

Submitted to Special Issue of MEMOCODE 2015 Best Papers

SCEst: Sequentially Constructive Esterel

STEVEN SMYTH, Kiel University
 CHRISTIAN MOTIKA, Kiel University
 KARSTEN RATHLEV, Kiel University
 REINHARD VON HANXLEDEN, Kiel University
 MICHAEL MENDLER, Bamberg University

The synchronous language Esterel provides determinate concurrency for reactive systems. Determinacy is ensured by the *signal coherence rule*, which demands that signals have a stable value throughout one reaction cycle. This is natural for the original application domains of Esterel, such as controller design and hardware development; however, it is unnecessarily restrictive for software development. Sequentially Constructive Esterel (SCEst) overcomes this restriction by allowing values to change instantaneously, as long as determinacy is still guaranteed, adopting the recently proposed Sequentially Constructive model of computation. SCEst is grounded in the minimal Sequentially Constructive Language (SCL), which also provides a novel semantic definition and compilation approach for Esterel.

CCS Concepts: • **Computer systems organization** → **Real-time languages**; *Embedded software*; *Real-time system specification*; • **Software and its engineering** → **Real-time systems software**; **Concurrent programming languages**; **Concurrent programming structures**; *Software safety*; *Compilers*; • **Theory of computation** → *Concurrency*;

Additional Key Words and Phrases: Synchronous Languages, Sequential Constructiveness, Esterel

ACM Reference Format:

Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. 2017. SCEst: Sequentially Constructive Esterel *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2017), 25 pages.

DOI: 0000001.0000001

1. INTRODUCTION

Embedded real-time systems or cyber-physical systems are typically *reactive systems*, meaning that they continuously react to their environment. A natural approach for developing reactive systems is to divide time into discrete *ticks*, each of which executes a reaction cycle: 1) read inputs (sensors), 2) compute a reaction, typically by calling a *tick function*, and 3) write outputs (actuators). A common requirement, in particular for safety-critical systems, is that the tick function is *determinate*, meaning that the same sequence of inputs produces the same sequence of outputs. This requirement of determinacy may be simple to achieve for traditional, sequential algorithms. Determinacy is not trivial for reactive systems, as these often entail concurrent and preemptive control-flow, which leads to race conditions. However, the family of *synchronous languages* [Benveniste et al. 2003] provides just that, determinacy for reactive systems. That is, a reactive system programmed or modeled in a

This work was supported by the German Science Foundation (DFG HA 4407/6-1 and ME 1427/6-2) as part of the PRETSY project. Author's addresses: S. Smyth, C. Motika, K. Rathlev, and R. von Hanxleden, Dept. of Computer Science, Kiel University, Kiel, Germany; E-mail: {ssm,krat,cmot,rvh}@informatik.uni-kiel.de; M. Mendler, Dept. of Computer Science, Bamberg University, Bamberg, Germany; E-mail: michael.mendler@uni-bamberg.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM. 1539-9087/2017/01-ART1 \$15.00

DOI: 0000001.0000001



Fig. 1: WriteAfterRead example.

synchronous language will always behave in the same manner, irrespective of how this system is realized, be it in hardware or software, or on a fast or a slow processor. Determinacy of concurrent interaction could also be achieved in general-purpose languages using standard mechanisms like semaphores, locks or other synchronization barriers. However, using these correctly without creating other problems such as deadlocks requires great skills on the side the programmer. Synchronous languages relieve the programmer from this burden.

The Esterel language is a well-studied, textual synchronous language that targets control-oriented systems. It offers concurrency and numerous forms of preemption and suspension [Potop-Butucaru et al. 2007]. Esterel offers shared *signals* and non-shared *variables*. Communication with signals is *instantaneous*, meaning that threads can communicate back and forth within a tick. This capability, which is lacking in many other languages, is rather powerful and valuable, as it for example facilitates the decomposition of a system without having to spread a computation across multiple ticks. For instance, in the hardware description language VHDL, which also distinguishes between signals and variables in this way, all process communication is delayed by one δ -cycle [Fuchs and Mendler 1995]. This generates determinacy trivially yet breaks the micro-step/macro-step abstraction of logical ticks. However, instantaneous communication also makes the compilation of Esterel rather challenging [Potop-Butucaru et al. 2007].

To facilitate determinacy, synchronous languages have adopted *fixed-point semantics*, which in the case of Esterel means that signals must be uniquely determined throughout a tick, that is, they must be either *present* or *absent*. This *signal coherence rule* is well suited for many application domains, such as circuits where wires should stabilize to either high or low voltage in each clock tick. A nice property of *constructive* Esterel programs is that they correspond to synchronous circuits in which the combinational next-state function may have instantaneous, yet delay-insensitive feed-back.

