Computer Science*
*Kiel, Germany*
*{als,ssm,rvh}@informatik.uni-kiel.de*

Michael Mendler
*Bamberg University*
*Faculty of Information Systems and Applied Computer Sciences*
*Bamberg, Germany*
*michael.mendler@uni-bamberg.de*

***Abstract*—A classic challenge in designing reactive systems is how to reconcile concurrency with determinacy. Synchronous languages, such as Esterel, SyncCharts or SCADE, resolve this by providing a semantics that does not depend on run-time scheduling decisions. Esterel's circuit semantics is grounded in physics: An Esterel program is considered valid *(constructive)* iff it corresponds to a delay-insensitive circuit. The circuit semantics provides on the one hand a mathematically grounded semantics, based on constructive logic, on the other hand it gives a direct path to a data-flow style software implementation. However, Esterel's constructive semantics entails a rather restricted regime for handling sequential accesses to shared data. Thus many programs are rejected as being non-constructive, even though they have a very natural, determinate interpretation. We here present a *sequentially constructive circuit semantics* (SCC) that lifts this restriction, by distinguishing sequential and concurrent execution contexts. This permits an imperative style familiar to programmers versed in C, for example, without leaving the sound physical foundation of synchronous programming.**

***Index Terms*—Reactive systems, determinacy, synchronous programming, sequential constructiveness, Esterel, circuit semantics**

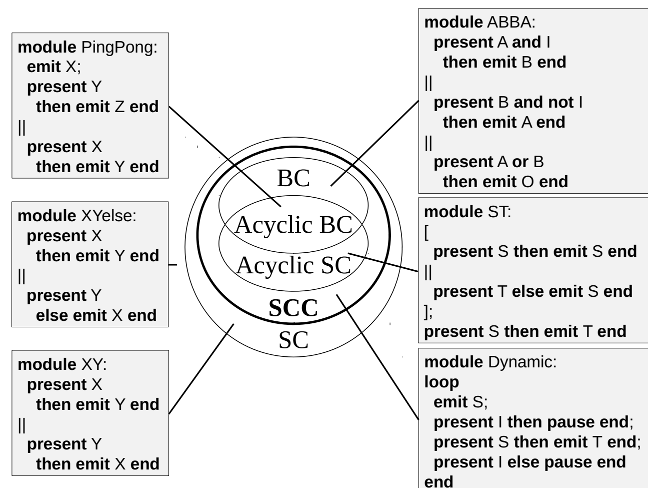



Figure 1: The class of programs considered valid under the Sequentially Constructive Circuit (SCC) semantics, proposed here, in relation to other program classes. "Acyclic SC" refers to programs that can be scheduled statically. Most Esterel compilers handle Acyclic BC (Berry Constructive) [3]. Esterel v5 handles all of BC [4]. The SCEst2SCL compiler handles Acyclic SC [5]. Our work extends compilation to SCC, that is SC (Sequentially Constructive) without "speculation."

## 1. Introduction

Synchronous programming languages [1] reconcile concurrency with determinate behavior with a semantics that abstracts from execution time. The execution of a program is divided into (logical) *ticks*, or *instants*/*reactions*. In each tick, (sensor) inputs are read from an environment and (actuator) outputs are written to the environment. The *synchrony hypothesis* states that for each tick, outputs are synchronous with inputs. This is traditionally reflected in the requirement that shared variable values are unique throughout a tick. This is a natural requirement for hardware design, where each wire must assume a unique value for each clock tick. However, this seems unduly restrictive from the perspective of imperative programming, where it is quite natural to read a variable and subsequently write a different value to it. This has recently motivated the *sequentially constructive* model of computation (SC MoC), which allows shared variables values to change within a reaction as long as the result is still determinate and does not depend on run-time scheduling choices [2]. More specifically, a *run* is considered *SC-admissible* if it adheres to certain restrictions concerning the access to shared variables, in particular that writes occur before reads; a program is considered SC if it allows SC-admissible runs for all possible input sequences, and if all such runs lead to the same result.

**Motivation.** There are different approaches to the work presented here. First, concerning the SC model of com-

978-1-5386-7013-2/18/$31.00 ©2018 IEEE
DOI 10.1109/ACSD.2018.00018

IEEE computer society

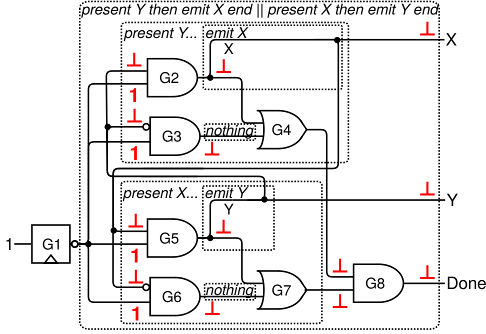



Figure 2: Circuit for XY, which is not constructive

putation, the SC definition based on runs is somewhat unsatisfactory from Esterel's constructiveness point of view in that it may be considered overly generous and does not correspond to constructive circuits. As noted in the original SC proposal [2], it accepts programs that in the traditional synchronous sense are considered "speculative." Fig. 1 presents the program classes discussed in this section, with examples. Consider the minimalistic XY Esterel example in the lower-left of Fig. 1. Its output [1] consists of the *signals* X and Y, which are *present* if and only if they are emitted by an emit statement; otherwise they are *absent*. XY consists of two parallel *threads*, where the first emits Y if X is present and the second emits X if Y is present. (For readers not familiar with Esterel, a brief summary of the language follows in Sec. 2.1.) The software view at XY is that a scheduler may choose between first testing X or first testing Y, but in both schedules the end result will be the same, namely both signals absent at the end of the tick. Thus XY is SC. However, either schedule requires a "leap of faith" when doing the first test, of X or of Y, by assuming that the tested signal will not be emitted by the other thread later. The hardware view of XY exposes this, as can be seen in the circuit in Fig. 2 that has been constructed according to Berry's circuit semantics for Esterel [6]. Gates G2 and G5 form a cycle. Thus some wires are known to be high (labeled 1, corresponding to "present"), but most stay unknown ($\perp$) according to ternary fixed point analysis [4], [7]. Very briefly, constructive logic differs from standard Boolean logic in that variables/wires may not only be 1 or 0 but also $\perp$, and there is no "law of excluded middle." Thus, under constructive logic the equation $S = S \vee \neg S$ yields $S = \perp$, not $S = 1$. This nicely corresponds to the fact that a circuit for $S = S \vee \neg S$ is not guaranteed to stabilize but, for some gate and wire delays, may oscillate forever. For the XY circuit, this means that we cannot guarantee unique stabilization, which is indeed what we mean by speculative behavior. Here, there are two possible stable states for this circuit, one with both X and Y considered absent and one with both considered present. One aim of this paper is to rule out such cases and to **define a notion of SC that has a firm physical grounding**, without speculation.

