INSTITUT FÜR INFORMATIK

A Sequentially Constructive Circuit Semantics for Esterel

Alexander Schulz-Rosengarten, Steven Smyth,
Reinhard von Hanxleden, Michael Mendler

Bericht Nr. 1801
February 2018
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

A Sequentially Constructive Circuit Semantics for Esterel

Alexander Schulz-Rosengarten, Steven Smyth,
Reinhard von Hanxleden, Michael Mendler

Report No. 1801
February 2018
ISSN 2192-6247

E-mail: {als,ssm,rvh}@informatik.uni-kiel.de
michael.mendler@uni-bamberg.de

Abstract

Static Single Assignment (SSA) is an established concept that facilitates various program optimizations. However, it is typically restricted to sequential programming. We present an approach that extends SSA for concurrent, reactive programming, specifically for the synchronous language Esterel. This extended SSA transformation expands the class of programs that can be compiled by existing Esterel compilers without causality problems. It also offers a new, efficient solution for the well-studied signal reincarnation problem. Finally, our approach rules out speculation/backtracking, unlike the recently proposed sequentially constructive model of computation.

Keywords: Static single assignment, concurrency, reactive systems, determinacy, synchronous programming, sequential constructiveness, Esterel

Contents

1 Introduction

A classic challenge in programming reactive systems is to reconcile concurrency with determinate behavior. Synchronous programming languages [3], such as Esterel [30], achieve this 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. The idea of this report is to provide this imperative programming convenience, without leaving Esterel's solid grounding in constructive logic.

To illustrate, consider the WriteAfterRead Java code fragment in Lst. 1.1. If the flag done is false, some code (replaced by ellipsis) is performed and subsequently done is set to true. This programming pattern is quite common for example in programmable logic controller code for embedded devices. We call this a *sequential update* of done, since there is a read of done followed sequentially by a write of done. This might lead to a situation where done is both false and true within a reaction, but in imperative programming this is still a perfectly legal programming pattern. There is no possible non-determinacy, as there is not even any concurrency that might lead to a race condition. In contrast, the "morally equivalent" Esterel code fragment in Lst. 1.2 is not accepted by an Esterel compiler, since done might be absent and present within the same tick. This is forbidden because of the aforementioned requirement of unique values throughout a tick. From the hardware/circuit point of view, where done would be represented by a single wire

```
1 boolean done;
2 ...
3 if  (!done) {
4   ...
5   done = true;
6 }
```

Listing 1.1: WriteAfterRead code fragment in Java, a *sequential update*

```
1 signal done;
2 ...
3 present (done) else
4   ...
5   emit done;
6 end
```

Listing 1.2: WriteAfterRead in Esterel, not accepted by Esterel compilers

```
1 signal done0, done1;
2 ...
3 present (done0) else
4   ...
5   emit done1;
6 end
```

Listing 1.3: WriteAfterRead in Esterel, after SSA transformation, accepted by Esterel compilers

```
1  module SignalReinc:
2  output O;
3  loop
4    signal S in
5      present S then
6        emit O end;
7      pause;
8      emit S
9    end
10 end
```

Listing 1.4: SignalReinc, with the S scope left and reentered (from [37]).

```
1  module SignalReincTdS:
2  output O;
3  loop
4    signal S in
5      present S then
6        emit O end;
7      gotopause P1;
8      emit S
9    end
10   signal S in
11     present S then
12       emit O end;
13     P1: pause;
14     emit S
15   end
16 end
```

Listing 1.5: SignalReinc with cured schizophrenia using the approach of Tardieu and de Simone [37].

```
1  module SignalReincSSA:
2  output O;
3  loop
4    signal S0, S1 in
5      present S0 then
6        emit O end;
7      pause;
8      emit S1
9    end
10 end
```

Listing 1.6: SignalReinc with cured schizophrenia, in SSA form, after applying SCC2BC.

with "low" encoding signal absence and "high" encoding signal presence, having done both absent and present within a tick would indeed be problematic. However, this can be resolved by splitting done into multiple *versions* done0 and done1. Lst. 1.3 shows a variant of Lst. 1.2, where done0 represents the value of done at the point of testing the conditional in line 3, and done1 is a new version of done that holds the value of done after the emission in line 5 for downstream readers. This makes the program acceptable for Esterel, also from the hardware view, as the possible value clash is resolved by having separate wires for the different values.

Splitting variables into different versions is just what the Static Single Assignment (SSA) form provides [13]. In sequential, non-reactive programming, SSA is a well-established compilation concept to facilitate various program optimizations, such as code motion, partial redundancy elimination, or constant propagation.

As another motivation for using SSA in Esterel, consider the Esterel program SignalReinc in Lst. 1.4. The pause instruction in line 7 separates reactions (logical ticks). From the second tick onwards, the signal S will be emitted in line 8, thus it will be present. However, when instantaneously looping around to the presence test of S in line 5, S will be considered absent, because a fresh signal scope for S has been just entered in line 4. In contrast to the WriteAfterRead example from Lst. 1.2, SignalReinc is a legal Esterel program, and Esterel compilers have to accept it. However, from the hardware view, this is again problematic, as we cannot dynamically create new wires in hardware the way we can re-use memory locations in software. Furthermore, a common, efficient approach to compile Esterel into software uses a data-flow style approach that again relies on the assumption of having a unique signal status for each reaction [30]. As it turns

out, this *signal reincarnation* issue illustrated in SignalReinc is an instance of the well-studied *schizophrenia* problem for Esterel compilation [35, 5, 37, 39]. The most efficient technique developed so far, proposed by Tardieu and de Simone [37], is to duplicate loop bodies into a *surface* and *depth* copies, and to replace pause instructions in the surface copy into gotopause statements that transfer control to the depth copy. Transforming SignalReinc this way results in SignalReincTdS shown in Lst. 1.5. This resolves statement reincarnation by separating the multiple signal instances into statically disjoint signal scopes. Again there are multiple copies of S, one corresponding to the scope opened in line 4, the other corresponding to a scope opened in line 10. In this particular example, the transformation result SignalReincTdS could be optimized by eliminating unreachable code. Still, this approach has potentially quadratic code size increase, and it requires a new gotopause instruction that is not part of standard Esterel. However, it turns out that in the signal reincarnation problem, there is no need to duplicate whole program parts, it is enough to duplicate just the signal instances. This is again exactly what SSA does. Applying SSA to SignalReinc results in the SignalReincSSA version shown in Lst. 1.6, which is more compact than SignalReincTdS and makes do without a gotopause instruction.

In light of these examples, it may seem surprising that SSA is not provided by existing Esterel compilers. So far, it is still left to the programmer to manually write SSA-style Esterel programs to emulate, for example, the sequential update of WriteAfterRead. Or, perhaps even worse, programmers resort to inserting additional pause instructions to split variable versions into different ticks, which quickly leads to delicate timing issues. However, applying SSA to Esterel is less trivial than the purely sequential examples discussed so far suggest. One challenge is the proper handling of concurrency, another difficulty is the division of the computation into different ticks and the implicit signal initialization.

The SSA transformation should also have a formal grounding. Esterel offers a choice of different, equivalent semantics [5]; we here choose the so-called *circuit semantics* as a reference, as it is conceptually relatively straightforward. The "circuit" part in the name can be somewhat misleading in that this semantics is not necessarily about hardware synthesis, but more about the usage of *constructive logic*. Very briefly, ternary constructive logic differs from standard Boolean logic in that variables/wires may not only be 0 (low) or 1 (high) but also $\bot$, and there is no "law of excluded middle." Thus, under constructive logic the equation $S = S \vee \neg S$ yields $S = \bot$, not $S = 1$. This nicely corresponds to the fact that a circuit for $S = S \vee \neg S$ is (perhaps surprisingly) not guaranteed to stabilize at $S = 1$ but, for some gate and wire delays, may oscillate forever. Constructive logic can be used to reason about hardware circuitry, but it is primarily a mathematical formalism that is agnostic to a particular implementation target. Nevertheless, conceptually reducing an Esterel program to a well-behaved netlist has the advantage that such a netlist can be rather trivially be mapped to data-flow style software, which implements the Esterel semantics by simulating the corresponding netlist. Another nice aspect of the circuit semantics is that it is grounded in physics: an Esterel program is considered valid if and only if it directly corresponds to a well-behaved circuit. "Well behaved" here means that for all ticks, for all possible input sequences,

all wires stabilize to uniquely defined values after finite time. (This requirement may be relaxed to requiring that just the output wires have a unique stabilization, but we here employ the stricter requirement that *all* wires must stabilize.) In this report, we will refer to the circuit semantics proposed by Berry [5] as *Berry Circuit Semantics* (BCC). Similarly, we will say that an Esterel program is *Berry Constructive* (BC) if it is constructive in the sense of BCC and thus should be accepted by an Esterel compiler.

Contributions/Outline. We augment classic, sequential SSA with concurrency and logical ticks, and practically explore this in the context of Esterel:

- We propose a new, broader semantic foundation for Esterel, called *SCC* (*SC Circuits*), which is still grounded in constructive logic, and which is practically implementable with a purely structural translation of the program (Sec. 2). SCC conservatively extends the Berry circuit semantics (BCC) with sequential updates 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 present a source-to-source transformation, *SCC2BC*, that transforms an SCC Esterel program p into an equivalent Berry constructive (BC) Esterel program p_B, by a new variant of SSA, *SCSSA*, that handles concurrency and tick boundaries (Sec. 3). Then p_B can be compiled with existing Esterel compilation technology, such as the causality analysis of the Esterel v5 compiler [36]. As illustrated with SignalReinc, the SCC2BC also constitutes a novel source-level transformation approach towards handling *signal reincarnation* that compares favorably with previous work [5, 37] (Sec. 3.4). We have implemented the SCC2BC transformation in the KIELER framework[1] (Sec. 3.5).

- We provide a formal argument that SCC is conservative with respect to BCC just as SCEst is conservative with respect to Esterel: if some Esterel program p corresponds to a constructive BCC circuit ("p is BC"), p also corresponds to a constructive SCC circuit ("p is SCC"), with the same input/output behavior (Sec. 4).

- We compare SCC with the recently proposed sequentially constructive (SC) model of computation [18] and argue that Esterel programs that are SCC are also sequentially constructive, but not the other way around (Sec. 5). More specifically, SCC programs are SC programs that are not "speculative." SCC presents a way to practically implement an interesting class of SC programs, including programs with cyclic signal dependencies that could not be compiled with the SC compilation approaches proposed so far [17].

We discuss related work in Sec. 6 and conclude in Sec. 7.

[1] `http://rtsys.informatik.uni-kiel.de/kieler`

2 The Sequentially Constructive Circuit (SCC) Semantics

We now provide a brief summary of the Esterel language as far as is required for this report. 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

Esterel has been originally developed to program embedded systems such as robots. Since then it has evolved into a language for arbitrary reactive system software and for hardware design. As it allows the target-independent, abstract specification of system behavior, it also has been employed for hardware-software codesign [2]. Esterel is an imperative, control-oriented synchronous language, which provides determinate concurrency and various forms of preemption. It has evolved through several versions, the most widely propagated being "v5." The versions up to v5 typically are used in combination with a *host language* such as C, for example to define non-primitive types or low-level interactions with the environment. The more recent Esterel "v7" provides a richer type system and other extensions such as multi-clocking.

The most interesting part of Esterel, namely the way it provides determinate reactive control flow, can be reduced to the *Esterel kernel language*. 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 (our implementation of the SCC2BC transformation presented later is not restricted to the kernel language). 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*. Full Esterel v5 also includes *variables*, which—unlike signals—cannot be accessed concurrently, and *valued signals*, which not only carry a presence status but also a value of some (primitive) type.

The kernel language contains a small set of *kernel statements*, which are summarized in Table 2.1. 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.

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 restriction is the basis for being able to decide interesting

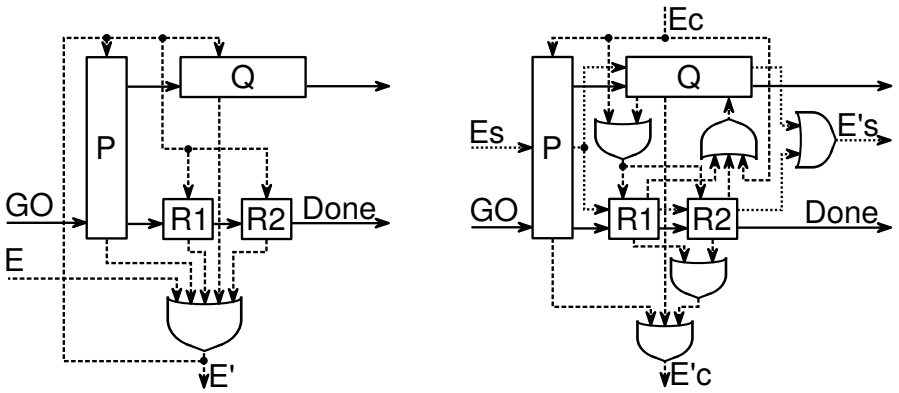
nothing	Terminates immediately.
pause	Pauses execution of the current thread until the next tick.
p ; q	Execute p; when p terminates, instantaneously start q
$p \parallel q$	Run "threads" p and q in parallel. The parallel terminates instantaneously when both threads have terminated.
loop p end	Restart p as soon as it terminates. Loops are not allowed to be instantaneous, that is, each path through p must contain at least one pause statement.
signal S in p end	Declares a local signal S.
emit S	Make signal S present in the current tick.
present S then p else q end	If signal S is present in the current tick, immediately run p, otherwise run q. Both branches are optional.
suspend p when S	Suspends the execution of p when signal S is present. However, this is not immediate, but only applies from the next tick after suspend has been entered.
trap T in p end	Declares a trap scope with label T.
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 in the same tick, the outermost trap scope takes precedence.

Table 2.1: Overview of Esterel kernel statements. p, q are program fragments, S is a pure signal, T is a trap label.

questions at compile time, such as whether there may be conflicting accesses to shared variables. If this is the case, 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**. Consequently, if a signal is both emitted and tested in a tick, the emit (at least the first one) has to be scheduled before the presence test, because otherwise the presence test would consider a signal absent even though it will become present in the current tick. We call this scheduling requirement the *emit-before-test* rule. Berry's constructive coherence law does not mention control flow and the ordering of program statements. Thus, concerning signals, there is no concept of order. The key idea behind SCC is to introduce sequentiality here. We exploit the sequential control flow in the source program to disambiguate multiple writes to a signal. In a program like **present** X **else emit** F **end**; **emit** X; **present** X **then emit** T **end** the write (**emit** X) is strictly after the first read (**present** X) and strictly before the second read. As all signals are initialised to be absent, the program emits both F and T, the former because X is initially absent and the latter because X is



(a) BC Circuit (BCC) (b) SC Circuit (SCC)

Figure 2.1: Control and signal wiring overview for $P;(Q\|(R_1;R_2))$.

later emitted. There is no read/write race condition because the signal tests and the emit are strictly ordered by the control flow. In Berry's semantics of Esterel, the sequential operator ";" does not work in this imperative way. In Esterel, the semicolon corresponds to a parallel composition, in our example corresponding to **present** X **else emit** F **end** || **emit** X || **present** X **then emit** T **end**, with the extra restriction that the behaviour remains the same if the statements are evaluated from left to right. Evidently, this is not the case for this example. In the parallel version, the first present test **present** X **else emit** F **end** will see the emission of X and thus skip the emission of F. In the BCC translation this is detected by a causal cycle between *statement activation*, which follows the (left-to-right) order of ";", and the data dependence from the emission **emit** X (right-to-left) across the parallel || to the first read **present** X. Thus, in BCC, all signal emissions are visible to all signal readers, and this code fragment will not be considered BC.