Motivation. The signal coherence rule is convenient from a semantical point of view, but is often considered restrictive by programmers, who are used to a more liberal model of computation that allows, for example, reading a variable in some tick and then writing a different value to the same variable, in the same tick. To illustrate, consider the minimal **WriteAfterRead** example shown in Fig. 1a. This module declares a signal **s** (which is possibly shared by concurrent threads within the signal scope), checks whether it is not present, *i.e.*, absent, and if so, makes **s** present by *emitting* it. In that case **s** will be first considered absent and then instantaneously made present; *i.e.*, **s** is *not coherent*. Thus **WriteAfterRead** is not a valid Esterel program, and an Esterel compiler will reject it as being *not causal*, or *not constructive*. However, if we consider **WriteAfterRead** as a sequential, imperative program, there is no reason to reject it, as there is a fixed ordering of the read and write of **s**, which ensures determinacy. This becomes clear when considering the equivalent, C-like program in Fig. 1b: **s** is initialized to **false**, but at the end of the reaction will have the value **true**. The motivation for the work presented here is to improve upon the capabilities of Esterel in that more programs should have a valid, natural semantics. Note that this does not necessarily mean increased expressiveness in the sense that one can write more programs or

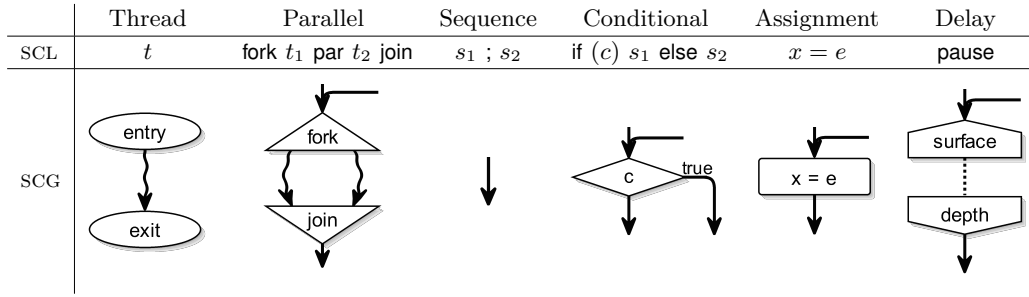


Fig. 2: Overview of SCL and SCG elements.

that programs become asymptotically smaller; however, we argue that in the Esterel variant presented here one can write programs in a more intuitive and more readable way.

Contributions and Outline. This paper presents *Sequentially Constructive Esterel (SCEst)*, which builds on the Sequentially Constructive Model of Computation (SC MoC) and the SC Language (SCL) [von Hanxleden et al. 2014b], briefly reviewed in Sec. 2. As detailed in Sec. 3, one extension of SCEst over Esterel concerns its data handling; in particular, SCEst variables can be shared among threads as well as sequentially modified. This unification not only permits programs such as **WriteAfterRead**, but also opens the door for re-initialization of signal statuses and values, expressed with the new SCEst statements **unemit** and **set**. As illustrated with the **Control** example in Sec. 4, SCEst thus adds convenience to both SCL and Esterel. The technical core of the paper is Sec. 5, which defines SCEst as a set of transformation rules from SCEst to SCL. The transformation rules are fairly straightforward, structural source-to-source transformations; therefore, one might also consider SCEst as a language extension to SCL, which adds Esterel-style control constructs (such as the various forms of preemption) as syntactic sugar to SCL. We thus leverage the existing formal semantics for SCL and build on the result [Aguado et al. 2014] that the SC MoC conservatively extends the “Berry constructiveness” (BC) demanded by Esterel. Conversely, the BC subset of SCEst without the new SCEst statements reduces to Esterel, thus our definition of SCEst can also be considered a new, alternative semantic grounding for Esterel. This unambiguously defines the semantics of SCEst, and is also the basis for a compilation procedure for SCEst, and hence also Esterel, as illustrated with experimental results in Sec. 6. We wrap up with related work in Sec. 7 and conclusions in Sec. 8.

An abbreviated version of this paper has been presented at the MEMOCODE conference [Rathlev et al. 2015]. This work expands the previous presentation in particular with respect to a more complete coverage of the SCEst transformation rules (Sec. 5), a discussion of the implementation of the transformation rules, down-stream synthesis and validation (Sec. 6), and a more in-depth coverage of related work (Sec. 7).

2. SEQUENTIAL CONSTRUCTIVENESS

We now briefly review of the SC MoC and the abstract syntax for its reference languages, the SCL and its graphical equivalent, the SCG. For full detail, including a formal semantics, we refer the reader elsewhere [von Hanxleden et al. 2014b; von Hanxleden et al. 2014a].

Fig. 2 summarizes the abstract syntax of SCL. A *thread* is a primitive or compound statement; the *parallel* instantaneously forks off multiple threads and joins (terminates) when all threads have terminated, corresponding to Esterel’s `||` operator; the *sequence*, *conditional* and *assignment* are as in standard imperative languages such as C; the *delay* corresponds to Esterel’s **pause** statement. The concrete syntax of SCL follows the C syntax, as illustrated in the SCL version of **WriteAfterRead** shown in Fig. 1b.

The SCL type system is not further specified, but we adopt Esterel’s concept of a *host language*, such as C, from which we can use types and an expression language and which can be linked to SCEst with function or procedure calls as in Esterel. To avoid confusion with the parallel operator \parallel of Esterel/SCEst, we use $|$ for (logical) *or* and similarly $\&$ for (logical) *and*. We also assume a type `bool` that can be `true` or `false`.