1. For conciseness, listings omit input/output declarations and end tokens for module and signal declarations.

Beyond this language-theoretic motivation, which—relative to the original SC proposal—results in a more restricted notion of what is considered acceptable, we are on the other hand concerned with enlarging the class of SC programs that can be handled in practice. The current definition of SC achieves the goal of a determinate semantics, but is not a viable basis for compile-time analysis of whether a program is SC or not. To check whether a program is SC would require an exhaustive construction of all SC-admissible runs, for all possible input sequences, for example using a mechanism based on backtracking; then one would have to check that for all possible input sequences, all runs lead to the same result. Thus compilers for languages based on the SC MoC, such as SCCharts [8] or SC Esterel (SCEst) [5], [8], so far only accept a subset of the SC MoC, namely those programs where scheduling constraints induced by control flow and shared variables are statically acyclic. One such program is ST shown on the middle-right in Fig. 1, where S is (re-)emitted if S is present in parallel to an emission of S if T is absent (S depends on T). Then T is emitted if S is present (T depends on S). There is a cyclic signal dependency between S and T, however, under SC this cycle is broken by the sequential statement order.

The restriction to acyclic dependencies and the requirement of static schedulability is common for compilers for synchronous languages, sometimes this is even built into the language. This is for example the case for Lustre [9] or the modeling language employed by the SCADE (Safety Critical Application Development Environment) tool from Esterel Technologies, which is, for example, used by Airbus for developing flight controller software [1]. For most programs, this seems acceptable, just as it is standard practice in hardware design to require acyclicity. However, there is also a large body of work on statically cyclic, yet determinate hardware circuits and synchronous programs [10], [11]. In ABBA, seen in the top-right of Fig. 1, B depends on A if input signal I is present, conversely A depends on B if I is absent. Thus there is a static dependency cycle between A and B. However, the program still has a well-defined, determinate semantics, for each possible status of I the output signal O will be present. This has been formalized by Berry as the *constructive* semantics of Esterel [6]. Most Esterel compilers are restricted to acyclic programs, such as PingPong shown on the top-left in Fig. 1; the emission of Y depends on X and Z on Y, but X does not depend on Y or Z. While few compilers can handle the full constructive semantics including statically cyclic programs, the class of cyclic yet constructive programs is interesting and well-studied. One attractive feature of that program class is that in some cases, a cyclic circuit may be smaller than an equivalent, acyclic circuit [10]. A classic example is a function that computes $y = i \ ? \ f(g(x)) : g(f(x))$, where, depending on some input $i$, $f$ must be computed before $g$ or the other way around. Another classic example is the token ring arbiter, where a rotating token dynamically determines the evaluation schedule [12]. There exist compilers that accept such programs, notably the Esterel v5 compiler, however, these are again limited to synchrony in the traditional sense

that does not take advantage of sequentiality. Existing work is again limited to classic synchrony that requires unique values per tick, which we henceforth refer to as *Berry constructiveness* (BC). In this paper, we thus aim to leverage these results on compiling constructive programs and to make them applicable to the SC MoC; i. e., we want to **provide a practical setting for compiling SC programs even if they are statically cyclic**. This includes programs such as Dynamic, seen in the bottom-right of Fig. 1. This is not Acyclic SC, because (as explained further in 2.2) no static execution schedule exists, and it is not BC, because S may be tested before it is emitted. Yet it is SC and does not require "speculation" in the sense of XY, so we consider it "well-behaved" once we accept the notion of sequentially evolving signal statuses (as exemplified already with ST) and wish to be able to compile it. We want to reject programs that require speculation, such as XY. Of course we also want to reject programs that are not SC, such as XYelse, seen in the left of Fig. 1; there, Y is emitted iff X is present, X is emitted iff Y is absent, thus there is no consistent signal evaluation.

**Contributions/Outline.** We explore the "middle ground" sketched above, which is more permissive than both BC and acyclic SC but less permissive than full SC, as illustrated in Fig. 1. Specifically:

- We propose a new, circuit-based semantics for SCEst, called *SCC* (*SC Circuits*), which is grounded in ternary constructive logic, and which is practically implementable with standard static circuit structures derived from a purely structural translation of the program (Sec. 2). SCC augments Esterel's circuit semantics (*Berry-Constructive Circuits*, *BCC*) with sequential update of variables. Again, the reference to "circuits" does not mean that SCC is applicable solely for hardware synthesis, it applies just as well to Esterel program synthesized into software, in which case the circuit netlists merely serve as "low level specifications" for the tick function to be generated.
- We provide a formal argument that SCC is conservative with respect to BCC: if some Esterel program $p$ corresponds to a constructive BCC circuit ("$p$ is BC") [2] , $p$ also corresponds to a constructive SCC circuit ("$p$ is SCC"), with the same input/output behavior (Sec. 3).

We discuss related work in Sec. 4 and conclude in Sec. 5.

For further reference, we provide a more detailed technical report that addresses topics related to the SCC semantics [13]. This includes a source-to-source transformation that transforms an SCC Esterel program into an equivalent Berry constructive (BC) Esterel program. This transformation lifts the circuit-level concepts presented here, to the Esterel source-level, which allows to reuse compilers based on BC semantics downstream. It is based on a new variant of the Static Single Assignment (SSA) form, which handles concurrency and tick boundaries and implements the concept of *SC-visibility* (Sec. 2.2) for Esterel. We further evaluate the implementation of this transformation in an experiment and discuss the effects on *schizophrenic* Esterel programs [6]. The technical report also contains a proof sketch of the conservativeness claimed in Sec. 3.

2. One might expect "$p$ is BCC" but since BC is fully equivalent to BCC, we stick to BC for consistency to previous papers.

| Statement | Description |
|---|---|
| nothing | Terminates immediately. |
| pause | Pauses execution of current thread until next tick. |
| $p$ ; $q$ | Execute $p$; when $p$ terminates, instantaneously start $q$. |
| $p$ \|\| $q$ | Run *threads* $p$ and $q$ in parallel. Terminate instantaneously when both threads have terminated. |
| loop $p$ end | Restart $p$ as soon as it terminates. Loops must be not instantaneous, each path through $p$ must contain at least one pause statement. |
| signal $S$ in $p$ end | Declares a local signal $S$ in $p$. |
| emit $S$ | Make signal $S$ present in the current tick; "write" $S$. |
| present $S$ then $p$ else $q$ end | If signal $S$ is present, immediately run $p$, otherwise run $q$. Both branches are optional. This "reads" $S$. |
| suspend $p$ when $S$ | Suspend the execution of $p$ when signal $S$ is present. |
| trap $T$ in $p$ end | Declares a trap scope with label $T$. This allows a variant of weak abort, see exit. |
| exit $T$ | Exit the trap scope labeled with $T$. Concurrent threads are weakly aborted, meaning that they can still execute until they terminate or reach a pause statement. If multiple nested traps are exited concurrently, the outermost trap scope takes precedence. |

TABLE 1: Overview of Esterel kernel statements. $p, q$ are program fragments, $S$ is a pure signal, $T$ is a trap label.