Fig. 2.1a presents an abstracted wiring in Berry's BCC circuit for a sequential-parallel program structure of the form $P;(Q\|(R_1;R_2))$. 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., R_2. 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. 4), 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 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 a point* p_1 *that is SC-visible for* p_2.** Thus the difference between SCCL and BCCL is that SCCL does not consider emitters

```
module PingPong:
output Ping, Pong, Done;
[
  emit Ping;
  present Pong then
    emit Done end
||
  present Ping then
    emit Pong end
]
```

Listing 2.1: PingPong motivates separate sequential and concurrent environments.

```
module ST:
output S, T;
[
  present S then
    emit S end
||
  present T else
    emit S end
];
present S then emit T end
```

Listing 2.2: ST, which is SCC but not BC, illustrates sequential and parallel signal visibility.

```
module ST_B:
output S, T;
signal S0, S1 in
[
  present S1 then emit S0 end
||
  emit S1
];
present S0 or S1 then
  emit S end;
present S then emit T end
```

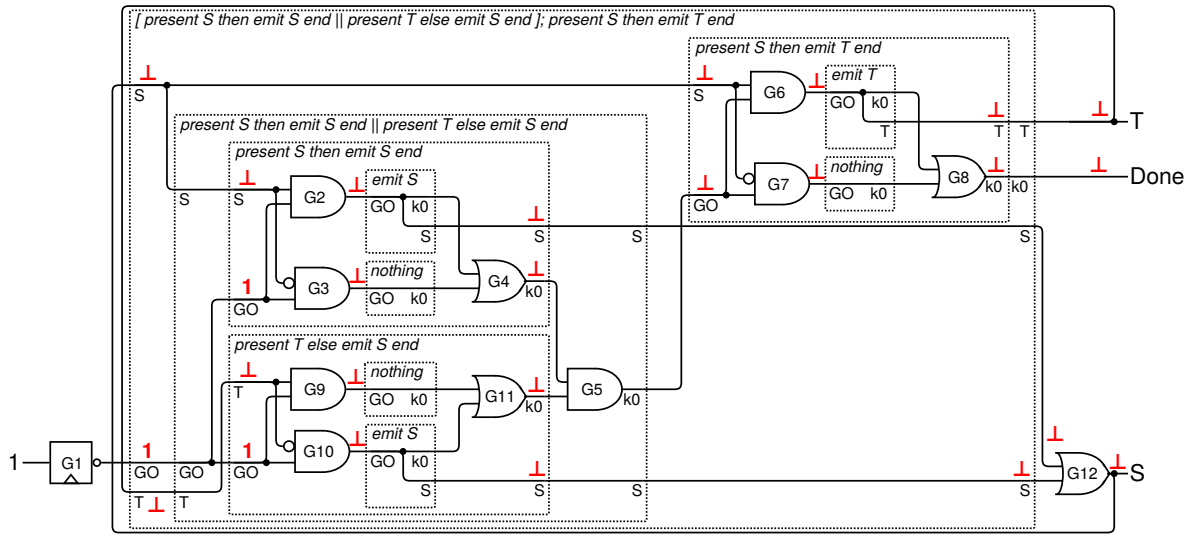
Listing 2.3: ST_B, which is SCC and also BC, results from applying the SCC2BC transformation to ST.

that are sequentially later.

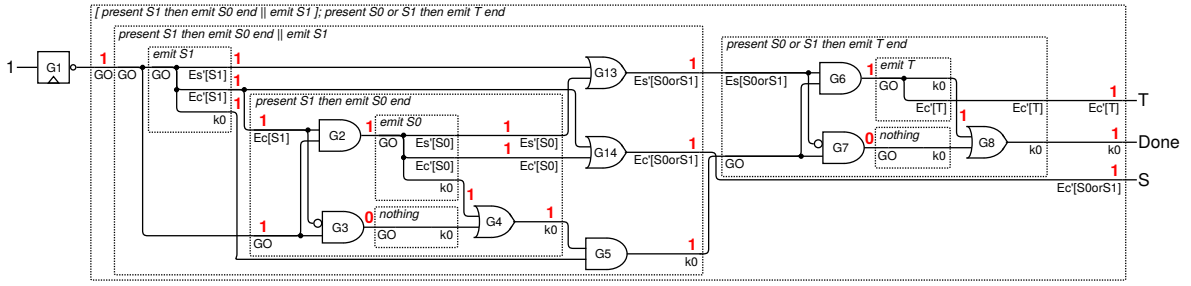
As illustrated in Fig. 2.1b, we split the signal interface of each component into sequential and concurrent inputs and outputs (E_s, E_c, E'_s, E'_c). We use E_s, E'_s to propagate signal emissions sequentially downstream and E_c, E'_c to wire up concurrent regions locally, preserving their sequential control flow relationships. Then, as seen in Fig. 2.1b, the upstream process P no longer depends on any emission from downstream statements. Any local node like R_1 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.

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. 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 have not yet a stable E'_s output, as illustrated in Fig. 2.1b. Thus E'_c is never blocked by inactive control flow, in contrast to E'_s. PingPong (Lst. 2.1) requires this separation. After the emit of Ping, 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 Pong. Hence E'_s cannot pass the emitted Ping 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. However, it is still acyclic at the signal level, and thus would be accepted also by a standard BC Esterel compiler.



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



(b) SCC. Register/gates G1–G8 correspond to BCC. G9–G12 have been removed, G13–G14 were added.

Figure 2.2: 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.

2.3 The ST Example

ST (Lst. 2.2) will serve as running example for the remainder of this report. S is (re-)emitted if S is present (line 4) in parallel to an emission of S if T is absent (line 6). Then T is emitted if S is present (line 8). This example illustrates concurrent and sequential communication in a program similar to Fig. 2.1. The reason is the global environment which has a feedback from E′ to E. The corresponding BCC circuit, depicted in Fig. 2.2a, 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 ⊥, and the status of T remains undefined. This in turn forbids to conclude a status for S. Thus ST is not BC and rejected by Esterel.

However, with the 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. 2.2b, the test of T (G9–G11) can be fully eliminated since there are no more visible emitters of T. The SCC circuit is considered 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* (see Sec. 5) for ST (line 6, then line 4, then line 8) which all lead to the same result.

Even though ST is only SCC and not BC, we can translate the SCC circuit for ST back into an Esterel program that is BC. Such a program is ST_B in Lst. 2.3. The BCC circuit for ST_B corresponds to the SCC circuit for ST. Even better, we can transform SCC programs directly into their BC equivalent, without going down to circuits, by the SCC2BC transformation presented in Sec. 3.

2.4 The SCC Circuit Rules

The circuit semantics of Esterel is defined by its translation rules and the property of a constructive circuit. The rules cover the Esterel *kernel* language and structurally translate a program. In the same manner we propose translation rules for the SCC semantics based on these BCC rules. Concerning the wiring of the control flow, the rules are identical to the Berry rules [5]. The main difference is in the handling of the environment containing the signal wires.

Fig. 2.3 and 2.4 present 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.

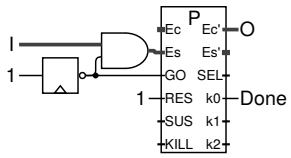
We assume that the input programs fulfill the same structural requirements as in BCC [5]. Specifically, we assume that loops are not instantaneous and that there is no statement reincarnation.

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 completion codes k0 (termination), k1 (pausing), k2 (innermost trap), k3, ... (further traps) is exactly the same in SCC and BCC.

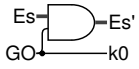
The following descriptions of the SCC rules focus on the extensions that SCC provides over BCC. For readers not familiar with the BCC Esterel circuit semantics, we briefly explain the BCC control logic as well. However, for a more detailed description we refer the interested reader to Berry [5].

Global (Fig. 2.3a)

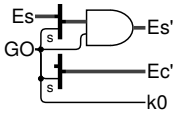
At the top level for a program P, inputs I feed into E_s when P is initially started. The inverted output of a register, which like all registers is initially 0, activates P via GO



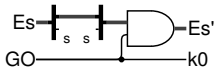
(a) Global
(**module**: P **end**)



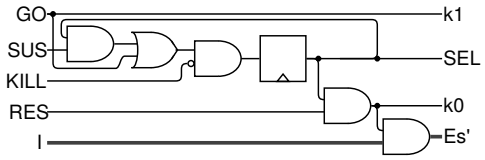
(b) Nothing (**nothing**)



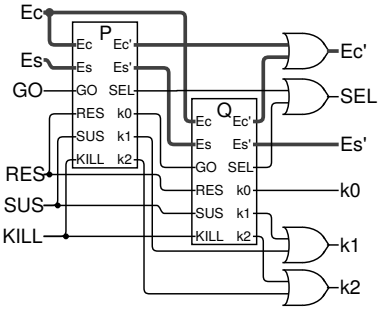
(c) Emit (**emit** s)



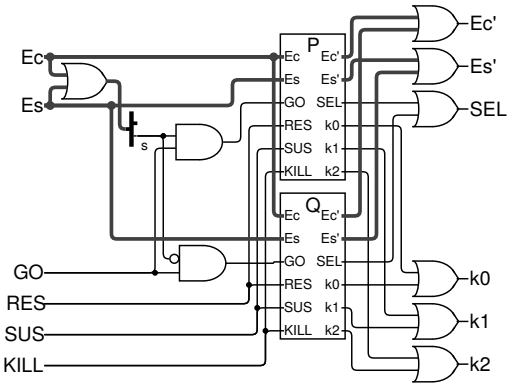
(d) Weak Unemit
(**weak** unemit s)



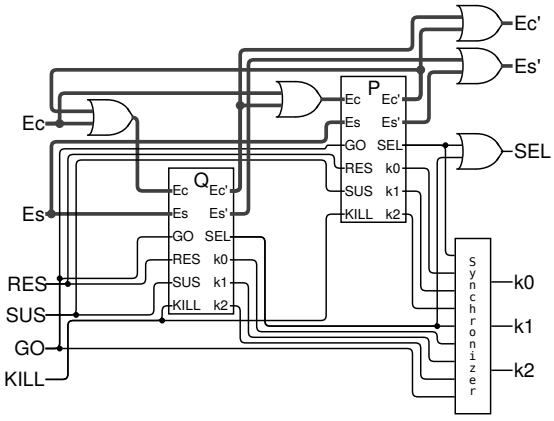
(e) Pause (**pause**)



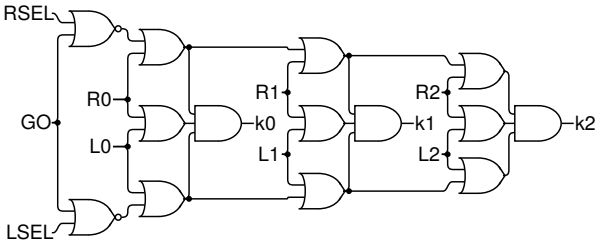
(f) Sequence (P; Q)



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



(h) Parallel (P || Q)



(i) The synchronizer used by Parallel

Figure 2.3: SCC construction rules

and enables the inputs with an AND gate. $\mathsf{E_c}$ is initialized to 0 since no signals can be emitted concurrently on this level. The outputs of P are taken from $\mathsf{E'_c}$.

According to the Esterel rules, RES is constantly 1 and SUS and KILL are 0. The outputs of the program are taken from $\mathsf{E'_c}$ since this bus represents all emitted signals. Here the environment is considered a concurrent reader on all output signals. Program

termination corresponds to completion code 0, indicated by the k_0/Done wire.

Nothing (Fig. 2.3b)

In the Esterel circuit rules, a nothing statement is translated into a wire connecting the GO input to the k_0 output. As discussed in Sec. 2.2, nothing must actively forward (i. e., potentially block) E_s. Thus, in the SCC rules, the E_s environment is additionally blocked by an AND gate such the information in the sequential environment is only propagated downstream by this component if it terminates.

Emit (Fig. 2.3c)

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. This way the emit only affects downstream readers but not sequentially preceding readers. This encodes the essential difference between SCC and BCC. The signal is also added to the E'_c bus such that the emit is also visible to concurrent readers. Additionally E'_s is again guarded by the GO wire because E_s must not be connected to E'_s if the component is not active. The circuit ignores E_c since it is not affected by concurrent emits.

Weak unemit (Fig. 2.3d)

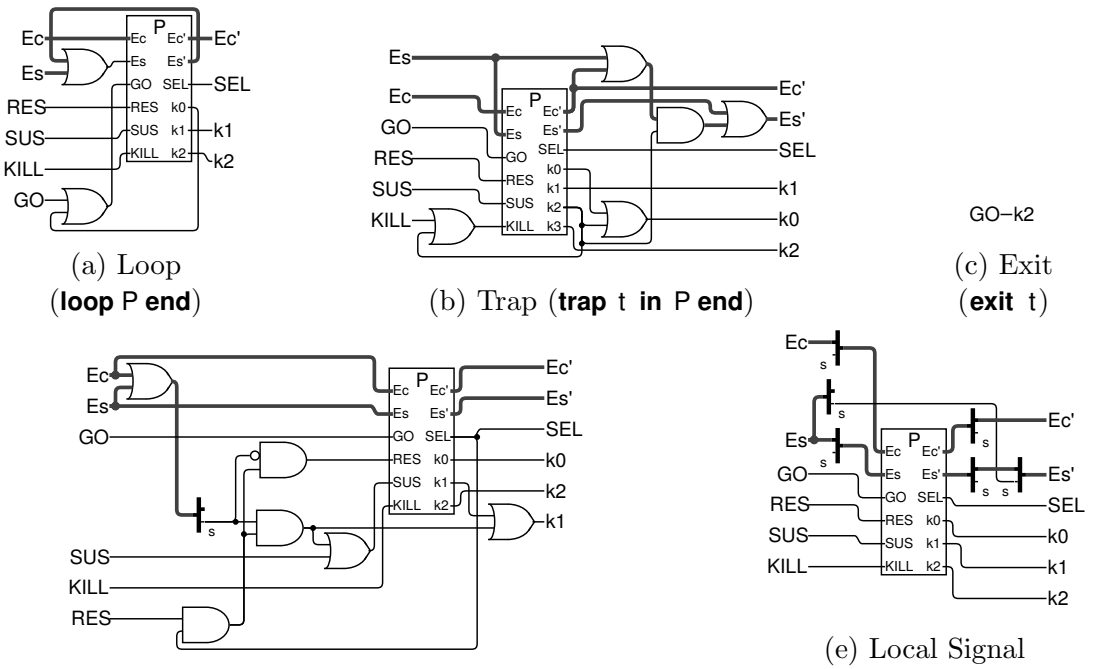
The SC MoC allows to change variable values throughout a tick. In SCEst, this has motivated the unemit statement, which is not included in Esterel [31].

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 much more complicated to do this in a concurrent context. This would require a refined version of 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 E_s, but does not affect 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. 2.3e)

The pausing logic is identical to the Berry circuit. If the pause is activated by the corresponding combination of inputs, a register is set. This register may start the further execution of the program in the next tick if it is allowed to resume. If a tick starts in a pause, indicated by k_0, the 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.



(a) Loop (**loop** P **end**)

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

(c) Exit (**exit** t)

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

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

Figure 2.4: Remaining SCC construction rules

Sequence (Fig. 2.3f)

Not surprisingly, this is the central rule to encode sequentiality, by forwarding E'_s of P to E_s of Q but not the other way around, as already illustrated in Fig. 2.1b. In a sequence, Q is started when P terminates. Q also receives the sequential environment E'_s of P. The active emits represented by E'_c in P and Q are made visible to concurrent readers independent from the termination. The incoming concurrent environment E_c and the remaining wires are passed to both components. The outgoing wires of P and Q are combined by OR gates and then passed out.

Conditional (Fig. 2.3g)

According to the SCCL and SC-visibility, the input signal that selects the branch is taken from E_c and E_s. In the conditional, the GO signal is given to one of the branch components based on the value of signal s or a boolean expression based on signals. The value is determined by a combination of E_c and E_s. This directly encodes the SC-visibility since a read is only affected by sequentially preceding or concurrent writers. The incoming environments E_c and E_s are passed to the two branch components together with the remaining incoming control wires. The outgoing wires of P and Q are combined by OR gates and then passed out.

Parallel (Fig. 2.3h)

Parallel components communicate via $\mathsf{E}'_\mathsf{c}/\mathsf{E}_\mathsf{c}$, see again Fig. 2.1b.