For purely sequential SCL programs, that is programs without parallel, the semantics is as one might expect from imperative, sequential programs. All syntactically correct purely sequential SCL programs such as `WriteAfterRead` have a well defined, determinate semantics and will not be rejected by an SCL compiler. This differs from the situation in Esterel, where even sequential programs might be rejected, as illustrated with `WriteAfterRead`.

Things become more interesting when concurrency is considered, in particular when concurrent threads share variables. SCL retains determinacy by demanding that **concurrent** accesses to the same variable follow the so-called *init-update-read discipline*, or IUR (discipline) in short. In spirit, IUR is similar to Esterel’s *emit-before-test* (EBT) discipline where the first emission of a signal must precede any test for presence of a signal. Subsequent emissions do not change the presence status anymore and hence may be scheduled before or after presence tests. With IUR, we classify writes as either *updates*, which can be scheduled in any order, typically because they are commutative and associative (such as additions/increments), or *initializations*, which is the default. More precisely, we consider an assignment as an update if it is of the form $x = f(x, e)$, where f is a *combination function* that fulfills $f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$ for all x, y_1, y_2 . In our SCEst implementation, we recognize writes as updates of type f if they are written as augmented assignments, such as, *e.g.*, $x += e$ if f is addition, and the right-hand side expression e does not reference x . More elaborate analyses might be feasible, but we actually prefer updates to be made explicit by an augmented assignment.

We say that two variable accesses are *confluent* if their order of scheduling does not matter. *E.g.*, updates of the same type (*e.g.*, of type addition) are confluent; likewise, in Esterel, signal emissions are confluent. IUR demands that for two *concurrent, non-confluent* accesses n_1 and n_2 to some variable x , n_1 must be scheduled before n_2 if (i) n_1 initializes x and n_2 updates x , or (ii) n_1 writes x and n_2 reads x . For a full, formal treatment, including a precise definition of confluence, we refer the reader elsewhere [von Hanxleden et al. 2014b]. For example, without IUR the SCL fragment `fork x+=2 par y=x par x=1 par x+=3 par x=1 join` could result in different values for y , depending on the scheduling order of the concurrent accesses to x . However, IUR demands that the (confluent) initializations of x ($x=1$) are scheduled first, in any order, followed by the two updates of x ($x+=\dots$), in any order, followed by the read of x ($y=x$). Thus y will always be assigned the value 6, and IUR guarantees that the program behavior is determinate. Note that this assumes atomicity and sequentially consistent execution of updates. This is innocuous for single-threaded compilation addressed in this paper. For multi-threaded code running on parallel architectures that suffer from weak memory anomalies (*e.g.*, see [Sewell et al. 2010]) the compiler may need to add appropriate memory barrier instructions.

A key difference between SCL’s IUR and Esterel’s EBT is that IUR only applies to *concurrent* variable accesses, while EBT applies to *all* accesses, even if they are already sequentially ordered, as in the `WriteAfterRead` example. Thus more programs are schedulable in SCL than in Esterel. Still, there are SCL programs that cannot be scheduled, *e.g.*, if they contain concurrent non-confluent initializations of the same variable; such programs are considered to be not *sequentially constructive*, and hence must be rejected.

Note that the classification of variable accesses (inits, updates, writes) and the choice of protocol (IUR) is somewhat arbitrary and not central to the SC MoC. For example, we expect that most SCL programs that do not happen to be synthesized from SCEst will not have concurrent inits and updates of the same shared variable; for such programs, a simple write-before-read protocol for concurrent, non-confluent accesses would suffice. Conversely,

	Variables			Pure Signals		Signal Values	
	C	Esterel	SCEst	Esterel	SCEst	Esterel	SCEst
Syntax	$x = y$ if (x)	$x := y$ if x	$x = y$ [$x := y$] if (x) [if x]	emit x present x	emit x, unemit x present x / if (x) [if x]	emit x(v) ?x	emit x(v), ?x set x(v), unemit x
Type	any	any	any	present / absent	present / absent	any	any
Initialized each tick	no	no	no	yes (absent)	yes (absent)	no	no
Persistence across ticks	yes	yes	yes	no	no	yes	yes
Allow multiple values per tick	yes	yes	yes	no	yes	no	yes
Sequential sched. constraints	none	none	none	first emit → reads	none	emits → reads	none
Concurrent scheduling constraints	none	read only	inits → updates → reads	first emit → reads	unemits → first emit → reads	emits → reads	unemits → sets → emits → reads
Determinate	no	yes	yes	yes	yes	yes	yes

Table I: Comparison of data handling in C, Esterel, and SCEst (which includes SCL). Note the syntactic detail that SCEst variable assignments may equivalently be written “=” (C-style, preferred) as well as “:=” (Esterel-style, for backwards compatibility, shown in square brackets). Likewise the conditional may be written C-style or Esterel-style.

there might be cases where a scheme more elaborate than IUR might be appropriate. The IUR protocol turns out to be convenient for, and was indeed motivated by, the emulation of Esterel-style signals with ordinary shared variables. This should become apparent in Sec. 5.2. The central point of the SC MoC is to take maximal advantage of the scheduling information already present in the program as expressed by sequential control-flow. It uses an underlying protocol (such as IUR) only as fall-back when a scheduler is involved (concurrency) and scheduling choices might result in non-determinacy (non-confluence).