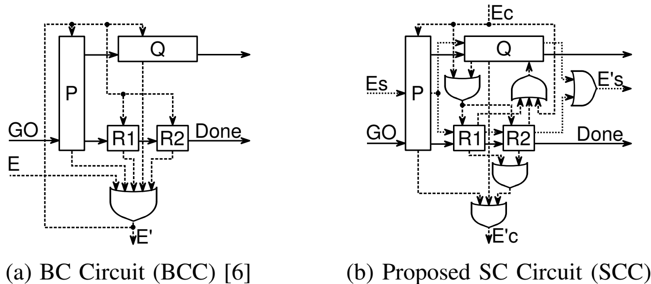
## 2. The Sequentially Constructive Circuit (SCC) Semantics

We now provide a brief summary of the Esterel language as far as required for this paper. Readers familiar with the language may advance to Sec. 2.2, which details how the SCC semantics builds on the notion of SC-visibility and a refinement of the original coherence law underlying Esterel.

### 2.1. Brief Review of Esterel

The most interesting part of Esterel, namely the way it provides determinate reactive control flow, can be reduced to the *Esterel kernel language*, see Table 1. Like most semantical treatments of Esterel, we thus concentrate the presentation of our work on that kernel language, as the extension to full Esterel is straightforward. All kernel statements are *instantaneous*, meaning that they do not consume time, except for the pause statement, which effectively separates one tick from the next. The kernel language includes only *pure signals*, which are characterized solely through the already mentioned presence status: per default, a signal is *absent*, unless it is emitted in the current tick, in which case it is *present*.

Like with most synchronous languages, Esterel programs are static in that there are no function calls, only a static module expansion mechanism, and there is no dynamic memory allocation. This is one reason why Esterel can be compiled not only into software but also into hardware. This




(a) BC Circuit (BCC) [6] (b) Proposed SC Circuit (SCC)

Figure 3: Control and signal wiring overview for P; (Q || (R1; R2)).

restriction is the basis for being able to decide interesting questions at compile time, such as whether there may be conflicting accesses to shared variables. If this is the case, we say that an Esterel program is "not causal" and the compiler rejects it. As explained before, one aim of the work presented here is to enlarge the class of programs that are considered "causal" and can be compiled into determinate code or hardware.

## 2.2. Constructive Coherence Laws (CCLs) and SC-Visibility

BCC is based on *Berry's constructive coherence law* (*BCCL*), which states that a signal is present (absent) in a tick if it must (cannot) be emitted in that tick. This law does not mention control flow and the ordering of program statements. Thus, concerning signals, there is no concept of order.

Fig. 3a presents an abstracted wiring in Berry's BCC circuit for a sequential-parallel program structure of the form P; (Q || (R1; R2)). The sequential control flow is explicitly represented through the GO activation signals directed horizontally from left to right. For signals, however, this control flow is ignored. All signal emissions, drawn vertically, are collected in a global output environment E', which is a bus of all visible signals that is fed back and combined with the global input environment E. Thus all emitters combine in a *global* OR, irrespective of the control flow relationship between the components emitting them. A present test in P therefore needs to wait for stabilisation of any downstream emitter in, e. g., R2. But since the downstream emitter depends on the GO to reach it from P, we may have a causality loop.

The key idea behind SCC is to exploit sequentiality for breaking the loop. For *(observation) points* $p_1$, $p_2$, which conceptually correspond to circuit gates/registers (see Sec. 3), we say that $p_1$ is *SC-visible* for $p_2$ iff $p_1$ is concurrent to or sequentially before $p_2$. Based on SC-visibility, we propose to refine the BCCL to the *sequentially constructive coherence law* (*SCCL*): A signal is present (absent) in a tick *at point* $p_2$ iff it must (cannot) be emitted in that tick *at some point* $p_1$ *that is SC-visible for* $p_2$. Thus SCCL differs from BCCL in that it does not consider emitters that are sequentially later.

As illustrated in Fig. 3b, we split the signal interface of each component into sequential and concurrent inputs and outputs ($E_s$, $E_c$, $E'_s$, $E'_c$). We use $E_s$, $E'_s$ to propagate signal emissions sequentially downstream and $E_c$, $E'_c$ to wire up concurrent regions locally, preserving their sequential control flow relationships. Then, as seen in Fig. 3b, the upstream process P no longer depends on any emission from downstream statements. Any local node like R1 sees signals from two "directions:" Emission upstream from it, in this case the sequential output of P, and concurrent to it, in this case Q and any concurrent environment $E_c$ of the composite program. $E_s$ is a locally restricted view on signals, whereas $E_c$ is a more global view, hence $E_s$ is a subset of $E_c$.

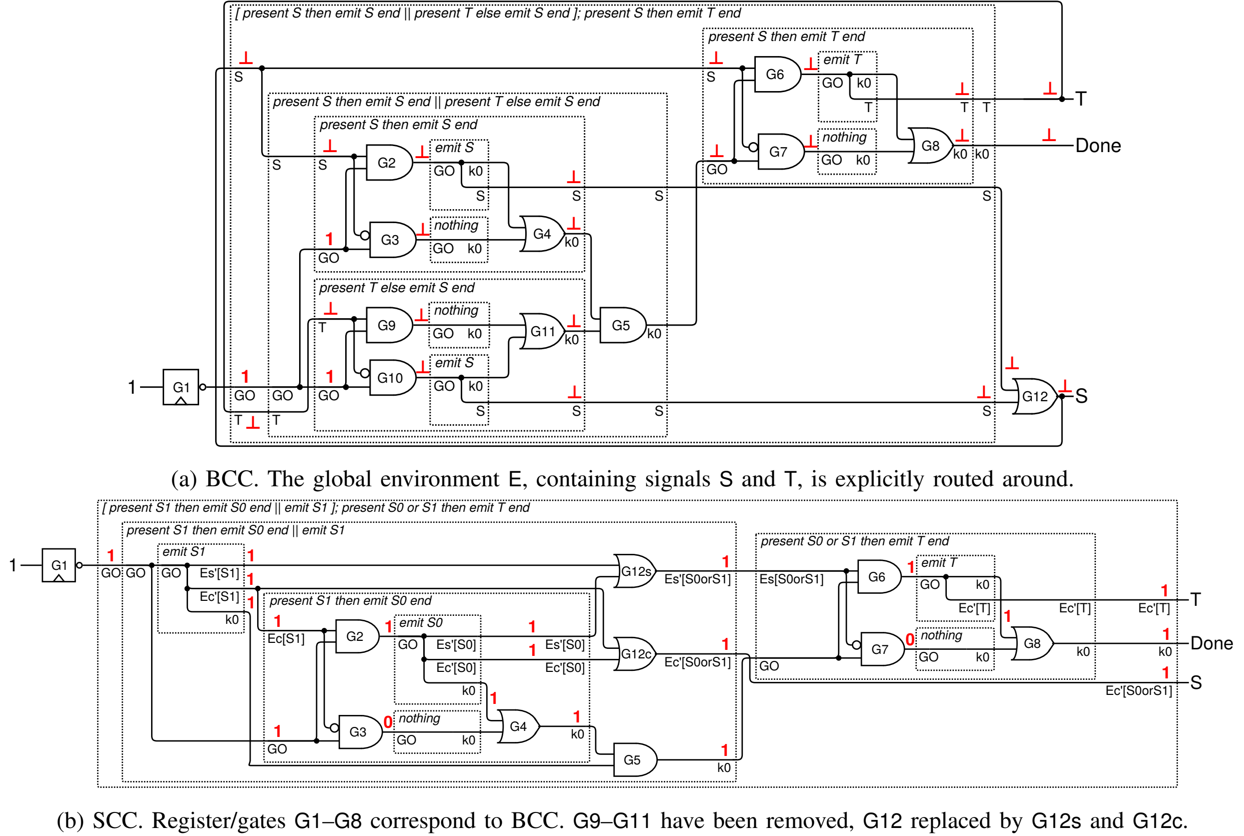
SC-visibility is not necessarily static. $E'_s$ must be blocked according to actual control flow to avoid unstable loops in case of static control flow cycles, as seen in Dynamic (bottom-right of Fig. 1). The propagation of the sequential environment must be guarded by actual control flow at run time, as discussed further in Sec. 2.4.