The parallel composition activates both components and feeds all the control wires and environments into them. The only connection between P and Q is via E_c and E'_c. P receives the incoming E_c, in case this component is also embedded in one or more parallel statements, combined with the E'_c environment of Q. Analogously for Q. The output environments of P and Q are combined by OR gates. The completion codes of both parallel components is calculated by a synchronizer logic, displayed in Fig. 2.3i. This logic computes the maximum of both completion codes.

As an aside for readers not familiar with Esterel's completion codes, the desire to compute the combined completion code k simply as the maximum of the completion codes k_i of the parallel components (where i is an index indicating the components) has motivated the encoding of the completion codes: if any component exits a trap ($k_i \geq 2$ for some i), the resulting completion code is the maximum k_i that indicates the outermost trap exited; otherwise, the parallel pauses ($k = 1$) if any parallel component pauses; only when all components terminate in the current tick ($k_i = 0$) or are have terminated before, then the whole circuit terminates ($k = 0$).

Loop (Fig. 2.4a)

In a loop structure, the loop body P is restarted when it terminates. Hence, the GO wire starts P if the loop is initially entered or P terminates, indicated by k_0. In the same manner E_s of P is either the incoming E_s or E'_s of P. The restriction to non-instantaneous loop bodies, in combination with the blocking of E'_s, prevents cyclic dependencies between GO and k_0, and E_s and E'_s. Since a loop cannot terminate normally, k_0 and E'_s are never set. Note that a loop can only be left using a trap. The remaining incoming wires are directly connected to P as well as the outgoing.

Trap (Fig. 2.4b)

A trap provides a jump structure triggered by exits. If an exit is executed, the program's execution immediately continues at the end of the corresponding trap. If the exit is in a parallel thread, the other threads are weakly aborted, meaning that they run until a pause, exit or the end of thread and then continue at the end of trap. If multiple exits are triggered, the outermost trap has precedence. In the circuit this is assured by the synchronizer in the Parallel component. When the trap is triggered in P, indicated by k_2, P receives a KILL signal to prevent pause registers from being set if they are activated in this tick. P can also be killed by an surrounding trap, encoded by the OR gate. The outgoing sequential environment E'_s is either E'_s of P, if the trap terminated normally, or 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 E'_s.

This logic is important to properly distinguish *surface* and *depth* outputs. We here follow the established Esterel terminology where the surface of a statement or program

fragment p refers to the behavior of p in the tick when p started executing, and the depth of p is its behavior in subsequent ticks.

The trap terminates, indicated by k_0, if P terminates with k_0, or k_2, if the trap is triggered. To retain the trap nesting hierarchy, all the completion codes of P are down-shifted such that the outgoing k_2 is the k_3 of P and surrounding traps can react to their correct completion code. The remaining wires are directly connected to P.

Exit (Fig. 2.4c)

The exit does not produce any E'_s, since in case of an exit the corresponding trap sequentially forwards E'_c, not E'_s.

The exit component sets the corresponding completion code if it receives a GO. In this example it triggers the innermost trap via k_2. All remaining incoming wires end in this component and it does not produce any outgoing information aside the termination code.

Suspend (Fig. 2.4d)

When suspending P based on a signal s, the state of s is determined considering both E_s and E_c, just as for the conditional.

If s is present, P is suspended by setting the SUS input. It may also be suspended by a surrounding suspend statement. However, P is only suspended by this suspend if some pause is active inside P, indicated by the SEL wire, and the suspend is allowed to resume, indicated by RES. Hence P cannot be suspended in the first tick when the suspend scope is entered. If s is absent and no surrounding suspend is active P can resume by setting RES. The suspend statement will indicate pausing via k_1 if P pauses or P is suspended. The remaining wires are directly connected to P.

Local Signal (Fig. 2.4e)

This creates a new scope for a wire s. P receives the environments E_s and 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 E'_s.

3 The SCC2BC Transformation

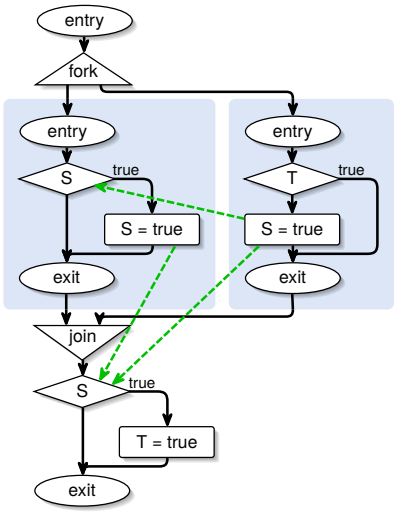
One option to compile SCC Esterel programs is to synthesize them into netlists, according to the rules presented in Sec. 2.4, and to either simulate these in software or create actual hardware from that. This, however, would require to re-do much of the engineering work of existing Esterel compilers. This concerns in particular the rather sophisticated constructiveness analysis present in the Esterel v5 compiler [36] for handling statically cyclic programs. The SCC2BC transformation presented now avoids this by translating SCC Esterel programs into equivalent BC Esterel programs. SCC2BC is minimally disruptive in that Esterel programs that are already BC undergo minimal changes, if any, even if they are statically cyclic. The key concept is to express the concept of SC-visibility at the Esterel level. We do so by (1) splitting signals into different versions, one for each signal emission (circuit in Fig. 2.3c), and (2) disjuncting signal versions according to their SC-visibility scopes whenever they are tested (see circuits in Fig. 2.3f/2.4d). This is akin to the well-known static single assignment (SSA) paradigm [13]; however, we have to extend SSA to properly handle tick boundaries and Esterel's concurrency and pre-emption operators.

The concept of SSA is to split up and rename variables, such that each assignment to the same variable assigns different versions of this variable in the SSA form. To provide a single reaching definition for each referenced variable, ϕ-functions select the value from the different variable versions, based on the last executed assignment in the active incoming control flow path. The concept of SSA allows to bypass the limitation of a single globally consistent state for each signal in each tick present in Esterel. Since SSA is developed for sequential control flow graphs, it provides the correct visibility of signals depending on the sequential location of a read to its writers.

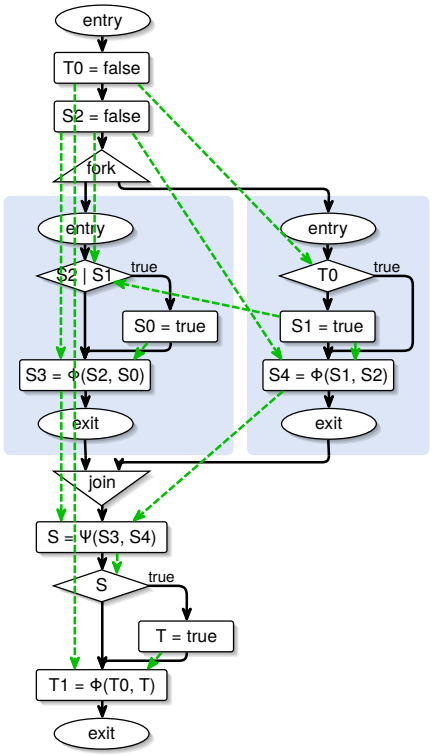
The SCC2BC transformation is separated into three steps:

1. Creating a control flow representation of the source Esterel program.
2. Performing an SSA transformation specially adjusted for this use case.
3. Translating the SCC Esterel program into BC Esterel using the SSA information.

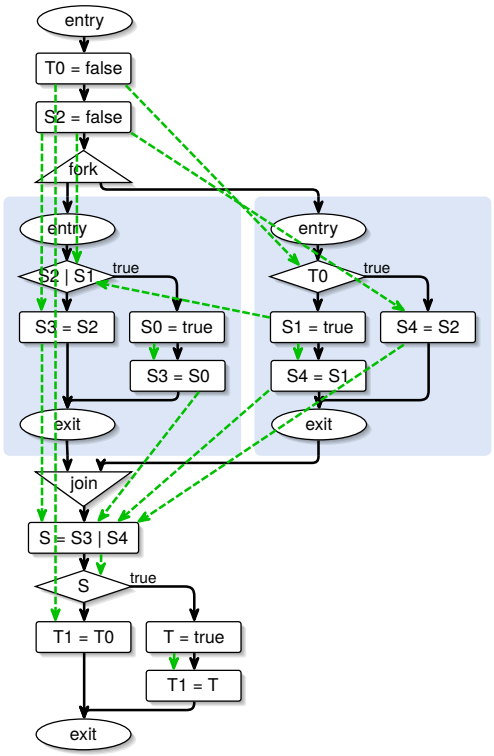
These steps are described in detail in the following sections. To illustrate the effect the ST program from Lst. 2.2 is translated.



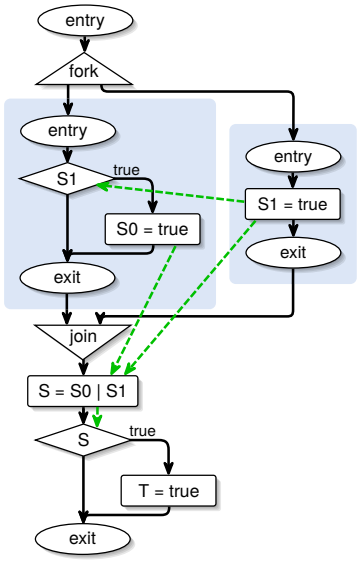
(a) SC graph (SCG) of ST



(b) With ϕ- and ψ-functions



(c) Transformed ϕ- and ψ-functions



(d) Optimized SCC2BC result

Figure 3.1: Stepwise SCC2BC transformation of the ST example represented by a control flow graph with dependencies.

3.1 Control Flow Representation

The SCC2BC transformation is based on a static analysis of the control flow of a program. Consequently, the first step is an intermediate representation of the source Esterel program as a control flow graph. Since Esterel is a synchronous language and includes explicit concurrency, it requires an extended controlfow graph notation representing pauses and threads. The SCC2BC transformation uses the SC Graph (SCG) [18] notation to analyze sequential and concurrent control flow. We also allow the SCG to leave a thread without an exit to correctly represent Esterel traps, which is not allowed in its original definition. Specifically, to capture the concept of trap exits, we do not require jumps to be thread-local anymore as the original SCG does. We also do not capture all semantic information of the original Esterel program, such as suspension scopes, but that is acceptable since we use the SCG only for the purpose of the SCC2BC transformation, we do not generate code from the SCG directly. Fig. 3.1a shows the SCG for ST (Fig. 2.2). The program start and end is represented by entry and exit nodes. The parallel statement is represented by a fork and a join node which spawn and join the two concurrent threads. The entry and exit nodes indicate the regular sequential start and end point of these threads.

Present tests are transformed into conditional nodes and emits result in assignment nodes. The signals are represented as boolean variables, with a true value indicating presence. Hence, emits assign true and unemits would result in assignments to false. Since these booleans represent signals they are implicitly set to false at the start of each tick.

In addition to the constructs illustrated in Fig. 3.1a, an SCG may also contain surface/depth nodes that correspond to Esterel's pause. Loops are simply represented by cyclic control flow in the graph. The SCG has no direct representation for suspension. However, for SCC2BC it suffices to add an assignment node to the SCG that computes the suspend expression to a temporary variable, at the beginning of the suspension scope.

Similarly, we emulate trap/exit with jumps from the exit to the end of the trap. In a completely sequential context, an exit is a simple jump, but if the exit is located in a concurrent thread and the end trap is located after the join of this thread, an exit causes the concurrent threads to be weakly aborted. This means they execute until the end of thread, an exit, or the next pause statement. However, the control flow does not end there, but it is joined with the thread(s) executing the exit and continues at the end of the trap. Lst. 3.1 illustrates such a trap in a concurrent context. If I is present, the first thread will emit B and then trigger the trap. If I is absent, the thread pauses. The other thread always emits A. After the parallel section, C is emitted if A and B are present. In the SCG representation in Fig. 3.2 the control flow of the trap is explicitly modeled. This is important for correctly analyzing the visibility of emissions to downstream statements. Algorithm 1 shows the pseudocode procedure to translate a trap and the corresponding exits into its SCG representation. Before all pause and exit nodes concurrent to an exit, there is a conditional node jumping to the exit. The exit itself results in an assignment triggering these conditionals. Then the control flow of the exiting and aborted thread is joined and continues at the first node after the end trap. Furthermore, there are

```
1  module TrapExample:
2  input I;
3  output A, B, C;
4  trap t in [
5    present I then
6      emit B;
7      exit t
8    else
9      pause end
10 ||
11   emit A
12 ]; end trap;
13 present A and B then
14   emit C end
```

Listing 3.1: The TrapExample with a trap and an exit in concurrent threads.

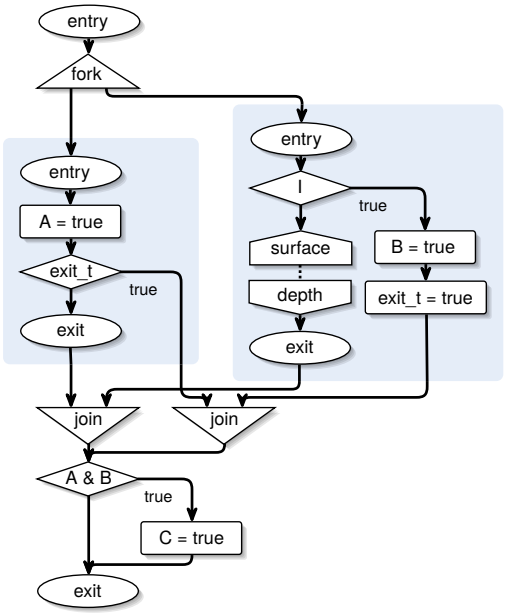


Figure 3.2: The SCG representation of TrapExample

precedences between traps according to their nesting hierarchy. Hence, an executed exit also has to pass its control flow to an exit of a surrounding trap if both are executed, resulting in corresponding conditional nodes before these subordinate exits.

3.2 Extending SSA to SCSSA

The SCG representation is transformed into SSA form using a standard dominator analysis [13]. However, to handle SCGs that represent Esterel programs, the SSA form must be extended. There are two aspects which are not considered by a classical dominator analysis and SSA transformation. First, the synchronous paradigm uses pauses that have an implicit effect on signals. To emulate signal initialization to absence in SCC2BC, all boolean variables representing signals are assigned to false at the start of the program and after each pause. This corresponds to the initialization of $\mathsf{E_s}$, see Fig. 2.3a and Fig. 2.3e. In Fig. 3.1b this occurs directly after the entry node, where T0 (version 0 of signal T) and S2 are set to false. Note that the version numbering in this example differs form the textual order to provide consistent version numbers throughout the transformation.

The second aspect is concurrency. Algorithm 2 presents a pseudocode sketch of the adapted SSA algorithm handling this aspect. The SSA transformation introduces ϕ-assignment nodes when two or more control flow paths join with different definitions of the same variable to set a new dominant version of the variable. The SCG in Fig. 3.1b contains such ϕ-assignments when conditional branches merge. The ϕ-function is defined such that it selects the incoming definition based on the active incoming control flow. This requires that only one incoming control flow is taken. However, in case of joining

Algorithm 1 Translation of Esterel traps into SCG representation

```
1: procedure TranslateTrap(trap t_s trap <body> end trap)
2:     Create new variable exit_t_s
3:     Translate body                          ▷ This includes further translation rules
4:     for exit assignment node e in body do
5:         if fork(thread(e)) is in nodes of body then
6:             Create join node j
7:             Set outgoing control-flow of j to node after end trap
8:             Set outgoing control-flow of e to j
9:             for node n in nodes of sibling threads of thread(e) do
10:                if n is surface node or exit node or
11:                    assignment exit_t_k = true with t_k != t_s then
12:                    Create conditional node c and insert before n
13:                    Set condition of c to exit_t_s == true
14:                    Set then-branch control-flow of c to j
15:                    Set else-branch control-flow of c to n
16:                end if
17:            end for
18:        else
19:            Set outgoing control-flow of e to node after end trap
20:        end if
21:    end for
22: end procedure
23: procedure TranslateExit(exit t_s)
24:    Create assignment node with exit_t_s = true
25: end procedure
```

threads, due to termination or the exit of traps, more than one control flow may be active and must be considered. This problem corresponds to the concurrent SSA form by Lee et al. [21]. Accordingly, we introduce a ψ-function node after the join node. Corresponding to the definition of the parallel, already seen in Fig. 2.3h, our ψ-function disjunctively combines all incoming versions. Represented in lines 3 to 8 of algorithm 2. According to the SC-visibility, all concurrent reads of a signal are replaced by a disjunction of the reaching definition and all concurrent definitions, illustrated in Fig. 3.1b by the conditional node testing S in the left-hand thread. The algorithm handles this in lines 9 to 17. A dependency analysis provides the necessary information for this transformation, the sequential and concurrent dependencies are visualized as dashed arrows in the SCG.