3. DATA HANDLING—VARIABLES AND SIGNALS

A key aspect of a language, in particular of a concurrent language like SCEst, is how data are handled. Like Esterel, SCEst provides variables as well as pure and valued signals. Table I provides an overview of their characteristics.

Variables in C or other classical imperative languages may have arbitrary types, as defined by the language. C does not have a built-in concept of a tick, thus there is no implicit initialization of a variable at the beginning of a tick. Values persist (at least if they are static), and there is no limitation on the number of assignments. There are no scheduling constraints and causality errors. C compilers do not reject programs because of the way variables are written and read. However, the price to pay for this freedom is that C etc. do not guarantee I/O determinacy when concurrency or preemption are used.

Variables in Esterel are more restrictive in that concurrent accesses are limited to read accesses to achieve determinacy. In contrast, **variables in SCEst** may be used concurrently as long as the accesses are schedulable under IUR (see Sec. 2), which still provides determinacy.

Pure signals in Esterel must follow the EBT discipline (Sec. 2), for both sequential and concurrent accesses. They are initialized to absent each tick, they can be emitted/made present, but then cannot be made absent again.

```

module ReEmit:
signal A : combine boolean with or in
  emit A(true);
  if ?A then emit 0 end;
  emit A(true)
end

```

Fig. 3: ReEmit is not valid in Esterel, due to the emission of A after its value is read with ?A, but valid in SCEst.

Pure signals in SCEst can be used more liberally. They can be explicitly set to absent with `unemit`, and as with SCEst variables, there are no scheduling constraints on sequential accesses. To achieve determinate concurrency in SCEst, concurrent signal accesses follow IUR where unemits are considered initializations and emits are treated as updates, which results in unemits being scheduled before concurrent emits. The signal status may be tested with `present` or equivalently with `if`.

Valued signals in SCEst are an analogous extension of valued signals in Esterel. Both carry a non-persistent signal presence/absence status like pure signals as well as a value that is persistent across ticks. Like pure signals, their status can be initialized to absent with `unemit`. Analogously, their value can be initialized with `set`. Emissions are considered updates. Equivalent to Esterel, some *combine function* is required to handle concurrent updates or an update concurrent with an initialization.

As explained, if variable accesses are ordered by sequential control flow, the SC MoC uses that sequential ordering to schedule the variable accesses. No causality issues arise, as already illustrated in the `WriteAfterRead` example in Fig. 1. This also applies to signal values, as can be seen in `ReEmit` from Fig. 3 (taken from [Potop-Butucaru et al. 2007, p. 22]). Here A is a valued boolean signal, where multiple signal emissions are combined with the logical or. Esterel rejects this, due to the second emission of A after the value access ?A, even though that second emission does not change the value of A anymore. In contrast, SCEst accepts this program based on the schedule imposed by “;” that orders the read ?A sequentially before the second emission. Hence, IUR does *not* apply because the accesses to A are not concurrent.

4. THE CONTROL EXAMPLE

`WriteAfterRead` and `ReEmit` provided a first indication of the additional capabilities of SCEst over Esterel. To illustrate this in more detail, we consider the `Control` example presented in Fig. 4. The original SCL description is shown in Fig. 4a. As discussed elsewhere [von Hanxleden et al. 2014b], this example is an abstracted version of Programmable Logic Controller software used in the railway domain. To be clear, the functionality of `Control` as shown here could be achieved with even less code (*e.g.*, without the usage of `checkReq` or the intermediate setting of `pend` to `true`), but we here follow the logic of the original application and concentrate on the usage of the five shown flags. By revisiting this example here we now take a more systematic look at the current capabilities and limitations of Esterel compared to what is done before [von Hanxleden et al. 2014b].

The functionality of `Control` is as follows. A `Request` thread takes resource requests, indicated by `req`, from the environment and internally signals requests with `CheckReq` to a `Dispatch` thread. If a resource is available, indicated by the environment with `free`, the request is granted, signaled to the `Request` thread and the environment with `grant`. Otherwise, the request is still pending, indicated by the `Request` thread with `pend`. An example trace is shown in Fig. 4b. In the initial tick, the resource is free but not requested; in the second tick, it is free, requested, and hence granted; in the third tick, the resource is requested but not free, hence the request remains pending.