Note that the separation between $E'_c$ and $E'_s$ allows to receive the effect of an emit even if sequentially succeeding components do not yet have a stable $E'_s$ output, as illustrated in Fig. 3b. Thus $E'_c$ is never blocked by inactive control flow, in contrast to $E'_s$. PingPong (Fig. 1) requires this separation. After the emit of X, it must reach the other thread to allow the evaluation at the condition. However this thread cannot yet terminate since its execution depends on the emission of Y. Hence $E'_s$ cannot pass the emitted X to the second thread, but $E'_c$ can. Note that in PingPong there is a mutual dependence of the concurrent threads, which would, e. g., make modular compilation difficult and is therefore discouraged in SCADE. However, it is still acyclic at the signal level, and thus would be accepted also by a standard BC Esterel compiler.

## 2.3. The ST Example

Consider again the ST example from Fig. 1. The corresponding BCC circuit, depicted in Fig. 4a, is not constructive. Since the test of T depends on the sequentially following emission, there is a static cycle through gates G6, G9/G10, G11, G5 (as well as another cycle involving S). None of the gates involved in the cycle has a stable input outside of the cycle that would provide a defined result under constructive (non-strict) evaluation. Thus the connecting wires remain at $\perp$, and the status of T remains undefined. This in turn forbids to conclude a status for S. Thus ST is not BC and is rejected by an Esterel compiler.

However, with the proposed SCC semantics, the causality loop due to T is eliminated, because the emission of T is not SC-visible to the upstream presence test of T. As illustrated in Fig. 4b, the feedback (G6 to G9) is cut and the test of T (G9–G11) can be fully eliminated since there are no more visible emitters of T. The SCC circuit is constructive and yields the output S and T present. Thus ST is SCC, and hence SC. This corresponds to the fact that there exist SC-admissible runs for ST which all lead to the same result.




(a) BCC. The global environment E, containing signals S and T, is explicitly routed around.

(b) SCC. Register/gates G1–G8 correspond to BCC. G9–G11 have been removed, G12 replaced by G12s and G12c.

Figure 4: Alternative circuit translations for ST with constructively allocated wires. Wires that are not used in the circuit, such as SUS for suspension, are omitted. Likewise gates with constant inputs are omitted or replaced by wires.

## 2.4. The SCC Circuit Rules

Fig. 5 presents the general SCC construction rules for all Esterel kernel statements. Environments are signal buses represented by bold lines. Single signal wires can be added or extracted from these buses, illustrated by vertical bars. Gates connected to a bus denote multiple gates, one for each wire in the bus. All unconnected inputs of any component are implicitly fed by 0.

We assume that the input programs fulfill the same structural requirements as in BCC [6]. Specifically, we assume that loops are not instantaneous and that there is no schizophrenic behavior in the program [6]. Such behavior occurs when statements are executed multiple times during a tick. Even in the absence of instantaneous loops, this may happen when a loop body terminates and is instantaneously reentered. As in BCC we assume that schizophrenia can be handled by known techniques [14].

Since SCC differs from BCC only with respect to the environments, the remaining control logic concerning the input pins for activation (GO), resumption (RES), suspension (SUS), preemption (KILL) and the outputs for register selection (SEL) and Esterel's completion codes k0 (termination), k1 (pausing), k2 (innermost trap), k3, …(further traps) is exactly the same in SCC and BCC.

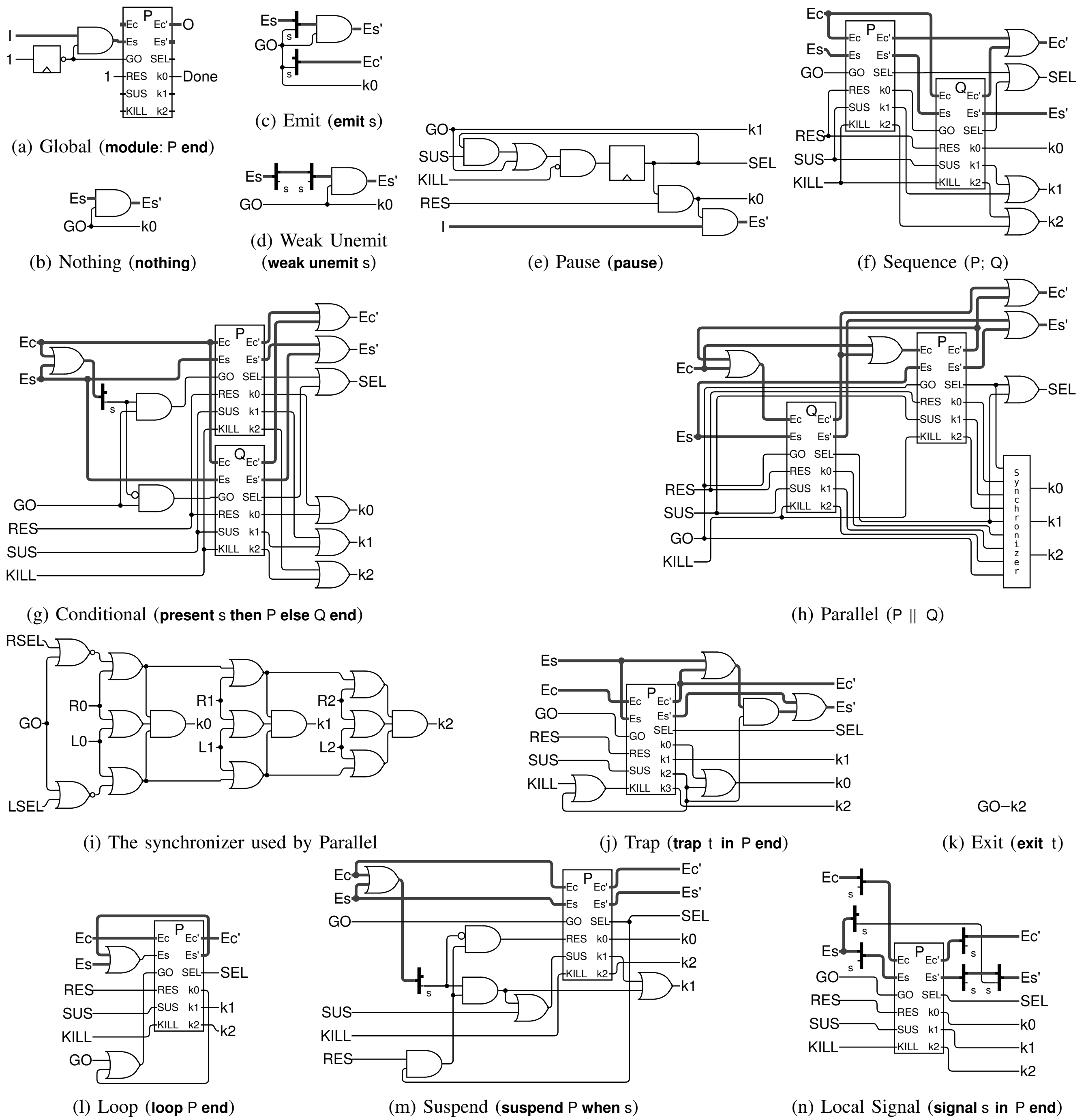
The following descriptions of the SCC rules focus on the extensions that SCC provides over BCC. For readers not familiar with the BCC Esterel circuit semantics, we briefly explain the BCC control logic as well. For a more detailed description we refer the interested reader to Berry [6].