Handling implicit initializations and concurrency results in a new, SC-specific SSA variant that we refer to as *SCSSA*. The ϕ- and ψ-nodes introduced by SCSSA must be translated further into executable Esterel code. Each ϕ-function is transformed into multiple assignments, one in each of the incoming control flow paths it combines. Each assignment assigns the incoming definition of this path to the new version. Fig. 3.1c illustrates the result of this ϕ-assignment transformation and lines 18 to 24 represent

Algorithm 2 SCSSA transformation

```
1: procedure SCSSA(SCG g)
2:     ConvertToSSA(g)                                  ▷ Regular SSA transformation
3:     for φ-node n with s_i = φ(s_j0, ..., s_jk) in g do   ▷ Convert to ψ-nodes
4:         if n is direct predecessor of a join-node j then
5:             Set s_i = s_j0 | ... | s_jk
6:             Move n behind j
7:         end if
8:     end for
9:     for assignments a in g do                        ▷ Handle concurrent emits
10:        for incoming concurrent data dependency d of a do
11:            for signal reference s_i in a do
12:                if source of d assigns s_j then
13:                    Replace s_i with expression (s_i | s_j)
14:                end if
15:            end for
16:        end for
17:    end for
18:    for φ-node n with s_i = φ(s_j0, ..., s_jk) in g do   ▷ Transform φ-nodes
19:        for l in 0 to k do
20:            Create assignment node a with s_i = s_jl
21:            Insert a in incoming control-flow branch for s_jl of n
22:        end for
23:        Remove n
24:    end for
25:    ConstantPropagation(g)                           ▷ Reduce # of new assignments
26:    MergeRedundantSignalVersions(g)                  ▷ Reduce # of signal versions
27: end procedure
```

this procedure in the algorithm. This transformation violates the SSA property in that it assigns the same variable multiple times. However, based on the assumption of non-instantaneous loops and the exclusion of schizophrenia, only one of the control flows will be active in a tick and only one assignment is effectively executed. Thus a single assignment of the signal is still preserved by this form. The ψ-functions are transformed into OR expressions based on their definition, see Fig. 3.1c.

We apply constant propagation to further reduce the number of new assignment nodes that simulate the ϕ-functions. All assignments to false are removed since the effect of setting a signal to absent is achieved by not emitting it. A weak unemit is realized by simply introducing a new variable version for which there is no emit. References to variable versions that are known to be false, i. e. absent, are removed and the corresponding expressions are partially reduced. This also allows to eliminate dead conditional branches, similar to a sparse conditional constant propagation [38].

As an optimization, we merge signals that have the same set of readers and are always

disjuncted. To detect such signals, a table is created for each original signal, where each version of the signals results in a column and the rows represent reading statements in the program. Identical columns indicate redundant signal versions. Each cell gets marked with a 1 if the version occurs in the read and 0 otherwise. Sorting the columns will group the versions for merging, since all versions which share the same vector can be reduced to one version. Algorithm 3 presents the pseudocode procedure performing this optimization.

Finally, the last assigned versions are renamed to their original signal names to match the interface. In programs with pauses, especially in parallel threads, this may require additional assignments. The final SCG for ST is seen in Fig. 3.1d.

Algorithm 3 Merging redundant signal versions

```
1:  procedure MergeRedundantSignalVersions(SCG g)
2:      for signal s with versions s_0 to s_i in g do
3:          Create table t with i columns
4:          for node n in g do
5:              Add new row j to t
6:              for k in range 0 to i do
7:                  if n references signal s_k then
8:                      Set t[s_k][j] = 1
9:                  end if
10:             end for
11:         end for
12:         Sort t by rows
13:         k = 0
14:         while t has column k do
15:             if t[k] == t[k + 1] then
16:                 for node n in g do
17:                     Replace all references to s_k by s_{k+1}
18:                 end for
19:                 Remove column k + 1
20:             else
21:                 k = k + 1
22:             end if
23:         end while
24:     end for
25: end procedure
```

3.3 SCC2BC at the Esterel level

Based on the control flow graph representation in SCSSA form, we can translate the SCC Esterel program into BC Esterel. This requires the correct association of nodes in the

SCG with the statements in the source Esterel program. Our KIELER tool (Sec. 3.5) provides compiler infrastructure with integrated tracing capabilities to produce these associations.

We so far discussed SCC2BC at the SCG level. To apply the transformation at the Esterel source program, we first add the additional signal versions to the program. Only effectively used versions are added, with ascending indices. Next, all emits are renamed according to the affected version. The expressions of the present tests are changed to the reaching signal versions. The expressions of suspend statements are replaced by the expressions in the corresponding suspend assignment nodes. Constant assignments to true are translated into emits. Assignments to false are removed, since signal absence is implied in Esterel. All assignment nodes that are introduced by SCC2BC are added. Other assignments, for example resulting from ψ-nodes, are translated into present tests with the assigned expression and a guarded emit of the assigned signal.

Lst. 2.3 presents the final Esterel transformation result for ST. The result of this SCC2BC transformation is a syntactically valid Esterel program with the behavior defined by the SCC semantics of the source Esterel program. It provides the desired sequential write and read behavior of Esterel using multiple signals versions. The Esterel compiler can be used to translate this program into software or a circuit and check its constructiveness or deploy to a target platform.

Corresponding to the conservativeness of the SCC semantics, an emitted variable version is only removed from a referencing expression if the emit does not sequentially reach the present statement. Furthermore, all added emits only re-emit signals that were already emitted in some other version. Hence, if all version are again collapsed into one signal, then the original Esterel semantics is restored, apart from the optimization mentioned in the last step of SCC2BC.

In contrast to the dataflow based compilation for SC programs [18] or other compilation approaches for circuits [30], this approach does not require a statically acyclic programs structure. Hence, also programs which are only dynamically acyclic, such as the token ring arbiter, are supported [29].

3.4 Schizophrenia

Schizophrenic behavior of a program 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 detailed by Berry [5], we distinguish *signal reincarnation*, where a signal scope is left and re-entered instantaneously due to a surrounding loop, and *statement reincarnation*, where a statement such as a signal emission is executed multiple times within a tick. The SCC implementation of parallel provided here is sensitive to the latter form of schizophrenia because it cannot distinguish emits in sequential thread incarnations.

Consider ThreadReinc (Lst. 3.2), which is not BC, because in the second tick when the pauses resume, the test for S (line 9) in the current loop iteration blocks on the sequentially later emission of S (line 5) in the next loop iteration. ThreadReinc is SC (S is

```
1  module ThreadReinc:
2  output O, S;
3  loop
4    [
5      emit S;
6      pause
7    ||
8      pause;
9      present S then
10       emit O end
11   ]
12 end
```

Listing 3.2: ThreadReinc, with S accessed across sequentially ordered thread instances.

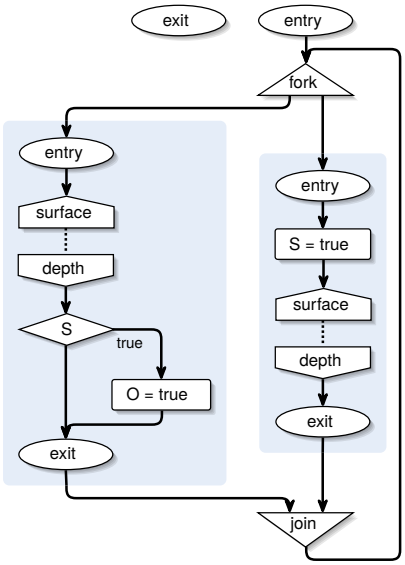


Figure 3.3: The SCG representation of ThreadReinc

considered absent) and does not require speculation. However, in the corresponding SCC circuit, loop body incarnations cannot be distinguished. The reason is the concurrent environment E_c which passes the signal S between the threads disregarding sequentially ordered incarnation of the the threads. Hence, the presence test of S would again block on the concurrently emitted S, thus ThreadReinc is not SCC. As a consequence, we require to cure schizophrenic parallels before the SCC translation is applied, just as in BCC.

However, as illustrated with the SignalReinc (Lst. 1.4) example in the introduction, the SCC2BC transformation does cure signal reincarnation by separating signal wires, without duplicating program logic.

3.5 Implementation and Validation

The compilation concepts presented in this section are fully implemented in the Eclipse-based open source KIELER tool. Fig. 3.4 shows a screenshot of the tool in use. In the editor on the left, the ST program is open. The Compiler Selection view on the bottom left controls the compiler and shows the compile chain for SCC2BC. In this case it is configured to transform the program into an SCG in optimized SSA form. The result is generated on the fly and an automatically layouted diagram of the SCG is displayed in the Diagram View on the right. The SCG corresponds to Fig. 3.1d but the side-bar on the right is used to configure the displayed diagram such that it does not contain the data-dependencies.

The tool is also used to create the circuits shown in Fig. 2.2, 2.3, and 2.4. This includes the diagram synthesis and an automatic layout.

Since a major objective of the SCC2BC transformation is to make Esterel programs

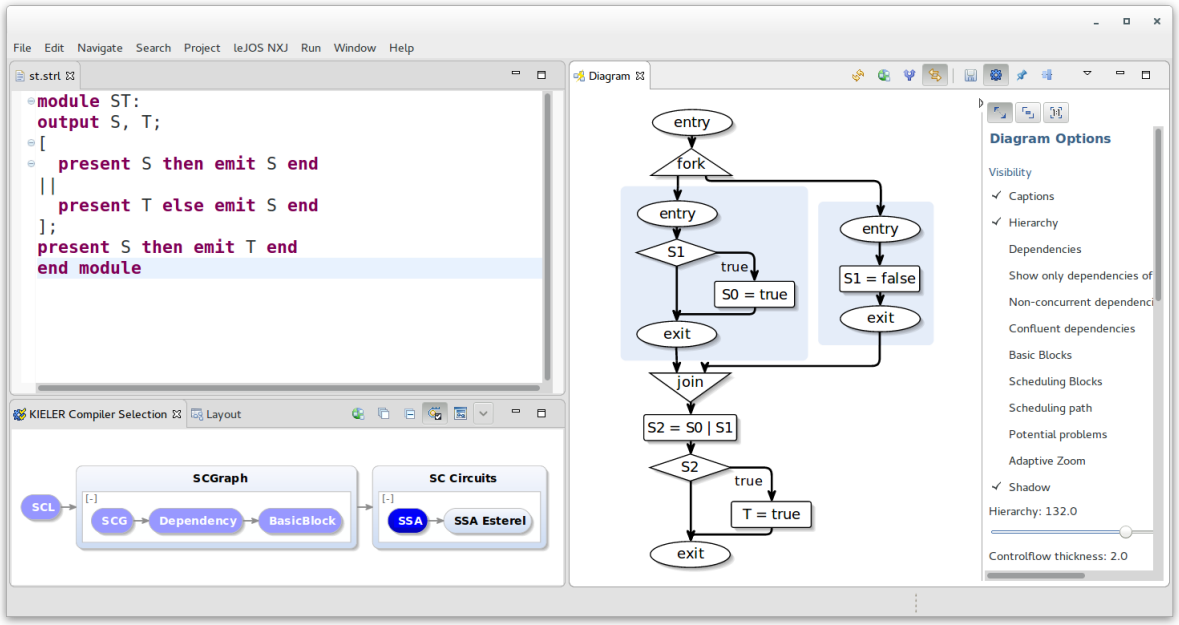


Figure 3.4: Screenshot of the KIELER implementation of the SCC2BC transformation, showing ST and its intermediate SCG representation.

compilable that are not accepted by existing Esterel compilers, it is a bit difficult to perform a meaningful quantitative evaluation based on existing Esterel programs. However, as experimental validation, we subjected a set of 134 Esterel programs to the SCC2BC tranformation. The programs consisted mainly of synthetic test programs. The individual programs were rather small, with an average size of 7 logical statements and about 3 signals, where we count for example **signal** A **in** ... **end signal** as one statement (excluding the body) and one signal. The programs were designed to cover a broad range of control flow scenarios.

We follow standard practice of defining the SCC2BC transformation for the Esterel kernel statements, which constitutes the semantic basis of the whole language. However, we also extended the tranformation to handle some common derived statements directly, such as await, to avoid excessive statement expansions. The experimental results yield that 28 of the 134 inspected programs were affected by the transformation. All other

Program	Original	SCC2BC	SCC2BC Opt.	Tardieu [37]	Note
ST	5 (2)	7 (4)	6 (4)	5 (2)	Original not BC
SignalReinc	6 (2)	6 (3)	4 (2)	11 (3)	Schizophrenic
TrapExample	11 (4)	11 (4)	11 (4)	11 (4)	Exception
ABRO	8 (4)	8 (4)	8 (4)	8 (4)	Non-kernel statements

Table 3.1: Selected results of the SCC2BC experiment. Values are given as number of logical statements (signals) in program.

programs did not change after the transformation, which is in line with our objective to be minimally invasive. Most of the programs affected by the transformation contain schizophrenic signals. For these, SCC2BC added on average 1.3 statements and 1.8 signals. Table 3.1 shows the detailed results for some programs of interest. The columns list the program name, the original size as number of logical statements (signals) in the program, and the size after the SCC2BC transformation. Since some signal versions can be removed when eliminating dead code, the optimized results are also listed. To give a rough quantitative comparison, the resulting code size when applying the approach of Tardieu et al. [37] is presented.

4 Formal Semantics and Conservativeness

We now formalize the notion of SCC with the goal of showing conservativeness relative to BC. Our formal semantics follows Berry [5] 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 statement reincarnation, specifically that statements from concurrent threads can never appear in sequential program order. We further assume that the sequential order in which two statements can appear is statically fixed. Under this assumption, which does not restrict expressiveness, a static order, SC-visibility, can be defined on the statements corresponding to the flow dependency analysis used in the SCSSA transformation (Sec. 3.2).

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. 2.2a implementing ST from Fig. 2.2.

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

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

$$\frac{w \in \mathcal{I}}{w \hookrightarrow_\pi I(w)}\ IN \qquad \frac{w \in \mathcal{R}}{w \hookrightarrow_\pi R(w)}\ REG \qquad \frac{e \hookrightarrow_\pi b}{\neg e \hookrightarrow_\pi \neg b}\ OP_\neg \qquad \frac{b \in \mathbb{B}}{b \hookrightarrow_\pi b}\ OP_c$$

$$\frac{e_1 \hookrightarrow_\pi 0 \quad e_2 \hookrightarrow_\pi 0}{e_1 \vee e_2 \hookrightarrow_\pi 0}\ OP_{\neg\vee} \qquad \frac{e_1 \hookrightarrow_\pi 1}{e_1 \vee e_2 \hookrightarrow_\pi 1}\ OP_{l\vee} \qquad \frac{e_2 \hookrightarrow_\pi 1}{e_1 \vee e_2 \hookrightarrow_\pi 1}\ OP_{r\vee}$$

$$\frac{e_1 \hookrightarrow_\pi 1 \quad e_2 \hookrightarrow_\pi 1}{e_1 \wedge e_2 \hookrightarrow_\pi 1}\ OP_\wedge \qquad \frac{e_1 \hookrightarrow_\pi 0}{e_1 \wedge e_2 \hookrightarrow_\pi 0}\ OP_{l\wedge} \qquad \frac{e_2 \hookrightarrow_\pi 0}{e_1 \wedge e_2 \hookrightarrow_\pi 0}\ OP_{r\wedge}$$

Figure 4.1: Visibility-Restricted Constructive Evaluation Rules. The evaluation context $\mathcal{C}, I, R$ is implicit.