The functionality of `Control` can be expressed in a rather straightforward fashion with SCL and, as shown later, also with SCEst. However, this requires the extensions provided

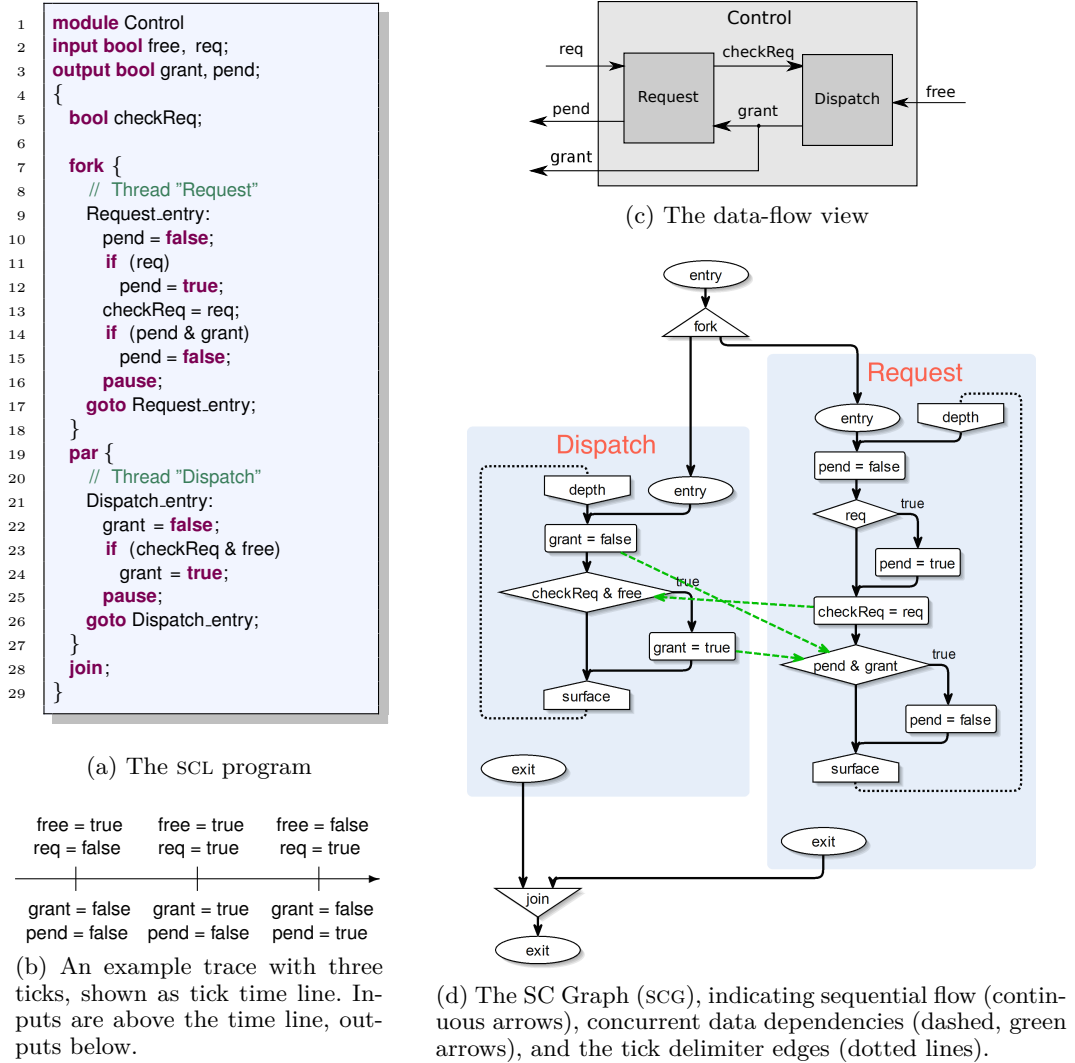


Fig. 4: The Control example, illustrating the sequential modification of shared variables.

by SCEst over Esterel. To illustrate, we now discuss different design alternatives for Control with Esterel. This also serves as a brief review of Esterel’s signal and variable mechanisms. A stumbling point is `pend`, which (1) serves to communicate with the (concurrent) environment, and (2) may change from false to true and back to false within a tick. Esterel signals can handle (1), encoding `true` as signal presence and `false` as absence. However, signals cannot deliver (2) because they must evolve monotonically within a tick and must obey the EBT discipline. Conversely, Esterel variables allow (2), but don’t allow (1).

Esterel with extra delay. One approach to resolve (2) is to split a reaction into multiple ticks. This is achieved by inserting `pause` statements and `pre`-operators that access the signal status in the previous tick, as realized in the `ControlPause` code in Fig. 5a. In the initial tick, `pend` is emitted if `req` is present; in the next tick, `pend` will be emitted *unless* `pend` and `grant`

```

1  module ControlPause:
2  input free, req;
3  output grant, pend;
4  signal checkReq in
5  % Thread "Request"
6  loop
7  present req then
8  emit pend;
9  emit checkReq;
10 end present;
11 pause; % Extra delay
12 present pre(pend)
13 and pre(grant)
14 else emit pend;
15 end present;
16 present pre(grant)
17 then emit grant;
18 end present;
19 pause;
20 end loop
21 ||
22 % Thread "Dispatch"
23 loop
24 present checkReq
25 and free then
26 emit grant;
27 end present;
28 pause;
29 pause % Extra delay
30 end loop
31 end signal
32 end module

```

(a) Esterel with extra delay

```

1  module ControlSSA:
2  input free, req;
3  output grant, pend;
4  signal checkReq, pend2 in
5  % Thread "Request"
6  loop
7  present req then
8  emit pend2;
9  emit checkReq
10 end present;
11 present pend2 and
12 not grant then
13 emit pend
14 end present;
15 pause
16 end loop
17 ||
18 % Thread "Dispatch"
19 loop
20 present checkReq
21 and free then
22 emit grant
23 end present;
24 pause
25 end loop
26 end signal
27 end module