**Global (Fig. 5a)** At the top level for a program P, inputs I feed into $E_s$ when P is initially started. The inverted output of a register, which like all registers is initially 0, activates P via GO and enables the inputs with an AND gate. $E_c$ is initialized to 0 since no signals can be emitted concurrently on this level. The outputs of P are taken from $E_c'$, which also includes signals in $E_s'$.

**Nothing (Fig. 5b)** As discussed in Sec. 2.2, nothing must actively forward (i. e., potentially block) $E_s$.

**Emit (Fig. 5c)** This drives the emitted signal on $E_s'$ and $E_c'$. As discussed in Sec. 2.2, $E_s'$ must be potentially blocked, but not $E_c'$.

**Weak unemit (Fig. 5d)** The SC MoC allows to change variable values throughout a tick. In SCEst, this has motivated the unemit statement, which is not included in Esterel [5]. An unemit reverts the effect of an emit and resets the signal to absent. However, even if the sequential signal environment is able to set a signal to absent for its sequential successors, it is more complicated to do this in a concurrent context. This would require a refined version of




(a) Global (**module**: P **end**)

(b) Nothing (**nothing**)

(c) Emit (**emit** s)

(d) Weak Unemit (**weak unemit** s)

(e) Pause (**pause**)

(f) Sequence (P; Q)

(g) Conditional (**present** s **then** P **else** Q **end**)

(h) Parallel (P || Q)

(i) The synchronizer used by Parallel

(j) Trap (**trap** t **in** P **end**)

(k) Exit (**exit** t)

(l) Loop (**loop** P **end**)

(m) Suspend (**suspend** P **when** s)

(n) Local Signal (**signal** s **in** P **end**)

Figure 5: SCC construction rules

the concurrent environment, which passes the correct signal value to concurrent readers when it is no longer modified. It also has to handle concurrent conflicts between emits and unemits.

Hence, we only introduce a *weak unemit*. This removes s from $\mathsf{E_s}$, but does not affect $\mathsf{E_c}$ to avoid conflicts with emits. The weak unemit has only a very local effect. The signal set to absent by a weak unemit is only visible to sequential successors in the same thread. The weak unemit has no effect on readers in other threads, when performing exits in traps, or on the outputs of the program, since all the related circuits use the concurrent environment.

**Pause (Fig. 5e)** If a tick starts in a pause, indicated by $\mathsf{k_0}$, the $\mathsf{E'_s}$ environment is initialized with the inputs I. For conciseness of the circuit rules, we do not hand I down through all component layers but take I directly from the global environment.

**Sequence (Fig. 5f)** Not surprisingly, this is the central rule to encode sequentiality, by forwarding $\mathsf{E'_s}$ of P to $\mathsf{E_s}$ of Q but not the other way around, as already illustrated in Fig. 3b.

**Conditional (Fig. 5g)** According to the SCCL and SC-visibility, the input signal that selects the branch is taken from an or between $\mathsf{E_c}$ and $\mathsf{E_s}$.

**Parallel (Fig. 5h)** Parallel components communicate via $\mathsf{E'_c}/\mathsf{E_c}$, see again Fig. 3b. The synchronizer, which computes the maximal completion code, is as in Berry [6].

**Trap (Fig. 5j)** $\mathsf{E'_s}$ is either $\mathsf{E'_s}$ of P, if the trap terminated normally, or $\mathsf{E'_c}$ of P, if the trap is triggered, because then the control flow jumps over the remaining statements in the trap body and P does not produce an $\mathsf{E'_s}$.

**Exit (Fig. 5k)** The exit does not produce any $\mathsf{E'_s}$, since in case of an exit the corresponding trap sequentially forwards $\mathsf{E'_c}$, not $\mathsf{E'_s}$.

**Loop (Fig. 5l)** P is restarted (via GO) when it terminates (k0). In the same manner, $\mathsf{E'_s}$ is fed back into $\mathsf{E_s}$.

**Suspend (Fig. 5m)** When suspending P based on a signal s, the state of s is determined considering both $\mathsf{E_s}$ and $\mathsf{E_c}$, just as for the conditional.

**Local Signal (Fig. 5n)** This creates a new scope for a wire s. P receives the environments $\mathsf{E_s}$ and $\mathsf{E_c}$ with s initialized to 0. s is removed from both outgoing environments. If another s exists outside the local declaration, its wire is forwarded to $\mathsf{E'_s}$.

## 3. Semantics and Conservativeness

We now formalize the notion of SCC with the goal of showing conservativeness relative to BC. Our formal semantics follows Berry [6] in representing circuits as networks of wire definitions in constructive boolean logic. For sequential constructiveness the wire definitions are stratified according to their SC-visibility capturing sequential control flow. The formal semantics relies on the same assumptions as the SCC circuit definition, i. e. a program does not contain any instantaneous loops and is free of schizophrenia, specifically that statements from concurrent threads can never appear in sequential program order. To simplify the formal treatment, we further assume that the sequential order in which two statements can appear is statically fixed. Under this assumption, which excludes programs such as Dynamic in Fig. 1, a static order, SC-visibility, can be defined on the statements corresponding to the flow dependency analysis. Without this assumption a dynamic tick dependent visibility ordering would be necessary.