The gates G2, G6, G10, G12 arise from wire definitions

$$S[G2] \Leftarrow_{G2} GO[G1] \wedge S[G12] \tag{4.1}$$
$$T[G6] \Leftarrow_{G6} S[G12] \wedge \mathsf{k}_0[G5] \tag{4.2}$$
$$S[G10] \Leftarrow_{G10} GO[G1] \wedge \neg T[G6] \tag{4.3}$$
$$S[G12] \Leftarrow_{G12} S[G10] \vee S[G2]. \tag{4.4}$$

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. 2.3 and 2.4) represented by the wire. The visibility ordering $\preceq$ is obtained from the control flow of the source program in Lst. 2.2. Since G10 comes from line 6 and G6 comes from line 8, 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 tests in lines 4 and 6 of Lst. 2.2. 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

$$\mathcal{C}, I, R \vdash e \hookrightarrow b \tag{4.5}$$

which evaluates a boolean expression e over $\mathcal{W}$ using the evaluation rules of Kleene ternary algebra (see e.g. [34]), 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

$$\mathcal{C} \vdash I, R \hookrightarrow O, R' \tag{4.6}$$

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

$$\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b \tag{4.7}$$

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 π are sequentially incomparable visibility indices (π 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. 4.1. For notational compactness we write $e \hookrightarrow_\pi b$ instead of (4.7) when the evaluation context $\mathcal{C}, I, R$ is clear.

The standard ternary evaluation of boolean expressions is implemented by the *OP* rules, which do not depend on the observation points π. Rules *IN* and *REG* are the evaluation of inputs and register wires. The visibility information π 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 π-*visible*, written $\pi \not\preceq l$, if l does not lie downstream from any observation point in π, 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 π or shifts to l otherwise. More precisely, $\pi \oplus l = \pi \setminus \{m \in \pi \mid l \prec m\} \cup \{l\}$. Note that if we would drop π and simply use l to evaluate e, we might eventually jump back to a wire (gate) that is downstream from some observation point in π. This is what we avoid if we preserve π 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 π-visible evaluate to 0. We add the relevant parameters π, l, w to the rule names for ease of reference.

The visibility information enters the evaluation rules named $PRES(\pi, l)$ 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. 4.1 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 [5].

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 contravariantly. E.g., changing π_1 to π_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.

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

Prop. 1 shows that restricting visibility is conservative for the BC circuits. Wires that have a decided value under BC will also stabilise to the same values under visibility restrictions. This highlights the key feature of Esterel circuits: If a signal stabilises then all sequentially downstream circuitry can be removed without changing the value. Suppose in a sequential composition $P; Q$ a wire w instantiated from P stabilises, i.e., $BCC(P; Q), I, R \vdash w \hookrightarrow_{\{m\}} b$ where m is a visibility index in P. Then by Prop. 1 we also have $BCC(P; Q), I, R \vdash w \hookrightarrow_{\{\infty(P)\}} b$, where $\infty(P)$ is the final index of P in the sense that $m \preceq \infty(P)$ and all wire labels l of Q have $\infty(P) \preceq l$. This in turn means $BCC(P), I, R \vdash w \hookrightarrow_{\{\infty(P)\}} b$ and thus $BCC(P), I, R \vdash w \hookrightarrow b$.

An asymmetry of Esterel circuits, however, lies in the fact that Prop. 1 is not invertible. Although a stabilising value does not depend on downstream statements, stabilisation itself does. Visibility restrictions can assign values to wires that have no value under BC. For instance, consider the gates G2, G6, G10, G12 from Fig. 2.2a with definitions (4.1)–(4.4). These definitions form a feed-back cycle in which no signal stabilises under ternary simulation. This is indicated by the $\perp$ values on the wires in Fig. 2.2a. However, if we apply our visibility-restricted evaluation rules of Fig. 4.1, we find that T stabilises, assuming $GO[G1] \hookrightarrow 1$. In fact, T stabilises in different ways, depending on the observation point, viz. $T[G6] \hookrightarrow_{\{G9\}} 0$ and $T[G6] \hookrightarrow_{\{G8\}} 1$. This corresponds to the two readings of the value of T in ST (Lst. 2.2) under SC semantics: The present test in line 6 (visibility $G9$) sees $T = 0$ while in line 8 (visibility $G8$) the signal is emitted and thus $T = 1$. Formally, there is only one wire equation (4.2) with visibility index $G6$. Hence for any observation points π with $\pi \preceq G6$ the definition is switched off from

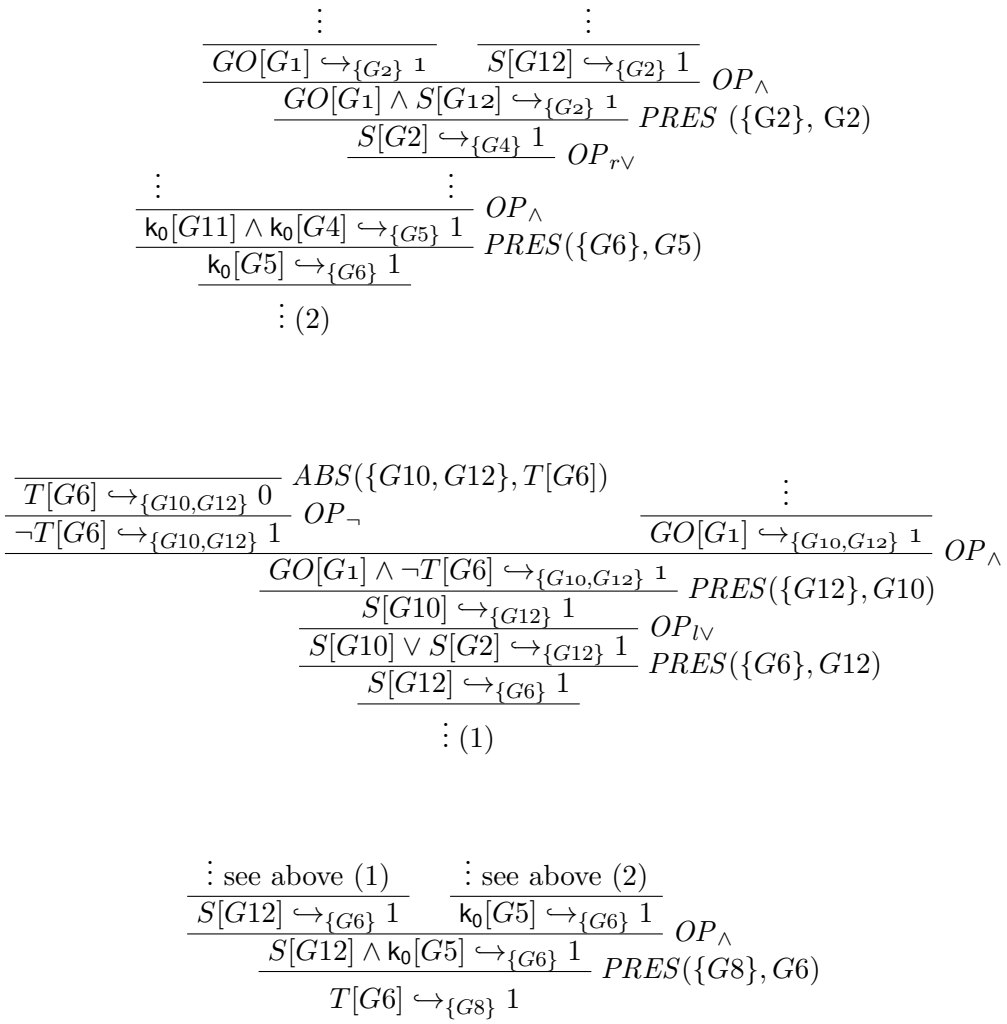


Figure 4.2: Visibility-restricted constructive evaluation of BCC (fragment)

the view of $ABS(\pi, T[G6])$ which happily (having no proof obligations in the premises) derives $T[G6] \hookrightarrow_\pi 0$. From there, $T[G6] \hookrightarrow_{\{G8\}} 1$ follows as seen in Fig. 4.2.

The difference in the evaluations $e \hookrightarrow b$ and $e \hookrightarrow_\pi b$ arises from wire definitions forcing evaluation against the visibility order. Let $\rightarrow_\mathcal{C}$ and $\twoheadrightarrow_\mathcal{C}$ denote the direct and transitive (instantaneous) dependency relations, respectively, between wires in circuit $\mathcal{C}$. A circuit $\mathcal{C}$ is called *flow-oriented* if for any two wire definitions $w_1 \Leftarrow_{l_1} e_1$ and $w_2 \Leftarrow_{l_2} e_2$ with $w_1 \twoheadrightarrow_\mathcal{C} w_2$, i.e., w_2 depends on w_1, we have $l_2 \not\preceq l_1$. Evaluation dependencies in flow-oriented circuits do not make forward references in the $\preceq$ ordering. This has the effect that evaluating $\mathcal{C}$ under visibility constraints gives the same result as evaluating it under the standard BC rules.

Proposition 2. *Let $\mathcal{C}$ be a flow-oriented circuit and $\pi \subset \mathcal{F}$ 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 $\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b$.*

$BCC(P)$ is not flow-oriented because of the feedback loops which make downstream emissions propagate backwards against the program order. In Fig. 2.2a such a violation occurs in the wire definitions (4.2) and (4.3) with wires $T[G6] \twoheadrightarrow S[G10]$ while $G10 \preceq G6$. In contrast, the $SCC(P)$ in Fig. 2.2b is flow-oriented. Specifically, the $T[G6]$ output of $G6$ is not wired back to the gates that implement sequentially upstream program blocks.

As a corollary of Prop. 2, thus, adding visibility restrictions in the evaluation of $SCC(P)$ does not have any effect. It turns out that under the visibility-restricted constructive evaluation $SCC(P)$ and $BCC(P)$ circuits are equivalent. To relate them we need to wire them up so their interfaces have a common form. We must form a global feed-back loop for the local signals of $BCC(P)$ to make concurrent threads communicate. Specifically, we take $BCC(P), E \Leftarrow_0 E_c \vee E', E'_c \Leftarrow_0 E'$, where 0 is a flow index concurrent to all indices in $BCC(P)$. The feedback $E \Leftarrow_0 E_c \vee E', E'_c \Leftarrow_0 E'$ added around $BCC(P)$ can be traversed by the constructive evaluation under visibility constraints, because $l \not< 0$ for any index l used inside $BCC(P)$. The final Theorem 1 states that if $BCC(P)$ stabilises an output signal $\mathsf{E}'_\mathsf{c}.s$ then $SCC(P)$ must also stabilise this signal on E'_c with the *same* value. This implies conservativeness of SCC over BCC. It also shows how the standard constructive semantics of the new SCC circuits can be obtained from the existing BCC circuits by the visibility-restricted constructive evaluation relation defined in Fig. 4.1.

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 $s \in Sig$ and observation points $\pi \subset \mathcal{F}$. Then,

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

Thm. 1 together with Prop. 2 shows how the standard constructive semantics of the new circuit translation $SSC(P)$ can be obtained for the standard Berry circuit $BCC(P)$ by the visibility-restricted constructive evaluation relation defined in Fig. 4.1. The SCSSA translation of Secs. 3.2 and 3.3 implements the visibility restriction at the source program level by the ϕ and ψ-functions.

Compare the BCC circuit (Fig. 2.2a) and the SCC circuit (Fig. 2.2b) generated for the ST example program regarding signal S. There are two emitters, wire definitions (4.1) and (4.3) from lines 4 and 6, respectively, of Lst. 2.2. The readers are the evaluations at gate $G2$, $GO[G2] \Leftarrow_{G2} GO[G1] \wedge S[G12]$ from the test in line 4 of Lst. 2.2, and at gate $G6$, $GO[G6] \Leftarrow_{G6} S[G12] \wedge \mathsf{k_0}[G5]$ from line 8. Where $BCC(ST)$ feeds all readers of signals S with a global OR (4.4), $SCC(ST)$ feeds gate $G2$ with a different emitter ("$S1$") than gate $G6$ ("$S0$ or $S1$"). This separation is a result of taking visibility into account. The signals S, $S0$ and $S1$ correspond to $S[G12]$, $S[G2]$ and $S[G10]$, respectively. One finds that for the observation point $\pi_1 = \{G2\}$ of $G2$, the evaluation of $S[G12]$ reduces

to the evaluation of $S[G10] \vee S[G2]$ with $\pi_1' = \pi_1 \oplus G2 = \{G2, G12\}$. Under π_1' the wire definition (4.1) for $S[G2] = S0$ is blocked and the disjunction reduces to $S[G10] = S1$. In contrast, with $\pi_2 = \{G6\}$ the evaluation of $S[G12]$ reduces to the evaluation of $S[G10] \vee S[G2]$ with $\pi_2' = \pi_2 \oplus G12 = \{G12\}$. Since $G12$ is $\prec$-incomparable with both $G10$ and $G2$, both terms $S[G10] = S1$ and $S[G2] = S0$ remain in the disjunction.

Note that the conservativity result is a consequence of all three results, Prop. 1, Prop. 2 and Thm. 1: For every Berry constructive program P, evaluating the $SCC(P)$ circuit of Sec. 2.4 under the standard BC semantics give the same behaviour as evaluating its Berry circuit $BCC(P)$.

4.1 Proofs for Conservativeness

Visibility indices are induced by sequential program order so that an implication $w_2 \Leftarrow_{l_2} e_2$ is *sequentially downstream* from another $w_1 \Leftarrow_{l_1} e_1$ in program order iff $l_1 \prec l_2$. These labels are generated in the circuit translation $\mathcal{C}(\mathrm{P})$ as the source program P is traversed recursively by the construction rules in Fig. 2.3 and 2.4. For instance, consider Fig. 2.3f defining the translation of a sequential composition $\mathcal{C}(\mathrm{P;Q})$: The indices attached to all (combinational) wire definitions in $\mathcal{C}(\mathrm{P})$ are $\prec$-ordered before all indices of wire definitions in $\mathcal{C}(\mathrm{Q})$. The index of the OR gates collecting the E_c', SEL, k_1 and k_2 wires are added with visibility index 0 that is strictly smaller than larger than any other index so that they can be traversed at least once in any evaluation without blocking. In contrast, for a parallel composition $\mathcal{C}(\mathrm{P} \,||\, \mathrm{Q})$ (Fig. 2.3h) all wire definitions in $\mathcal{C}(\mathrm{P})$ are made $\prec$-incomparable with all wire definitions in $\mathcal{C}(\mathrm{Q})$. The two OR gates creating the cross-coupled feedback for E_c, E_c' likewise are indexed incomparably with any other index, as are the synchronizer and collecting OR-gates for E_c', E_s', SEL, k_0, k_1 and k_2 on the output side. The tricky part is the loop which creates static cycles. However, we assume cyclic programs P^* are loop-safe, free of schizophrenia and each statement in P is either in the depth or in the surface[1]. Therefore, it is possible to arrange the flow indices in a loop $\mathcal{C}(\mathrm{P}^*)$ in such a way that all depth statements of P receive a flow index which is $\prec$-before the surface statements of P. This is achieved, e.g., if the flow indices are restarted with a minimal index with every pause in P and the start index for translating $\mathcal{C}(\mathrm{P}^*)$ is chosen to be greater than the largest termination index of any instantaneous depth path out of P.

The following statements and proofs are stated for a minor variation of the constructive evaluation rules of Fig. 4.1 where the side conditions in rules $PRES(\pi, l)$ and $ABS(\pi, w)$ are changed as $\pi \not\prec l$ instead of $\pi \not\preceq l$. This is a minor shift of position that helps keep the notation somewhat simpler. As done in the main text, we wish to identify gates G with visibility indices and name the output signal S of the gate as $S[G]$. Then, the side condition $\pi \not\prec l$ has the effect that an evaluation of signal $S[G]$ at index G behaves like an evaluation of $S[G]$ at the *input* side of the gate G. Since the wire definition for

[1] A statement is in the depth if the termination point of P can be reached from it instantaneously. A statement is in the surface if it is instantaneously reachable from the start point of P.