```

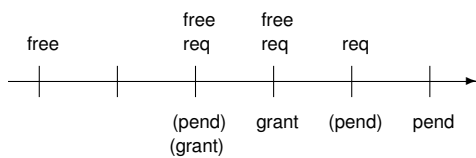
(b) SSA-style Esterel

```

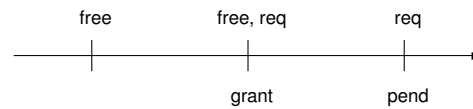
1  module ControlVar:
2  input free, req;
3  output grant, pend;
4  signal checkReq in
5  % Thread "Request"
6  loop
7  var pendv := false;
8  boolean in
9  present req then
10 pendv := true;
11 emit checkReq
12 end present;
13 if pendv then
14 present grant
15 then pendv := false
16 end present
17 end if;
18 if pendv then emit pend
19 end if
20 end var;
21 pause
22 end loop
23 ||
24 % Thread "Dispatch"
25 loop
26 present checkReq
27 and free then
28 emit grant
29 end present;
30 pause
31 end loop
32 end signal
33 end module

```

(c) Esterel with local variable



(d) Example trace for ControlPause, indicating the present (encoding true) input/output signals. Output signals occurring outside of output ticks are in parentheses.



(e) Example trace for ControlSSA/ControlVar and for ControlSCEstVar/ControlSCEstSig (Fig. 6)

Fig. 5: The Control example in Esterel with extra delay, SSA-style Esterel, and Esterel with local variables, illustrating the sequential modification of shared variables.

have been present in the initial tick (note the **else** instead of a **then**). Thus, instead of the standard behavior where at each tick inputs are consumed and outputs are produced, we have a division into *input ticks* (in **ControlPause** these are ticks 1, 3, ...) and *output ticks* (2, 4, ...). An example trace is shown in Fig. 5d. This, however, has several disadvantages:

- The environment cannot immediately access valid outputs in the same tick with the inputs.
- Introducing **pause** statements may introduce timing issues and in general results in a brittle design. This breaks the *synchrony hypothesis* which achieves robust designs by abstracting from computation times. To keep the thread logic in synch in **ControlPause**, the extra delay introduced in the **Request** thread must be matched by an extra delay in **Dispatch**. Similarly, to keep the output of **grant** in synch with the output of **pend**, **grant** must be re-emitted after the extra delay if it is present in an input tick.
- The clocking of the module must be doubled to achieve the same interaction rate with the environment.
- The interaction with the environment becomes more complicated, as outputs are not in the same tick as the inputs anymore. In **Control**, the environment should provide inputs only every other tick, and it should access the outputs one tick after providing the inputs.

Note that these disadvantages are common to languages and models of computation that do not support instantaneous reactions, *e.g.*, by disallowing instantaneous reaction to signal absence. It is a strength of Esterel that it does not have this limitation; however, in situations as the one here, where values do not evolve monotonically, that strength cannot be applied. Instead, a conceptually instantaneous computation must be broken into several parts, using a mechanism (**pause/pre**) whose original purpose is to indicate the passage of physical time, not to schedule computations. Note, we do not claim that SCEst produces *faster* code by avoiding pauses. We are more concerned with separating the passage of physical time from scheduling issues. In sum, the inability to handle (2) burdens the programmer with the task to construct a disambiguating schedule across clock ticks. In SCEst this scheduling within a tick is automatically done by the compiler.

SSA-style Esterel. Another approach to circumvent (2) is to split a signal into multiple copies. This is akin to the well-known Static Single Assignment (SSA) paradigm [Cytron et al. 1991], where variables are split up such that each copy of a variable is assigned a value only once. This is illustrated in **ControlSSA** in Fig. 5b, where a new local signal **pend2** serves as “intermediate signal.” **ControlSSA**, unlike **ControlPause**, retains the original timing and input/output behavior of **Control** and thus seems preferable over **ControlPause**. However, the disadvantage of this solution, apart from the need to introduce another signal, is that this “manual SSA-conversion” requires a close analysis of potential emissions and tests of the different instances of the re-instantiated signal. In **ControlSSA**, the programmer must analyze that **pend** is emitted (true at the end of the tick) if **pend2** is present but **grant** is absent. Again, the programmer is burdened with a transformation task that could be taken over by the compiler and in this case would be unnecessary if we were to permit variable **pend** to be reset.

Esterel with variables. Finally, we may employ a combination of Esterel variables, which allow (2), and signals, which handle (1). As in the SSA solution this retains the original timing of **Control**, but is equally cumbersome as SSA since variable values must explicitly be “copied” into signals, as illustrated in **ControlVar** in Fig. 5c (lines 18/19).