A circuit $\mathcal{C} = (\mathcal{W}, \mathcal{D}, \mathcal{F}, \preceq)$ consists of *wires* $\mathcal{W}$, *wire definitions* $\mathcal{D}$, and the *SC-visibility* ordering $(\mathcal{F}, \preceq)$, which attaches visibility indices $l \in \mathcal{F}$ to the gates in the circuit. Without loss of generality assume the indices $\mathcal{F}$ are identical with the gates. The wires are partitioned into *registers* $\mathcal{R}$ and *combinational wires* $\mathcal{S}$, i.e., $\mathcal{W} = \mathcal{R} \cup \mathcal{S}$ and $\mathcal{R} \cap \mathcal{S} = \emptyset$. The combinational wires split into *inputs* $\mathcal{I} \subseteq \mathcal{S}$ and *outputs* $\mathcal{O} \subseteq \mathcal{S}$ such that $\mathcal{I} \cap \mathcal{O} = \emptyset$. A wire definition is either a *register definition* of the form $w := e$ for $w \in \mathcal{R}$ or an *implication* $w \Leftarrow_l e$ for a combinational wire $w \in \mathcal{S}$ and visibility index $l \in \mathcal{F}$. In both cases $e$ is a boolean *value expression*. There is exactly one definition $w := e$ for each register. We use the notation $\mathcal{C}(w)$ to refer to the unique expression $e$ of a register $w \in \mathcal{R}$. A combinational wire $w \in \mathcal{S}$ can have several definitions $w \Leftarrow_l e$. Observe that register definitions are used at the end of a tick to compute the next sequential state. Therefore, they do not need visibility indices because they are implicitly the last during a tick. The combinational wires are typically further partitioned as $\mathcal{W} = \mathcal{I} \cup \mathcal{L} \cup \mathcal{O}$ with (primary) *inputs* $\mathcal{I}$, *local* wires $\mathcal{L}$ and (primary) *outputs* $\mathcal{O}$.

The ordering $\preceq$ on visibility indices captures the sequential control flow in the source program. A wire definition $w_1 \Leftarrow_{l_1} e_1$ is *visible* from another $w_2 \Leftarrow_{l_2} e_2$ iff $l_1$ is not sequentially downstream from $l_2$, i.e., if $l_2 \not\preceq l_1$. For instance, consider the BCC circuit in Fig. 4a implementing ST from Fig. 1. The gates G2, G6, G10, G12 arise from wire definitions

$$S[G2] \quad \Leftarrow_{G2} \quad GO[G1] \wedge S[G12] \tag{1}$$
$$T[G6] \quad \Leftarrow_{G6} \quad S[G12] \wedge \mathsf{k_0}[G5] \tag{2}$$
$$S[G10] \quad \Leftarrow_{G10} \quad GO[G1] \wedge \neg T[G6] \tag{3}$$
$$S[G12] \quad \Leftarrow_{G12} \quad S[G10] \vee S[G2]. \tag{4}$$

where the notation $X[G]$ identifies the gate $G$ from which the wire is driven and the name $X$ of the control signal in the circuit translation (Fig. 5) represented by the wire. The visibility ordering $\preceq$ is obtained from the control flow of the source program in Fig. 1. Since G10 comes from **present** T **else emit** S and G6 comes from **present** S **then emit** T, G6 is sequentially downstream from G10, so that $G10 \preceq G6$. In contrast, we have $X \not\preceq Y$ for all $X, Y \in \{G2, G10, G12\}$. G2 and G10 are incomparable because they are instantiated from the concurrent present tests in ST. G12 is the global disjunction collecting and feeding back all emissions on S from these two parallel threads. Therefore, $G12$ is not sequentially ordered relative to either G2 or G10, but $G6$ is downstream from $G12$, i.e., $G12 \preceq G6$.

The semantics of a circuit is based on constructive value propagation

$$\mathcal{C}, I, R \vdash e \hookrightarrow b \tag{5}$$

$$\frac{\exists w \Leftarrow_l e \in \mathcal{C}.\ \pi \not\preceq l \wedge e \hookrightarrow_{\pi \oplus l} 1}{w \hookrightarrow_\pi 1}\ PRES(\pi, l) \qquad \frac{\forall w \Leftarrow_l e \in \mathcal{C}.\ \pi \not\preceq l \Rightarrow e \hookrightarrow_{\pi \oplus l} 0}{w \hookrightarrow_\pi 0}\ ABS(\pi, w)$$

$$\frac{w \in \mathcal{I}}{w \hookrightarrow_\pi I(w)}\ IN \quad \frac{w \in \mathcal{R}}{w \hookrightarrow_\pi R(w)}\ REG \quad \frac{e_1 \hookrightarrow_\pi 0 \quad e_2 \hookrightarrow_\pi 0}{e_1 \vee e_2 \hookrightarrow_\pi 0}\ OP_{\neg\vee} \quad \frac{e_1 \hookrightarrow_\pi 1}{e_1 \vee e_2 \hookrightarrow_\pi 1}\ OP_{l\vee} \quad \frac{e_2 \hookrightarrow_\pi 1}{e_1 \vee e_2 \hookrightarrow_\pi 1}\ OP_{r\vee}$$

$$\frac{e \hookrightarrow_\pi b}{\neg e \hookrightarrow_\pi \neg b}\ OP_\neg \quad \frac{b \in \mathbb{B}}{b \hookrightarrow_\pi b}\ OP_c \quad \frac{e_1 \hookrightarrow_\pi 1 \quad e_2 \hookrightarrow_\pi 1}{e_1 \wedge e_2 \hookrightarrow_\pi 1}\ OP_\wedge \quad \frac{e_1 \hookrightarrow_\pi 0}{e_1 \wedge e_2 \hookrightarrow_\pi 0}\ OP_{l\wedge} \quad \frac{e_2 \hookrightarrow_\pi 0}{e_1 \wedge e_2 \hookrightarrow_\pi 0}\ OP_{r\wedge}$$

Figure 6: Visibility-Restricted Constructive Evaluation Rules. The evaluation context $\mathcal{C}, I, R$ is implicit.

which evaluates a boolean expression $e$ over $\mathcal{W}$ using the evaluation rules of Kleene ternary algebra [15], in the context of a circuit $\mathcal{C}$ and under input *event* $I$ and register *state* $R$. Input events are assignments of boolean values to all input wires. A register state is an assignment of boolean values to all register wires. The *constructive macro step reaction* then is a relation

$$\mathcal{C} \vdash I, R \hookrightarrow O, R' \tag{6}$$

expressing that in register state $R$ for the input event $I$ the circuit constructively evaluates to output event $O$ and new register state $R'$. The macro step reaction then states that (i) for all $w \in \mathcal{O}$, we have $\mathcal{C}, I, R \vdash w \hookrightarrow O(w)$ and (ii) for all $w \in \mathcal{R}$, $\mathcal{C}, I, R \vdash \mathcal{C}(w) \hookrightarrow R(w)$. Note that we evaluate the expression $\mathcal{C}(w)$ rather than $w$, because we are interested in the next state value of the register, not its current value.