$S[G]$ drives the output side of the gate, it is sequentially downstream and cannot be seen in the evaluation of the inputs. Accordingly, the falsity of $G \not\preceq G$ blocks the wire definition from being used. However, this has the disadvantage that in order to evaluate $S[G]$ from the gate we need to pick an arbitrary index $m \not\preceq G$. Such always exists but is cumbersome to refer to, notationally. If we use the side condition $\pi \not\prec l$ instead then we can evaluate $S[G]$ at index G, since $G \not\prec G$, which is more convenient. This modification makes it theoretically possible that a gate reads its own output. This pathological case does not occur, however, in our BCC and SCC circuit structures. Hence, the two versions of the evaluation system can be considered equivalent.

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

Proof. For efficiency of notation let us drop the circuit evaluation context and write $e \hookrightarrow b$ and $e \hookrightarrow_\pi b$ instead of $BCC(P), I, R \vdash e \hookrightarrow b$ and $BCC(P), I, R \vdash e \hookrightarrow b$, respectively.

We exploit the fact that if an evaluation of a wire $w \hookrightarrow 1$ with $PRES(k)$ depends on the evaluation of another wire $z \hookrightarrow 1$ with $PRES(m)$, then m cannot be sequentially downstream from k, i.e., $k \not\prec m$. This is a result of the strictly sequential activation of execution via GO wires. Let us call this property of BCC circuits 1-*sequentiality*. Now consider an arbitrary evaluation tree for $e \hookrightarrow b$. Let $\pi \subset \mathcal{F}$ be an antichain such that for each rule application $PRES(k)$ occurring in this derivation tree we have $\pi \not\prec k$. We call π a 1-*covering* of the derivation $e \hookrightarrow b$. Observe that if for every wire implication $w \Leftarrow_k d$ in $BCC(P)$ the index k is visible from an antichain π, then π is a 1-covering. The statement of Prop. 1 is proven by induction on the structure of the derivation for arbitrary 1-coverings π.

Suppose π is a 1-covering for a derivation $e \hookrightarrow b$. Clearly, if the last rule applied is any one of the expression evaluations OP the induction hypothesis directly gives $e \hookrightarrow_\pi b$. The same holds for the input and state axioms IN and REG, which have no side conditions on π. The critical rules are $PRES(k)$ and $ABS(w)$.

Suppose $e = w \in \mathcal{S}$, $b = 1$ and $e \hookrightarrow b$ is the same as $w \hookrightarrow 1$, obtained by an application of $PRES(k)$ using a wire definition $w \Leftarrow_k d$ and a sub-evaluation $d \hookrightarrow 1$. By assumption, π is a 1-covering of and thus $\pi \not\prec k$. We claim that $\pi \oplus k$ is a 1-covering of the sub-derivation $d \hookrightarrow 1$. This follows by 1-sequentiality and the fact $\pi \oplus k \subseteq \pi \cup \{k\}$. For if there were a wire evaluation $z \hookrightarrow 1$ with $PRES(m)$ appearing in the tree for $d \hookrightarrow 1$ with $\pi \oplus k \prec m$, then either $\pi \prec m$ or $k \prec m$. The former is contradicting 1-coverage of π and the latter contradicts 1-sequentiality. Now, given that $\pi \oplus k$ is a 1-covering of the sub-derivation $d \hookrightarrow 1$ we can apply the induction hypothesis to obtain $d \hookrightarrow_{\pi \oplus k} 1$. From this an application of $PRES(\pi, k)$ with the wire definition $w \Leftarrow_k d$ is possible to derive $w \hookrightarrow_\pi 1$. This complete the induction step where the last rule is $PRES(k)$.

Finally, if the last rule for $e \hookrightarrow b$ is an application of $ABS(w)$ then $e = w \in \mathcal{S}$ and $b = 0$ and we have the immediate sub-derivations $d \hookrightarrow 0$ for all wire definitions $w \Leftarrow_k d$ in $BCC(P)$. We now simply prune all premises that violate the visibility constraint, i.e.,

for which $\pi \prec k$. The remaining premises then come from wire definitions $w \Leftarrow_k d$ with $\pi \not\prec k$. As above we argue that $\pi \oplus k$ is a 1-covering of the sub-derivation for $d \hookrightarrow 0$ and thus by induction hypothesis, $d \hookrightarrow_{\pi \oplus k} 0$. Hence, we can replace the application of $ABS(w)$ by an application of $ABS(\pi, w)$ with the pruned premises and obtain an evaluation for $w \hookrightarrow_\pi 0$ as desired. □

For flow-oriented circuits the visibility constraints are redundant and the constructive evaluation collapses to the standard semantics of Berry.

PROPOSITION 2. Let $\mathcal{C}$ be a flow-oriented circuit and $\pi \subset \mathcal{F}$ observation points such that $\pi \not\prec l$ for all wire definitions $w \Leftarrow_l d$ in $\mathcal{C}$. Then, $\mathcal{C}, I, R \vdash e \hookrightarrow b$ iff $\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b$.

Proof. Since the circuit evaluation context is fixed throughout we can leave it implicit and write $e \hookrightarrow b$ and $e \hookrightarrow_\pi b$ instead of $\mathcal{C}, I, R \vdash e \hookrightarrow b$ and $\mathcal{C}, I, R \vdash e \hookrightarrow_\pi b$, respectively. We say π *covers* a wire w if $\pi \not\prec k$ for every definition $w \Leftarrow_k d$ of wire w in $\mathcal{C}$. We say π is a *cover* of the derivation $e \hookrightarrow b$ or $e \hookrightarrow_\pi b$ if π covers all variables evaluated in the respective derivation tree. In particular, if $\pi \not\prec l$ for all wire definitions $w \Leftarrow_l d$ in $\mathcal{C}$, then π covers every possible derivation $e \hookrightarrow b$ and $e \hookrightarrow_\pi b$. We prove the Prop. 2 for arbitrary covers π.

Consider an evaluation tree for $e \hookrightarrow b$ with cover π. If e is a boolean expression we can directly apply the induction hypothesis on the sub-derivations, because the evaluation rules OP are all ignorant of the visibility index. Similarly, if $e = w \in \mathcal{I} \cup \mathcal{R}$ then we immediately get the equivalence $w \hookrightarrow_\pi b$ iff $w \hookrightarrow b$, for any π, as desired. So, in the sequel we only need to consider standard wire evaluations $z \hookrightarrow b$ and $z \hookrightarrow_\pi b$ for $z \in \mathcal{S}$. We argue by induction on the tree structure.

Let us first observe that if an evaluation tree $w \hookrightarrow b$ for a wire w contains the evaluation $z \hookrightarrow c$ for another wire z then $l_i \not\prec k_j$ for all wire definitions $w \Leftarrow_{l_i} e_i$ and $z \Leftarrow_{k_j} d_j$ in $\mathcal{C}$. This is because the evaluation tree witnesses that $z \twoheadrightarrow_{\mathcal{C}} w$ from which orientedness of $\mathcal{C}$ implies $l_i \not\prec k_j$. It follows that if π covers $w \hookrightarrow b$ then $\pi \oplus l_i$ also covers $w \hookrightarrow b$. If $\pi \oplus l_i$ does not cover $w \hookrightarrow b$, then this can only be if $w \hookrightarrow b$ contains an application of $PRES(k)$ with $\pi \oplus l_i \prec k$. Again, $\pi \oplus l_i \subseteq \pi \cup \{l_i\}$ implies $\pi \prec k$ or $l_i \prec k$. Since we cannot have $\pi \prec k$ by the coverage assumption, $l_i \prec k$. The $PRES(k)$ rule application must be associated with a wire implication $z \Leftarrow_k d$. But now by orientedness we infer $l_i \not\prec k$, a contradiction. The very same argument can be used to show that if π covers $w \hookrightarrow_\pi b$ then $\pi \oplus l_i$ also covers $w \hookrightarrow_{\pi \oplus l_i} b$.

The translation of trees in the induction step is trivial if we have $w \hookrightarrow 0$ and the last rule is an application of a $ABS(w)$. In this case the premises are derivations of $e_i \hookrightarrow 0$ for all implications $w \Leftarrow_{l_i} e_i$ associated with wire w. As argued above $\pi \oplus l_i$ covers $e_i \hookrightarrow 0$. By induction hypothesis this gives $e_i \hookrightarrow_{\pi \oplus l_i} 0$ and thus $w \hookrightarrow_\pi 0$ with rule $ABS(\pi, w)$. Secondly, the translation is trivial in the other direction, if the derivation is $w \hookrightarrow_\pi 1$ and the last rule is $PRES(\pi, l)$, because we can transform directly: Then the rule has a single premise proving $e_i \hookrightarrow_{\pi \oplus l_i} 1$ for some selected wire equation $w \Leftarrow_{l_i} e_i$ such that $\pi \not\prec l_i$. As before, by orientedness, $\pi \oplus l_i$ must cover the derivation $e_i \hookrightarrow_{\pi \oplus l_i} 1$.

The induction hypothesis then generates a constructive evaluation $e_i \hookrightarrow 1$ from which we get $w \hookrightarrow 1$ by rule $PRES(l_i)$, completely ignoring the visibility information.

The more interesting cases are the addition of visibility to a derivation $w \hookrightarrow 1$ and the removal of visibility in a derivation $w \hookrightarrow_\pi 0$. The reason is that in an application of $PRES(l)$ which ignores visibility the selected premise may be actually be down-stream and violate $\pi \not\prec l$. So, $PRES(l)$ cannot be replaced by $PRES(\pi, l)$. Dually, in an application of $ABS(w)$ the visibility constraint might ignore a wire definition $w \Leftarrow_l d$ that is down-stream ($\pi \prec l$) and that does not evaluate to 0. So, $ABS(\pi, w)$ cannot be replaced by $ABS(w)$. The point is, however, that because of the covering property of π this cannot happen.

Consider a tree for $w \hookrightarrow 1$ finishing in an application of $PRES(l_i)$ with a single premise $e_i \hookrightarrow 1$ for *one of* the wire definitions $w \Leftarrow_{l_i} e_i$ in $\mathcal{C}$ for w. But by assumption, π covers the derivation $w \hookrightarrow 1$, whence $\pi \not\prec l_i$. Recalling as above that $\pi \oplus l_i$ covers $w \hookrightarrow 1$ and thus also the sub-derivation $e_i \hookrightarrow 1$ we can invoke the induction hypothesis to obtain a derivation $e_i \hookrightarrow_{\pi \oplus l_i} 1$ and from here, qua rule $PRES(\pi, l_i)$, a derivation $w \hookrightarrow_\pi 1$.

The same argument works in the other direction starting from an evaluation tree for $w \hookrightarrow_\pi 0$ ending in an application of the $ABS(\pi, w)$ rule. The premises are of the form $e_i \hookrightarrow_{\pi \oplus l_i} 0$ for all definitions $w \Leftarrow_{l_i} e_i$ of wire w satisfying the visibility constraint $\pi \not\prec l_i$. Now, since π covers w, we get $\pi \not\prec l_i$ for all $0 \leq i \leq n$. Hence the premises of $ABS(\pi, w)$ include derivations $e_i \hookrightarrow_{\pi \oplus l_i} 0$ for all wire definitions of w. By induction these translate into derivations $e_i \hookrightarrow 0$ from which an application of $ABS(w)$ finally yields $w \hookrightarrow 0$. □

Since *SCC* circuits are flow-oriented, by Prop. 2 it does not matter if we evaluate $SCC(P)$ under visibility constraints or not.

Proposition 3. *Let $SCC(P)$ be the SCC circuit translation of a program P. Then, $SCC(P)$ is flow-oriented. Further, let $\pi \subset \mathcal{F}$ be observation points from which every wire implication $w \Leftarrow_l e$ in $SCC(P)$ is visible, i.e., such that $\pi \not\prec l$. Then, $SCC(P), I, R \vdash e \hookrightarrow b$ iff $SCC(P), I, R \vdash w \hookrightarrow_\pi b$.*

Proof. Flow-orientation of $SCC(P)$ is a direct consequence of the definition of the visibility indices generated with the recursive translation of P following the rules in Fig. 2.3 and 2.4, as described at the beginning of this section. The second part of Prop. 3 is a consequence of flow-orientation and Prop. 2. □

The final theorem Thm. 1 not only implies conservativeness of *SCC* over *BCC*. It also shows how the standard constructive semantics of the new *SCC* circuits can be obtained from the existing *BCC* circuits by the new visibility-based constructive evaluation relation defined in Fig. 4.1. The following theorem states only one direction of this equivalence. The proof for the converse direction, which we claim is true as well, will be added in future version of this text.

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

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

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

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

where index 0 is chosen so that it is $\preceq$-incomparable to all indices appearing in either $BCC(P)$ or $SCC(P)$.

Sketch. Recall that the wiring of $BCC(P)$ is the same as for $SCC(P)$ except for the treatment of the signal buses. This means the evaluation of instances of control wires

$$\mathsf{CTR} = \{\mathsf{GO}, \mathsf{RES}, \mathsf{SUS}, \mathsf{KILL}, \mathsf{SEL}, \mathsf{k}_0, \mathsf{k}_1, \mathsf{k}_{2+}\}$$

in $BCC(P)$ and in $SCC(P)$ exactly match up each other to the point where either circuit reads a signal $s \in Sig$. In $BCC(P)$ the signals are read from the global input bus $\mathsf{E}.s$ and written to global output bus $\mathsf{E'}.s$. In $SCC(P)$ signals are passed around on two types of local buses, in sequential direction read from $\mathsf{E_s}.s$ and written to $\mathsf{E'_s}.s$, in concurrent direction read from $\mathsf{E_c}.s$ and written to $\mathsf{E'_c}.s$. However, these buses are instantiated recursively for each sub-program of P at different visibility indices. So, the ground instances of these "signal" wires read $\mathsf{E}.s$, $\mathsf{E'}.s$, $\mathsf{E_s}[l].s$, $\mathsf{E'_s}[l].s$ $\mathsf{E_c}[l].s$ and $\mathsf{E'_c}[l].s$ where $l \in \mathcal{F}$ and $s \in Sig$.

Let $BCC(P)^*$ abbreviate the context $BCC(P), E \Leftarrow_{\mathrm{o}} I \vee E_c \vee E', E'_c \Leftarrow_{\mathrm{o}} E', I, R$. Similarly, $SCC(P)^*$ stands for

$$SCC(P), \mathsf{E_s} \Leftarrow_0 I \wedge GO[\mathrm{o}], \mathsf{E'_c} \Leftarrow_{\mathrm{o}} \mathsf{E'_c}[\mathrm{o}], I, R.$$

For technical convenience, we assume that $0 \in \pi$, so that $\pi \oplus 0 = \pi$. This is without generality because 0 is globally $\prec$-incomparable. Hence, it does not impose any visibility constraint and acts neutrally: $e \hookrightarrow_\pi b$ iff $e \hookrightarrow_{\pi \cup \{0\}} b$. Our argument proceeds by induction on the size of derivations together with following auxiliary correlations:

(a) $BCC(P)^* \vdash \mathsf{GO}[l] \hookrightarrow_\pi b \Leftrightarrow SCC(P)^* \vdash \mathsf{GO}[l] \hookrightarrow_\pi b$.

(b) $BCC(P)^* \vdash \mathsf{E'_c}.s \hookrightarrow_\pi b \Rightarrow SCC(P)^* \vdash E'_c[l].s \hookrightarrow_\pi b$ for some $l \in \mathcal{F}$ with $\pi \not\asymp l$.

(c) $SCC(P)^* \vdash E_c[l].s \hookrightarrow_\pi b \Rightarrow SCC(P)^* \vdash E_c[m].s \hookrightarrow_\pi b$ for all $l \preceq m$ and $\pi \not\asymp m$.

(d) $SCC(P)^* \vdash E_c[l].s \hookrightarrow_\pi 1$ iff for all visible "concurrent" m, i.e., such that $\pi \not\asymp m$ and both $l \not\asymp m$ and $m \not\asymp l$ we have $SCC(P)^* \vdash E'_c[m].s \hookrightarrow_\pi 1$.