Sequentially Constructive Esterel. In contrast to Esterel, SCEst has no difficulties reconciling (1) and (2). As discussed in Sec. 3, SCEst provides variables with the same capabilities as SCL. The SCEst-equivalent of **Control** based on variables is shown in Fig. 6a. In addition, SCEst provides signals that can be used as in Esterel, but with fewer restrictions:

```

1  module ControlSCEstVar:
2  input bool free, req;
3  output bool grant, pend;
4  var checkReq: boolean in
5  % Thread "Request"
6  loop
7    pend := false;
8    if req then
9      pend := true
10   end;
11   checkReq := req;
12   if pend and grant then
13     pend := false
14   end if;
15   pause
16 end loop
17 ||
18 % Thread "Dispatch"
19 loop
20   grant := false;
21   if checkReq and free then
22     grant := true
23   end;
24   pause
25 end loop
26 end var
27 end module

```

(a) SCEst with variables

```

1  module ControlSCEstSig:
2  input free, req;
3  output grant, pend;
4  signal checkReq in
5  [
6  % Thread "Request"
7  loop
8    present req then
9      emit pend;
10   emit checkReq
11   end present;
12   present pend and
13     grant then
14     unemit pend
15   end present;
16   pause
17 end loop
18 ||
19 % Thread "Dispatch"
20 loop
21   present checkReq
22     and free then
23     emit grant
24   end present;
25   pause
26 end loop
27 ]
28 end signal
29 end module

```

(b) SCEst with signals

```

1  module ControlSCEstSig
2  input bool free;
3  input bool req;
4  output bool grant;
5  output bool pend;
6
7  fork
8    .I7: grant = false;
9    pend = false;
10   pause;
11   goto .I7;
12  par
13   bool checkReq;
14   fork
15     fork
16       .I3: if (req) {
17         pend |= true;
18         checkReq |= true };
19     if (pend & grant) {
20       pend = false };
21     pause;
22     goto .I3
23   par
24     .I5: if (checkReq & free) {
25       grant |= true };
26     pause;
27     goto .I5
28   join
29  par
30   .I6: checkReq = false;
31   pause;
32   goto .I6;
33  join
34  join

```

(c) SCEst with signals transformed to SCL

Fig. 6: The Control example in SCEst with variables, and with signals, including transformation to SCL.

- (1) SCEst signals may be emitted *after* they have been tested and possibly have been determined to be absent. There is no general EBT requirement.
- (2) SCEst signals can be *re-initialized* to absent, with the newly added `unemit` statement.

This allows modeling the behavior of `pend` in Control directly with a signal, without extra delays, signal splitting or variable copying. The resulting `ControlSCEstSig` code is shown in Fig. 6b. This is also more concise than the original SCL version since `pend` and `grant` need not be explicitly initialized to false/absent at the beginning of each tick.

5. SCEST LANGUAGE DEFINITION

The SCEst language extends Esterel in two ways. First, it uses sequential scheduling information to reduce causality problems. Second, SCEst adds three new statements: the signal handling statement `unemit` that re-initializes a signal status, the data-handling statement `set` that initializes a signal value, both as introduced in Sec. 3, and a new control flow statement, a restricted form of `goto`.



Fig. 7: Transforming loop and parallel from SCEst to SCL.

A nice feature of Esterel is that it is grounded on a small number of *kernel statements*, from which the remaining statements can be derived with simple structural translations. These *derived statements* are thus syntactic sugar that helps to write concise programs. There are some Esterel features not covered by the kernel statements, such as the interfacing with a host language, but these do not pose particular semantic challenges. Our baseline is Esterel v5, although most features of Esterel v7, such as its rich type system, should be straightforward to adopt as well. Berry [Potop-Butucaru et al. 2007] defined the following eleven *pure Esterel kernel statements* (with the classification suggested by us), where p and q are Esterel statements, S is a signal name, and T is a trap name: the *control-flow statements* `nothing`, `pause`, p ; q , `loop p end`, $p || q$ (see Sec. 5.1), the *signal handling statements* `signal S in p end`, `emit S`, `present S then p else q end` (Sec. 5.2), and the *preemption statements* `trap T`, `exit T`, and `suspend p when S` (Sec. 5.3).

In our definition of SCEst, we build on this concept by reducing the Esterel kernel statements and the aforementioned new SCEst statements to another *kernel language*, namely SCL, which is already formally defined [von Hanxleden et al. 2014b].

In the following, we present the necessary transformation rules from SCEst to SCL. These transformation rules must be applied inside-out (bottom-up in the abstract syntax tree) to generate the corresponding SCL program.

5.1. Control-Flow Statements

SCEst’s control-flow statements are rather simple to map to SCL. The `nothing` statement is not really meant to be used in programming, but helps in formalizations, *e.g.*, to ensure well-formed conditionals when a branch is missing. It corresponds to the empty statement in SCL. Esterel’s `pause` corresponds to SCL’s `pause`, the same for the sequence operator `;` and for SCEst’s `goto`. As in SCL, SCEst’s `goto` is not allowed to cross thread boundaries. The remaining control-flow transformation rules are shown in Fig. 7.

Notation: p , q are arbitrary (possibly compound) statements. Some transformations produce fresh variables or labels, indicated in their names with a leading underscore “ $_$ ”. $p[s_1 \rightarrow s_2 \mid \dots]$ replaces all s_1 in p by s_2 , for an arbitrary number of replacement patterns.