To exploit visibility we introduce a labelled version

$$\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b \tag{7}$$

of the standard constructive semantics which obtains the constructive value $b$ of an expression $e$ *visible* relative to a set $\pi \subset \mathcal{F}$ of visibility indices. These represent a set of *observation points* from concurrent threads that are active in an evaluation. Each one is sequentially first in its thread. Hence, the indices in $\pi$ are sequentially incomparable visibility indices ($\pi$ is an *antichain*), so that for all $l_1, l_2 \in \pi$ if $l_1 \preceq l_2$ then $l_1 = l_2$.

The evaluation rules are shown in Fig. 6. For notational compactness we write $e \hookrightarrow_\pi b$ instead of (7). The standard ternary evaluation of boolean expressions is implemented by the $OP$ rules, which do not depend on the observation points $\pi$. Rules $IN$ and $REG$ are the evaluation of inputs and register wires. The visibility information $\pi$ becomes relevant in the evaluation of standard wires described by the rules $PRES$ and $ABS$. The former stabilises a wire $w \in \mathcal{S}$ to 1 if there is some visible wire definition $w \Leftarrow_l e$ in the circuit whose expression $e$ evaluates to 1. We say a wire definition with index $l$ is *$\pi$-visible*, written $\pi \not\preceq l$, if $l$ does not lie downstream from any observation point in $\pi$, i.e., there is no $m \in \pi$ with $m \preceq l$. If this condition is met, $PRES$ evaluates the expression $e$ under the observation points $\pi \oplus l$, which adds $l$ to the anti-chain if it is concurrent to $\pi$ or shifts to $l$ otherwise. Formally, $\pi \oplus l = \pi \setminus \{m \in \pi \mid l \prec m\} \cup \{l\}$. Note that if we would drop $\pi$ and simply use $l$ to evaluate $e$, we might eventually jump back to a wire (gate) that is downstream from some observation point in $\pi$. This is what we avoid if we preserve $\pi$ in the premise of the $PRES$ rule. The $ABS$ rule is dual to $PRES$. It stabilises a standard wire $w$ to 0 if the expressions $e$ in all wire definitions for $w$ that are $\pi$-visible evaluate to 0. We add the relevant parameters $\pi$, $l$, $w$ to the rule names for ease of reference.

The visibility information enters the evaluation rules $PRES(\pi, l)$ and $ABS(\pi, w)$ in line with the sequentially constructive coherence law (Sec. 2.2). Let $PRES(l)$ and $ABS(w)$ refer to the same rules but without the side-conditions "$\pi \not\preceq l$." Let us write $\mathcal{C}, I, R \vdash e \hookrightarrow b$ for an evaluation in the system of Fig. 6 with the unconstrained rules $PRES(l)$ and $ABS(w)$ instead of $PRES(\pi, l)$ and $ABS(\pi, w)$. This is precisely the standard constructive value propagation of Berry [6].

Equivalently, we obtain Berry's evaluation semantics if we assume each wire definition is labelled with a different visibility index and the flow ordering $\preceq$ makes any two wire definitions incomparable, e.g., if $\preceq$ is the identity relation on $\mathcal{F}$. This is the same as saying every wire is concurrent to every other. Then, the side conditions in $PRES(\pi, l)$ and $ABS(w)$ become redundant. In other words, constructiveness of Berry circuits has "maximal visibility." For a non-trivial flow ordering, Berry circuits will evaluate in a different way, depending on whether the $PRES(l)/ABS(w)$ or the $PRES(\pi, l)/ABS(\pi, w)$ rules are used. However, the effect is conservative in the sense that the visibility constraints only make more wires stabilise but never change their value. This is a consequence of the following property of Berry circuits: If a wire evaluation with $PRES(k)$ depends on the evaluation of another with $PRES(m)$, then $m$ cannot be sequentially downstream from $k$, i.e., $k \not\prec m$. The reason is that all emissions must be activated by GO wires and these are chained up in program order. Hence, the GO activation wires hold up downstream emitters until all control flow has been resolved upstream.

Adding visibility is non-trivial, because the side-conditions act both co- and contra-variantly. E.g., changing $\pi_1$ to $\pi_2$ with $\pi_1 \preceq \pi_2$ preserves every application of $PRES(\pi, l)$ but may invalidate some application of $ABS(\pi, l)$. Since an evaluation $\vdash e_1 \hookrightarrow_{\pi_1} 1$ may depend on another $\vdash e_2 \hookrightarrow_{\pi_2} 0$, it is not immediately obvious how the semantics generated by the two systems are related. In particular, evaluating a circuit under visibility constraints does not warrant the conclusion, in general, that we get more signals being decided absent than without visibility.

A proof sketch for the following propositions and the theorem is provided in the technical report [13].

**Proposition 1.** *Let $\mathcal{C} = BCC(P)$ be the Berry circuit of $P$. Then, $\mathcal{C}, I, R \vdash e \hookrightarrow b$ implies $\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b$ for all antichains $\pi \subset \mathcal{F}$ from which every wire implication $w \Leftarrow_l d$ in $\mathcal{C}$ is visible, i.e., such that $\pi \not\preceq l$.*

The SCC circuit translation produces circuits which are "flow-oriented," i.e., wires are never passed against sequential control flow. Thus, adding visibility restrictions in the evaluation does not have any effect.

**Proposition 2.** *Let $\mathcal{C} = SCC(P)$ be the circuit of $P$ obtained under the new SCC circuit translation (Sec. 2.4). Let $\pi \subset \mathcal{F}$ such that $\pi \not\preceq l$ for all wire definitions $w \Leftarrow_l d$ in $\mathcal{C}$. Then, $\mathcal{C}, I, R \vdash e \hookrightarrow b$ iff $\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b$.*

The final Theorem 1 states that if $BCC(P)$ stabilises an output signal then $SCC(P)$ must also stabilise this signal with the *same* value.

**Theorem 1.** *Let $SCC(P)$ and $BCC(P)$ be the circuits of a program $P$ under the new sequentially constructive and standard Berry translation, respectively. Assume*

$$BCC(P), \mathsf{E} \Leftarrow_0 I \vee \mathsf{E_c} \vee \mathsf{E'}, \mathsf{E'_c} \Leftarrow_0 \mathsf{E'}, I, R \vdash \mathsf{E'_c}.s \hookrightarrow_\pi b$$

*for some signal $s$ and observation points $\pi \subset \mathcal{F}$. Then,*

$$SCC(P), \mathsf{E_s} \Leftarrow_0 I, I, R \vdash \mathsf{E'_c}.s \hookrightarrow_\pi b.$$

Theorem 1, in combination with Proposition 2, implies conservativeness of $SCC$ over $BCC$.

## 4. Related Work

The semantics introduced here for Esterel deviates from the constructive semantics of Berry [6] in that sequential compositions are executed like ordinary imperative programs and signal emissions behave like assignments to boolean variables. As noted earlier, this is inspired by the SC proposal [2], which however, permits speculation and does not lend itself to efficient compilation.