(e) $SCC(P)^* \vdash E_s[l].s \hookrightarrow_\pi 1$ iff $SCC(P)^* \vdash E_s.s \hookrightarrow_\pi 1$ or there exists some visible upstream index m, i.e., with $\pi \not\asymp m$ and $m \preceq l$ such that $SCC(P)^* \vdash E'_s[m].s \hookrightarrow_\pi 1$.

(f) $GO \hookrightarrow 1$ evaluations strictly dominate all activations of downstream statements.

Obviously, the ($\Rightarrow$) direction of Thm. 1 is a consequence of (b). We only prove the most important statements (a) and (b) here. The proofs for (c)–(f), which are properties purely of the SCC constructions are omitted.

• (b) Suppose $BCC(P)^* \vdash \mathsf{E}'_{\mathsf{c}}.s \hookrightarrow_\pi b$. Then, $0 \preceq \pi \not\prec 0$ and $BCC(P)^* \vdash \mathsf{E}'.s \hookrightarrow_\pi b$ by rule $ABS(\pi, \mathsf{E}'.s)$. Then, there must exist a visible wire definition $E'.s \Leftarrow_l GO[l]$ in $BCC(P)$ with $\pi \not\prec l$ and $BCC(P)^* \vdash GO[l] \hookrightarrow_{\pi\oplus l} b$. By construction, the $SCC(P)^*$ circuit contains the very same wire definition in the form $E'_c[l].s \Leftarrow_l GO[l]$ with the same index l. By induction hypothesis (a) on the height of the $BCC(P)^*$ derivation we conclude $SCC(P)^* \vdash GO[l] \hookrightarrow_{\pi\oplus l} b$ and so $SCC(P)^* \vdash E'_c[l].s \hookrightarrow_\pi b$ as desired.

• (a) Let $BCC(P)^* \vdash GO[l] \hookrightarrow_\pi b$ for some control wire $GO[l]$ in $BCC(P)^*$. These GO wires are connected in both circuit semantics in the same way except for conditionals, where $SCC(P)$ feeds $GO[l]$ from both the concurrent and sequential rails $E_s[l].s$, $E_c[l].s$, respectively, of a signal $s \in Sig$, whereas $BCC(P)^*$ only uses the single input $E.s$. Hence, it is enough to consider those GO wires which drive the branches P, Q of a conditional **present** s **then** P **else** Q.

Without loss of generality, let us assume the $GO[l]$ at hand is the input of a positive conditional branch P, say given through a wire definition

$$GO[l] \Leftarrow_l GO[k] \wedge E.s \in BCC(P) \tag{4.8}$$

where $GO[k]$ is the upstream go with $k \preceq l$ that starts the present test. Then, $\pi \not\prec l$ and $BCC(P)^* \vdash GO[k] \wedge E.s \hookrightarrow_{\pi\oplus l} b$. The $SCC(P)^*$ circuit instead has the corresponding wire definition

$$GO[l] \Leftarrow_l GO[k] \wedge (E_c[l].s \vee E_s[l].s) \in SCC(P) \tag{4.9}$$

The case where of an activation of the negative branch of a present test is treated symmetrically. We make a case analysis on b. Note that always $l \not\prec l$ and therefore $\pi \not\prec l$ implies $\pi \oplus l \not\prec l$.

If $b = 1$ we must have $BCC(P)^* \vdash GO[k] \hookrightarrow_{\pi\oplus l} 1$ and $BCC(P)^* \vdash E.s \hookrightarrow_{\pi\oplus l} 1$. By induction hypothesis (a), the former evaluation directly carries over and we obtain $SCC(P)^* \vdash GO[k] \hookrightarrow_{\pi\oplus l} 1$. The latter, because of the feedback wiring $E \Leftarrow_{\mathrm{o}} I \vee E_c \vee E'$, may either arise from one of the following:

(i) $BCC(P)^* \vdash I.s \hookrightarrow_{\pi\oplus l} 1$,

(ii) $BCC(P)^* \vdash E_c.s \hookrightarrow_{\pi\oplus l} 1$,

(iii) $BCC(P)^* \vdash E'.s \hookrightarrow_{\pi\oplus l} 1$.

In the first case (i) we get $SCC(P)^* \vdash I.s \hookrightarrow_{\pi\oplus l} 1$. By the properties of 0 we have $\pi \oplus l \oplus 0 = \pi \oplus l$, whence $SCC(P)^* \vdash I.s \hookrightarrow_{\pi\oplus l\oplus 0} 1$. Then, since $\pi \oplus l \not\prec 0$ the external wiring $\mathsf{E_s} \Leftarrow_0 I$ implies $SCC(P)^* \vdash \mathsf{E_s}.s \hookrightarrow_{\pi\oplus l} 1$ by rule $PRES(\pi, l)$. Now we exploit the auxiliary fact (e) to obtain $SCC(P)^* \vdash \mathsf{E_s}[l].s \hookrightarrow_{\pi\oplus l} 1$. From this and (4.9) it follows that $SCC(P)^* \vdash GO[l] \hookrightarrow_\pi 1$.

In the second case (ii) we are done since this means $SCC(P)^* \vdash E_c.s \hookrightarrow_{\pi\oplus l} 1$ and thus by (c), $SCC(P)^* \vdash E_c[l].s \hookrightarrow_{\pi\oplus l} 1$. Therefore, $SCC(P)^* \vdash E_c[l].s \vee E_s[l].s \hookrightarrow_{\pi\oplus l} 1$, which gives $SCC(P)^* \vdash GO[l] \hookrightarrow_{\pi} 1$. This deals with the situation where the signal s is evaluated from the external environment.

What if we have (iii) where $BCC(P)^*$ receives the signal value from the internal circuit feedback, i.e., $BCC(P)^* \vdash E'.s \hookrightarrow_{\pi\oplus l} 1$? This evaluation must be generated from an **emit** s circuitry with wire definition $E'.s \Leftarrow_m GO[m]$ in $BCC(P)^*$, so that $\pi \oplus l \not\prec m$ and $BCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l\oplus m} 1$. This implies that $\pi \not\prec m$, i.e., the **emit** is visible for π. Regarding the relative visibility of l and m we claim that $l \not\prec m$: This is obvious if $l \not\prec \pi$ since then $\pi \oplus l = \pi \cup \{l\}$. What if $l \preceq \pi$? Our circuits are constructed so that the activation of a program statement dominates the activation of all down-stream statements, which is our auxiliary fact (f). Specifically, here $GO[l]$ is the activation control of the then branch of the present test **present** s **then** P **else** Q at index l and $GO[m]$ the activation control of the **emit** at index m. If $l \prec m$ was true, i.e., the emission downstream from the present test, the evaluation tree for $BCC(P)^* \vdash GO[k] \hookrightarrow_{\pi\oplus l\oplus m} 1$ would not be able to contain the sub-evaluation $BCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l} 1$. Hence index m can only be either concurrent to l, i.e., $l \not\prec m$ and $m \not\prec l$ or upstream from it, i.e., $m \preceq l$.

Next, consider the matching wire definitions for the **emit**, $E'_s[m].s \Leftarrow_m GO[m]$ and $E'_c[m].s \Leftarrow_m GO[m]$, in the circuit $SCC(P)^*$. By induction hypothesis (a), the activation of the **emit** carries over, whence $SCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l\oplus m} 1$. This means we have both $SCC(P)^* \vdash E'_c[m].s \hookrightarrow_{\pi\oplus l} 1$ and $SCC(P)^* \vdash E'_s[m].s \hookrightarrow_{\pi\oplus l} 1$. Now, if m is concurrent to l then by (d) the former propagates as $SCC(P)^* \vdash E_c[l].s \hookrightarrow_{\pi\oplus l} 1$ through the internal feedback wiring. If m is upstream, $m \preceq l$, the latter propagates as $SCC(P)^* \vdash E_s[l].s \hookrightarrow_{\pi\oplus l} 1$ by sequential forwarding (e). Again, in each case, this implies $SCC(P)^* \vdash E_c[l].s \vee E_s[l].s \hookrightarrow_{\pi\oplus l} 1$ and thus $SCC(P)^* \vdash GO[l] \hookrightarrow_{\pi} 1$ as desired.

Now we come to treat the case $b = 0$, which implies $BCC(P)^* \vdash GO[k] \hookrightarrow_{\pi\oplus l} 0$ or $BCC(P)^* \vdash E.s \hookrightarrow_{\pi\oplus l} 0$ considering (4.8). If the former holds we invoke the induction hypothesis (a) inducing $SCC(P)^* \vdash GO[k] \hookrightarrow_{\pi\oplus l} 0$ and thus by (4.9) we have $SCC(P)^* \vdash GO[l] \hookrightarrow_{\pi} 0$. If the latter holds, $BCC(P)^* \vdash E.s \hookrightarrow_{\pi\oplus l} 0$, then with the outside marshalling $E \Leftarrow_0 I \vee E_c \vee E'$, we must have

$$BCC(P)^* \vdash I \hookrightarrow_{\pi\oplus l} 0 \tag{4.10}$$

$$BCC(P)^* \vdash E_c.s \hookrightarrow_{\pi\oplus l} 0 \tag{4.11}$$

$$BCC(P)^* \vdash E'.s \hookrightarrow_{\pi\oplus l} 0. \tag{4.12}$$

We claim that then both

$$SCC(P)^* \vdash E_c[l].s \hookrightarrow_{\pi\oplus l} 0 \text{ and } SCC(P)^* \vdash E_s[l].s \hookrightarrow_{\pi\oplus l} 0.$$

This, too, implies $SCC(P)^* \vdash GO[l] \hookrightarrow_{\pi} 0$ by (4.9). Intuitively, in $SCC(P)^*$ both the sequential and concurrent signal wires $E_c[l].s$ and $E_s[l].s$ must be switched off because the global input (4.10) and all emitters **emit** s in $BCC(P)^*$, which are visible for index $\pi \oplus l$, are switched off.

Let us argue this is more detail. Consider that $E_c[l].s$ in $SCC(P)^*$ is a tree of disjunctions fed by the global (concurrent) input $E_c.s$ and wire definitions $E_c[m].s \Leftarrow_m GO[m]$ for some (innermost) wires $E_c[m].s$ where m is concurrent with l. This is the content of auxiliary fact (d). All of these wire definitions $E_c[m].s \Leftarrow_m GO[m]$ are associated with an emitting wire $E'.s \Leftarrow_m GO[m]$ in $BCC(P)^*$. But (4.12) then implies $BCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l}$ o for all $\pi \not\prec m$. By induction hypothesis this also means these activation wires are off in $SCC(P)^*$, i.e., $SCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l}$ o for all $\pi \not\prec m$. This shows that the concurrent wire $E_c[l].s$ must be off in $SCC(P)^*$ by (d), i.e., $SCC(P)^* \vdash E_c[l].s \hookrightarrow_{\pi\oplus l}$ o.

Each sequential signal wire $E_s[l].s$ in $SCC(P)^*$ ultimately depends on possibly the global input I, or a set of upstream emitter definitions $E'_s[m].s \Leftarrow_m GO[m]$, with $m \preceq l$, in such a way that if all of them evaluate to 0, wire $E_s[l].s$ must necessarily evaluate to 0 as well. This is the content of fact (e). The global input evaluates to 0 by (4.10). Further, since each sequential emission $\mathsf{E}'_\mathsf{s}[m].s \Leftarrow_m GO[m]$ in $SCC(P)^*$ is associated with a wire definition $E'.s \Leftarrow_m GO[m]$ in $BCC(P)^*$, by (4.12) we must have

$$BCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l\oplus m} \mathrm{o} \tag{4.13}$$

provided $\pi \oplus l \not\prec m$. (Otherwise the evaluation to 0 at index $\pi \oplus l$ would be down to the fact that m is downstream and thus not visible.) But since $\pi \not\prec l$ and $m \preceq l$ it follows that $\pi \not\prec m$. This implies $\pi \oplus l \not\prec m$ as one easily shows from the definition of the $\oplus$ operator. Hence we know (4.13) is true and apply the induction hypothesis to get $SCC(P)^* \vdash GO[m] \hookrightarrow_{\pi\oplus l\oplus m}$ o from which readily $SCC(P)^* \vdash \mathsf{E}'_\mathsf{s}[m].s \hookrightarrow_{\pi\oplus l} 0$ taking into account that there is exactly one wire definition for a sequential emission $\mathsf{E}'_\mathsf{s}[m].s$. Since this is established for all upstream sequential emitters we consequently get $SCC(P)^* \vdash E_s[l].s \hookrightarrow_{\pi\oplus l}$ o by (e) and from this, finally using (4.9), we get the desired result: $SCC(P)^* \vdash GO[l] \hookrightarrow_\pi$ o. □

5 SCC vs. the SC MoC

The desire to handle sequential updates in a synchronous setting 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 [18]. 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.

However, 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 is not

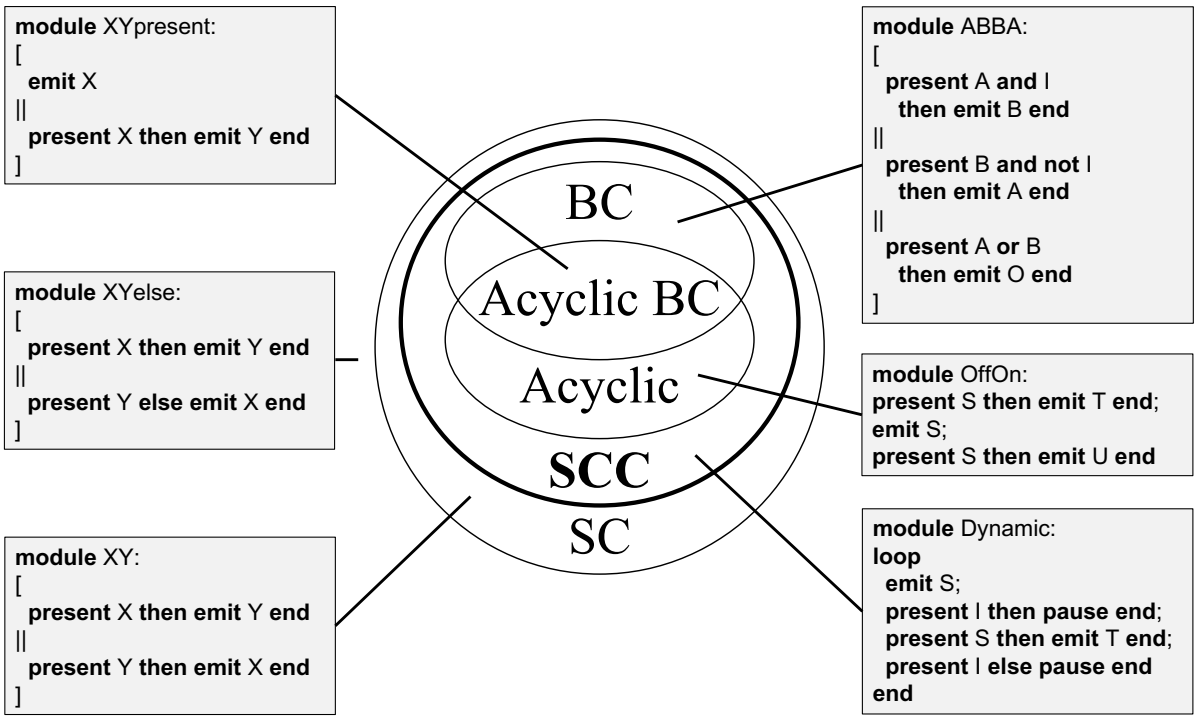


Figure 5.1: The class of programs considered valid under the Sequentially Constructive Circuit (SCC) semantics, proposed here, in relation to other program classes. "Acyclic" refers to programs that do not have static cycles involving concurrent data dependencies (*structurally iur-acyclic* in [18]). Most Esterel compilers handle only Acyclic BC [30]. Esterel v5 handles all of BC [36]. The SCEst2SCL compiler handles Acyclic [31]. Our work extends compilation to SCC, that is SC without "speculation."