Another way to understand our work is as an approach to relax the traditional, rather rigid, synchronous model of concurrent programming by a more generous use of shared communication structure. The communication structure here are the signals and the relaxation consists in permitting sequential threads to change signal values more than once during a synchronous tick. This permits signals to be used like variables and reduces the gap between synchronous control flow and standard imperative programming. For data flow synchronous programming an analogous approach has been proposed by Cohen et al. [16] on $N$-synchronous Kahn networks.

The compilation of Esterel and its potentially quite intricate reactive control flow structures has sparked the interest of a number of researchers, as discussed by Potop-Butucaru et al. [3]. The circuit-based compilation, where the synthesized code simulates a netlist, produces compact code, that scales basically linearly with the original Esterel program [3]. However, since the code simulates the whole program irrespective of whether it is "active" in the current tick, the code tends to become rather slow for larger programs. A good compromise between speed and size is achieved by a more software-like approach, where concurrent threads are statically scheduled and interleaved at compile time, which is for example implemented in the Columbia Esterel Compiler [17]. A good overview of this and other approaches for compiling concurrent programs (not necessarily Esterel) is presented by Edwards [18]. Any of these compilation approaches may potentially be used for further downstream compilation of SCC programs, at least as far as sequential constructiveness is concerned. Not all Esterel compilers can handle all programs, in particular if there are static cycles in the program as in ABBA from Fig. 1. This, however, is an orthogonal issue to the work presented here.

As pointed out in the introduction, most EDA tools require acyclic circuits, as do most synchronous language compilers. This has motivated numerous works on transforming cyclic circuits into equivalent acyclic ones; Neiroukh et al. provide a good overview and present a technique of their own [11]. Lukoschus et al. present an approach to remove cycles at the Esterel level [19]. Schneider et al. have suggested the use of scheduling or atomicity constraints for increasing constructiveness of cyclic circuits [15], [20]. The idea of flow indices to express evaluation order in ternary analysis, as explored here, seems to be new.

In this work we stress the role of sequential program order ("visibility") in order to permit several write accesses to Esterel signals within a tick. The sequential order resolves the potential non-determinacy because every read access only sees the sequentially last write. There are other ways to resolve multiple writes, preserving determinacy, namely if these writes are accessing disjoint parts of signal value. Following this idea, a powerful technique to generate coherent shared memory structures for functional programs has recently been proposed by Kuper et. al. [21].

## 5. Conclusion and Future Work

This work defines the program class SCC, which encompasses the programs for which a circuit generated according to the SCC circuit semantics is constructive. SCC on the one hand defines a significant subset of SC programs, namely those that can be executed without "speculation," and on the other hand extends compilation technology for synchronous programs, as illustrated in Fig. 1. SCC programs can be structurally translated to circuits, according to the SCC rules set down in Sec. 2, and then be compiled further into hardware or software using standard techniques. Alternatively, SCC programs can be translated into BC programs on the source-level [13].

There are numerous directions to proceed from here. To begin with, while this work is mostly about expanding the range of compilable programs, a natural question is how the

size of SCC circuits compares to BCC circuits. Our intuition is that there should be no significant increase, and often the circuits should be even smaller due to the increased partial evaluation done at compile time. One example is ST (Fig. 4), where SCC has two gates less than BCC.

Conservativeness (Sec. 3) is a combination of the result that if a Berry circuit of a program $P$ stabilises a signal, then it stabilises it under visibility (Prop. 1); that this further implies that the corresponding SCC of $P$ also stabilises it under visibility (Thm. 1); and finally that if SCC stabilises a signal with visibility restriction, then it stabilises without them (Prop. 2). This chain gives more information than just conservativeness. It goes some way to explain SCC as a flow-sensitive evaluation of Berry circuits. For an exact characterisation it would be interesting to prove the converse of Thm. 1 in future work. Also, we plan to extend the formalisation for dynamic visibility relations to lift the restrictions on programs mentioned at the beginning of Sec. 3.

We have developed our results in the setting of pure Esterel. The extension to remaining Esterel features, such as valued signals, variables etc., should be mostly straightforward, but still remains to be done. An interesting statement is the (strong) unemit provided by SCEst [5], which may lead to conflicts if performed concurrently with an emit. We can augment SCC with conflicts by emitting an error signal whenever such a conflict occurs, and feeding that error into a circuit that is constructive iff the error cannot occur, e. g., a concurrently running **signal** helper **in present** error **and** helper **then emit** helper **end end**. However, there is still the difficulty that a thread may perform both an emit and an unemit with dynamic ordering between them, and a concurrent thread has to decide which of these is (un)emitted last.

Finally, we would also like to apply our results to other languages building on the SC MoC, such as SCCharts [8]. It would be interesting to explore how much could be gained by adopting the SC MoC and SCC in other synchronous languages as well, such as Lustre or SCADE.

## References


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

[2] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop, "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation," *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, vol. 13, no. 4s, pp. 144:1–144:26, Jul. 2014.

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

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

[5] K. Rathlev, S. Smyth, C. Motika, R. von Hanxleden, and M. Mendler, "SCEst: Sequentially Constructive Esterel," in *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '15)*, Austin, TX, USA, Sep. 2015.

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

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

[8] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, Jun. 2014.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: a declarative language for programming synchronous systems," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*. Munich, Germany: ACM, 1987, pp. 178–188.

[10] M. D. Riedel and J. Bruck, "The Synthesis of Cyclic Combinational Circuits," in *Proceedings of the conference on Design automation (DAC '03)*, Anaheim, California, USA, Jun. 2003.

[11] O. Neiroukh, S. A. Edwards, and X. Song, "Transforming cyclic circuits into acyclic equivalents," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1775–1787, 2008.

[12] P. Pandya, "The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs," in *Electronic Notes in Theoretical Computer Science*, F. Maraninchi, A. Girault, and É. Rutten, Eds., vol. 65. Elsevier, 2002.

[13] A. Schulz-Rosengarten, S. Smyth, R. von Hanxleden, and M. Mendler, "A sequentially constructive circuit semantics for esterel," Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1801, Feb. 2018, ISSN 2192-6247.

[14] O. Tardieu and R. de Simone, "Curing schizophrenia by program rewriting in Esterel," in *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)*, San Diego, CA, USA, 2004.

[15] K. Schneider, J. Brandt, T. Schüle, and T. Türk, "Maximal causality analysis," in *Conference on Application of Concurrency to System Design (ACSD'05)*, St. Malo, France, Jun. 2005, pp. 106–115.

[16] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-synchronous Kahn networks: A relaxed model of synchrony for real-time systems," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 180–193.

[17] S. A. Edwards and J. Zeng, "Code generation in the Columbia Esterel Compiler," *EURASIP Journal on Embedded Systems*, vol. Article ID 52651, 31 pages, 2007.

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

[19] J. Lukoschus and R. von Hanxleden, "Removing cycles in Esterel programs," in *International Workshop on Synchronous Languages, Applications and Programming (SLAP '05)*, Edinburgh, Apr. 2005.

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

[21] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton, "Freeze after writing: Quasi-deterministic parallel programming with LVars," in *Principles of Programming Languages (POPL '14)*. New York, USA: ACM, 2014, pp. 257–270.