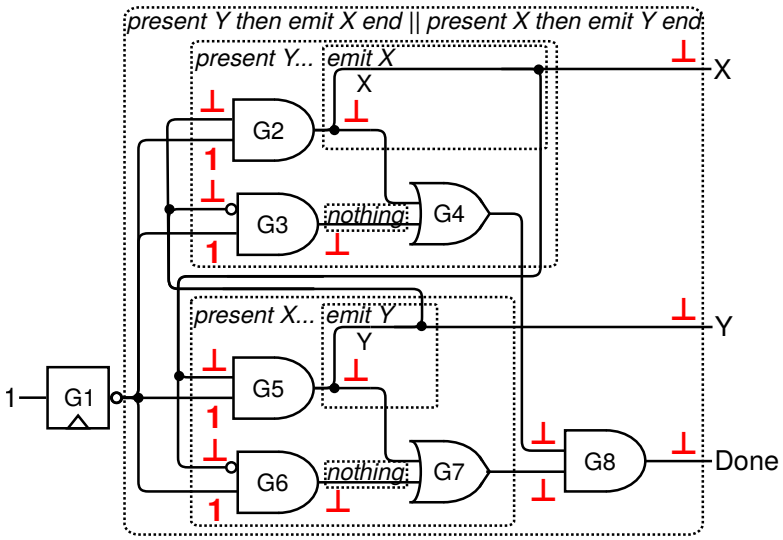


Figure 5.2: Circuit for XY, which is not constructive

founded on constructive logic. As noted in the original SC proposal [18], it accepts programs that in the traditional synchronous sense are considered "speculative." Consider the minimalistic XY Esterel example in the lower-left of Fig. 5.1. Its output 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. 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. 5.2 that has been constructed according to Berry's circuit semantics for Esterel [5] and where 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 constructive logic [36, 24]. For the XY circuit, this means that we cannot guarantee unique stabilization, and indeed there are two possible stable states for this circuit, one with both X and Y considered absent and one with both considered present. Our SCC proposal rules out such cases and **defines a notion of SC that has a firm physical grounding**.

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 [17] or SC Esterel (SCEst) [31, 17], 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 XYpresent shown on the top-left in Fig. 5.1; the emission of Y depends on X, but not the other way around. Another example, not BC but still acyclic, is OffOn shown on the right in Fig. 5.1, where S is absent when it is tested the first time and present the second time around. 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 [10] 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 [3]. 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 [23, 25, 32, 11, 26]. In ABBA, seen in the top-right of Fig. 5.1, 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, and most existing compilers for synchronous programs will reject this; 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 [5]. 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 [32]. 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 [29]. 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. The SCC2BC transformation **provides 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. 5.1. This is not Acyclic, because (as explained in Sec. 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 OffOn) 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. 5.1; there, Y is emitted iff X is present, X is emitted iff Y is absent, thus there is no consistent signal evaluation.

6 Related Work

There is a large body of work on SSA transformations. However, relatively little appears to have been developed for concurrent programs. Lee et al. [21] present one approach, extended by Novillo et al. [27] to handle mutual exclusion. However, they do not address programs divided into discrete ticks with value re-initialization at tick boundaries. Kalla et al. [19] use SSA to translate C code into synchronous data-flow equations but without concurrency.

The initially proposed semantics for Esterel [4, 7] did already provide synchrony and determinacy, but were rather restrictive as to which programs were considered "causal." This has been subsequently resolved with the constructive semantics, for which Berry has proposed several alternative, but equivalent formalizations [5]. (1) The *constructive behavioral semantics* is a non-speculative refinement of the logical behavioral semantics. It is in a way the simplest and most abstract formalization of what an Esterel program means, but not a suitable basis for a compiler. (2) The *constructive operational semantics* is a micro-step semantics, an earlier version of this has been used in the Esterel v4 compiler. (3) The *constructive circuit semantics*, which we refer to as BCC here, is the basis of the Esterel v5 compiler.

The semantics introduced here for Esterel/SCEst deviates from the constructive semantics of Berry in that sequential compositions $R_1; W; R_2$ are executed like ordinary imperative programs and signal emissions behave like assignments to boolean variables. Specifically, variables read by R_1 may be overwritten by assignments in W so they have a different value when read by R_2. Under the constructive semantics of Esterel [5] there cannot be an emission to a signal in W after it has been read in R_1. The reason is that in the circuit semantics of Esterel the writer W is wired not only to sequentially down-stream R_2 but also to upstream R_1 which creates a causality loop: The reading R_1 depends on the writing by W, yet the write cannot start until the read R_1 has terminated. By removing such back-flow dependencies against the sequential program order our semantics eliminates such causality problems. As noted earlier, this is inspired by the SC proposal [18]. Rathlev et al. [31] present an SC-based semantics for (kernel) Esterel that also covers valued signals while we only treat boolean signals here. However, the core of our results is to show how the SC semantics of boolean Esterel can be reduced to the constructive semantics of Esterel by SSA transformation. This is an important advance over [18], in that the existing results on Esterel [24] now imply that the SSA transformed program is delay-insensitive considered as a boolean control circuit. We believe this is a strong guarantee on the robustness of the generated code, even if the implementation is not a circuit but single-threaded imperative code facing scheduling uncertainties arising from weak memory models [28].

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 consist 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. [12] on N-synchronous Kahn networks. There, the shared communication structure are data-flow variables. The relaxation is to decouple the writing and reading of a variable by a bounded number of ticks, which makes it possible to program multi-rate data-streaming very conveniently. Analogous to what we do here with the SSA transformation of signals, the semantics of the N-synchronous programming model is defined by an "expansion transformation" into the traditional 0-synchronous model [16]. The expansion of variables in this case are buffers, automatically synthesised by static type-checking using a clock calculus with causality sub-typing.

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. [30]. One early approach has been the automata-based compilation, where an Esterel program is translated into a Mealy machine. The Esterel v2 compiler was based on Brzozwski's residual technique to translate regular expressions into automata [8]. Transitions were derived directly from Esterel's behavioral semantics (then still in its non-constructive variant), given as Plotkin-style Structural Operational Semantics (SOS) rules [6]. States then correspond to the program derivatives. This tends to produce fast code, as basically the programs are already partially evaluated as much as possible at compile time. However, since concurrency is compiled away into product automata, the state space and the resulting code may become unacceptably large. The aforementioned circuit-based compilation, where the synthesized code simulates a netlist, does not have that size explosion problem, since the resulting code size scales basically linearly with the original Esterel program [30]. 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 [15]. A good overview of this and other approaches for compiling concurrent programs (not necessarily Esterel) is presented by Edwards [14]. As our SCC2BC transformation is a source-to-source transformation that results in standard Esterel code, 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. 5.1. This, however, is an orthogonal issue to the work presented here.

As already pointed out, 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 [26]. Lukoschus et al. present an approach to remove cycles at the Esterel level [22]. Their technique, which maps BC programs to statically acyclic BC programs, could be combined with ours that maps SCC to BC into a transformation that maps SCC to acyclic BC. The role of cyclic circuits in hardware synthesis has been discussed in [23, 32] and their analysis in ternary algebra in [23, 9, 36, 25]. Schneider et al. have suggested the use of scheduling or atomicity constraints for increasing constructiveness of cyclic circuits [33, 34]. The idea of flow indices to express evaluation order in ternary analysis and connecting them with SSA transformation, 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 (dominator analysis) 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 structure for functional programs has recently been proposed by Kuper et. al. [20]. They introduce lattice-based data structures, called LVars, in which all write accesses produce a monotonic value increase in the lattice and all read accesses are blocked until the memory value has passed a read-specific threshold. Each variable's domain is organised as a lattice of states with $\bot$ and $\top$ representing an empty new location and an error, respectively. A write operation of the form $\mathsf{put}\ \mathsf{lv}\ v$ computes the least upper bound (join) of the current state of lv and the value v. The read operation $\mathsf{get}\ \mathsf{lv}\ \theta$ blocks until the state of lv reaches a value in the threshold set θ, and from then on any execution of $\mathsf{get}\ \mathsf{lv}\ \theta$ will return the same value independently of any interleaved execution of a put. Because of monotonicity all writes are confluent with each other. Since reads are blocked each LVar data type can thus be viewed as a class of signals with a threshold-determined protocol.

As already mentioned in the introduction and in Sec. 3.4, several approaches have been developed to handle schizophrenia. A simple method for Esterel is to duplicate all loop bodies, which may lead to exponential code/circuit size increase [5]. Tardieu et al. [37] have improved this to at worst quadratic increase. For signal reincarnation, SCC2BC is an efficient alternative, as discussed in Sec. 3.4. Beyond the realm of Esterel-style synchronous signals, Aguado et al. [1] demonstrated that sequential variables can elegantly circumvent the reincarnation issue by making signal initialization explicit (rather than implicit as in Esterel) and separating surface and depth initializations.

Not considered here are programs that are SC but contain instantaneous loops. These are forbidden in SCC and the program classes contained in it, but can be handled by the SCEst2SCL compiler based on priorities [17].

7 Conclusion and Future Work

In this report, we have explored how to extend the concept of static single assignment to reactive, synchronous programming. Specifically, we presented the SCC2BC source-to-source tranformation procedure for Esterel, which can be used as a pre-processing step for standard Esterel compilers. For the Esterel programmer, SCC2BC allows the convenience of sequential, imperative-style programming familiar from languages such as C or Java, without leaving the solid foundation of determinate concurrency. Furthermore, as illustrated in the introduction with the SignalReinc example (Lst. 1.4), SCC2BC handles signal reincarnation naturally, without the code duplication inherent in earlier proposals [5, 37]. SCC2BC is a minimally invasive source-to-source transformation that splits up signals into different versions only when needed to eliminate signal dependencies that go against sequential control flow. To confirm the correctness and completeness of our approach, we have implemented SCC2BC and validated it with a range of both BC and non-BC, cyclic and acyclic programs. Modulo optimization, SCC2BC typically leaves BC programs untouched, as desired. This includes non-trivial, statically cyclic cases such as the token ring arbiter. Programs that are not BC but SCC are transformed into equivalent BC programs.

This work also 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. 5.1. SCC programs can either 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, using the SCC2BC transformation presented in Sec. 3.

There are numerous directions to procede 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. 2.2), where SCC has two fewer gates than BCC.

Conservativeness (Sec. 4) 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. 4.

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, 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. This can be implemented for example as a concurrently running **signal** helper **in present** error **and** helper **then emit** helper **end end**; if error is proven to be always absent, we can also prove that helper is absent and everything is well-defined, otherwise we can neither prove helper to be present nor to be absent and the whole program must be rejected, at compile time. 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 [17]. 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.

Bibliography

[1] Joaquín Aguado et al. "Grounding Synchronous Deterministic Concurrency in Sequential Programming". In: *Proceedings of the 23rd European Symposium on Programming (ESOP '14), LNCS 8410*. Grenoble, France: Springer, Apr. 2014, pp. 229–248.

[2] Felice Balarin et al. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, Apr. 1997.

[3] Albert Benveniste et al. "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.

[4] Gérard Berry. "Preemption in Concurrent Systems". In: *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1993, pp. 72–93. ISBN: 3-540-57529-4.

[5] Gérard 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.

[6] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.

[7] Gérard Berry and Georges Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152.

[8] Janusz A. Brzozowski. "Derivatives of regular expressions". In: *Journal of the ACM* 11.4 (Oct. 1964), pp. 481–494.

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

[10] P. Caspi et al. "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.

[11] Koen Claessen. "Safety Property Verification of Cyclic Synchronous Circuits". In: *Electronic Notes in Theoretical Computer Science*. Vol. 88. Elsevier, July 2003, pp. 55–69.

[12] A. Cohen et al. “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*. POPL ’06. New York, NY, USA: ACM, 2006, pp. 180–193.

[13] Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490.

[14] Stephen A. Edwards. “Tutorial: Compiling concurrent languages for sequential processors”. In: *ACM Transactions on Design Automation of Electronic Systems* 8.2 (Apr. 2003), pp. 141–187.

[15] Stephen A. Edwards and Jia Zeng. “Code Generation in the Columbia Esterel Compiler”. In: *EURASIP Journal on Embedded Systems* Article ID 52651, 31 pages (2007).

[16] Nicolas Halbwachs et al. “The synchronous data-flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[17] Reinhard von Hanxleden et al. “SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014.

[18] Reinhard von Hanxleden et al. “Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation”. In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.

[19] Hamoudi Kalla et al. “Automated translation of C/C++ models into a synchronous formalism”. In: *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS’06)*. Mar. 2006, 9 pp.-436.

[20] Lindsey Kuper et al. “Freeze after writing: Quasi-deterministic parallel programming with LVars”. In: *Principles of Programming Languages (POPL ’14)*. Dan Diego, USA: ACM, 2014, pp. 257–270.

[21] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. “Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs”. In: *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*. LCPC ’97. Springer-Verlag, 1998, pp. 114–130.

[22] Jan Lukoschus and Reinhard von Hanxleden. “Removing Cycles in Esterel Programs”. In: *International Workshop on Synchronous Languages, Applications and Programming (SLAP ’05)*. Edinburgh, Apr. 2005.

[23] Sharad Malik. “Analysis of Cyclic Combinational Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.7 (July 1994), pp. 950–956.

[24] Michael Mendler, Thomas R. Shiple, and Gérard Berry. "Constructive Boolean circuits and the exactness of timed ternary simulation." In: *Formal Methods in System Design* 40.3 (2012), pp. 283–329.

[25] Kedar S. Namjoshi and Robert P. Kurshan. "Efficient Analysis of Cyclic Definitions". In: *Proceedings of the 11th International Conference on Computer Aided Verification*. Vol. 1633. LNCS. Springer, 1999, pp. 394–405.

[26] Osama Neiroukh, Stephen A. Edwards, and Xiaoyu Song. "Transforming Cyclic Circuits Into Acyclic Equivalents". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 27.10 (2008), pp. 1775–1787.

[27] Diego Novillo, Ronald C. Unrau, and Jonathan Schaeffer. "Concurrent SSA Form in the Presence of Mutual Exclusion". In: *Proc. 1998 International Conference on Parallel Processing (ICPP'98)*. Minneapolis, MN, USA, Aug. 1998, pp. 356–365.

[28] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: X86-TSO". In: *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 391–407. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_27.

[29] Paritosh Pandya. "The Saga of Synchronous Bus Arbiter: On Model Checking Quantitative Timing Properties of Synchronous Programs". In: *Electronic Notes in Theoretical Computer Science*. Ed. by Florence Maraninchi, Alain Girault, and Éric Rutten. Vol. 65. Elsevier, 2002.

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

[31] Karsten Rathlev et al. "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, Sept. 2015.

[32] Marc D. Riedel and Jehoshua Bruck. "The Synthesis of Cyclic Combinational Circuits". In: *Proceedings of the conference on Design automation (DAC '03)*. Anaheim, California, USA, June 2003.

[33] Klaus Schneider et al. "Improving Constructiveness in Code Generators". In: *Int'l Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*. Ed. by Florence Maraninchi, Marc Pouzet, and Valérie Roy. Edinburgh, Scotland, UK: ENTCS, Apr. 2005, pp. 1–19.

[34] Klaus Schneider et al. "Maximal Causality Analysis". In: *Conference on Application of Concurrency to System Design (ACSD'05)*. St. Malo, France, June 2005, pp. 106–115.

[35] K. Schneider and M. Wenz. "A new method for compiling schizophrenic synchronous programs". In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*. ACM. Atlanta, Georgia, USA, Nov. 2001, pp. 49–58.

[36] Thomas R. Shiple, Gérard Berry, and Hervé Touati. "Constructive Analysis of Cyclic Circuits". In: *Proc. European Design and Test Conference (ED&TC'96), Paris, France*. Paris, France: IEEE Computer Society Press, Mar. 1996, pp. 328–333.

[37] Olivier Tardieu and Robert 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.

[38] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches". In: *ACM Transactions on Programming Languages and Systems* 13.2 (Apr. 1991), pp. 181–210.

[39] Jeong-Han Yun et al. "Detection of Harmful Schizophrenic Statements in Esterel". In: *ACM Trans. Embed. Comput. Syst.* 12.3 (Apr. 2013), 80:1–80:23.