5.2. Signal Handling Statements

To emulate a signal s , we map it to a boolean variable s , and encode `present` as `true` and `absent` as `false`. The non-trivial question is how to initialize s to `absent` at each tick, before it is potentially emitted. One approach to handle initialization is to `unemit s` (*i.e.*, set it to `false`) whenever the scope of s is entered and, within the scope of s , to `unemit` it again after each tick boundary, *i.e.*, after each `pause` statement. This approach was used to derive the SCL version of `WriteAfterRead` (Fig. 1b). However, this does not scale well to signal scopes with an arbitrary number of internal tick boundaries.

For a more general solution that makes do with only one signal initialization per tick, we make use of concurrency and the IUR protocol, which permits concurrent writes under certain conditions. In particular, an initialization will be scheduled before any number of updates. What remains then is to ensure that the SCL scheduler considers `emit` as an update and only `unemit` as initialization. To achieve this, we encode `emit` not as an absolute write “`s = true`,” but instead as a relative write “`s = s | true`,” abbreviated “`s | = true`.” Note that multiple concurrent `unemits` do not pose a scheduling problem, as they are confluent.

The resulting transformation rules are shown in Fig. 8. The flag `_term` indicates when the signal scope is left and the signal initialization thread should terminate. Braces delineate

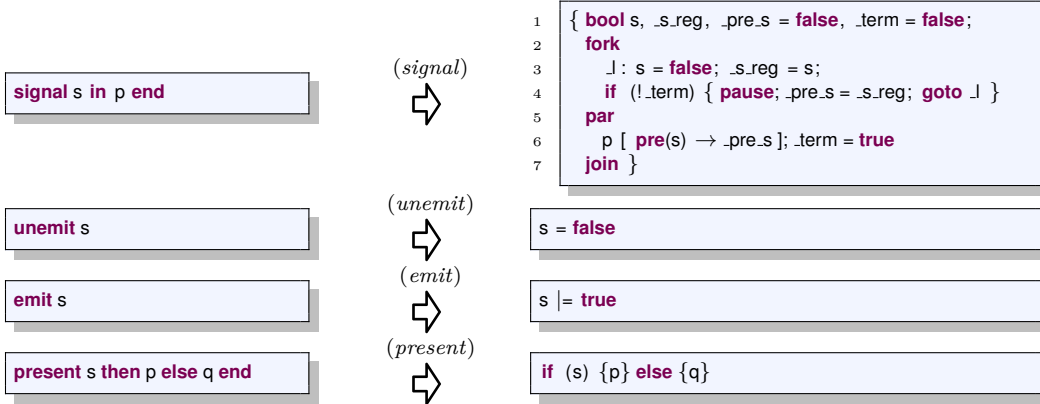


Fig. 8: Transformations for pure signals.

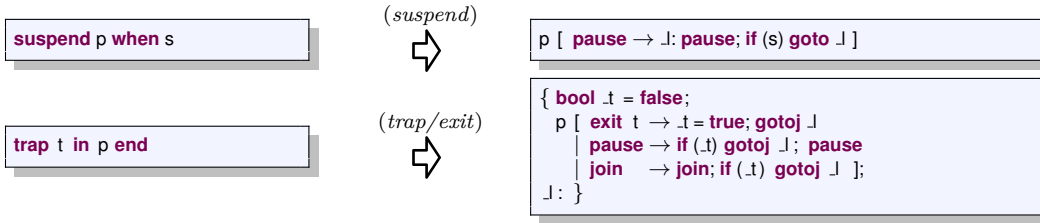


Fig. 9: Transformations for preemption statements.

the scope of s and $_term$. Signals may also be reincarnated, meaning that their signal scope is instantaneously left and entered again, in which case they are correctly re-initialized. The *(signal)* rule also supports Esterel’s **pre** operator, which allows us to access the status of a signal in the previous tick. To implement **pre**, we store the value of s at the end of a tick in $_s_reg$, and make this available at the next tick as $_pre.s$. One might also attempt to do without the $_s_reg$, by simply assigning $_pre.s = s$, but this would result in a causality loop if s would depend on $pre(s)$. Note that this makes use of sequentiality: in a tick, we first read $_s_reg$ (line 4), and then sequentially overwrite it (line 3). Obviously, if $pre(s)$ is not used, we do not need the bookkeeping in $_s_reg$ and $_pre.s$.

The rule for output signals is similar to the rule for local signals. Input signals are initialized by the environment and hence need no initialization to absence, they are mapped directly to SCL boolean inputs.

5.3. Preemption Statements

A **suspend** statement prevents the execution of the current tick’s micro steps when a specified signal s is present after the initial tick. The execution continues in the next tick in which s is absent. In SCL, **suspend** is realized by a conditional **goto** loop surrounding every **pause** in the body of the statement, see Fig. 9. Hence, at the beginning of each tick, i.e., after each **pause**, the condition is checked and, when evaluated to **true**, the **pause** is repeated. Note that Esterel does not allow self-suspension. *I.e.*, if s is emitted after the initial tick of entering the **suspend** and while control is still inside the **suspend** (we then say that we are in the *depth* of the **suspend**, as opposed to its *surface*), then s must be emitted outside of the body of the **suspend** statement. In principle, it could additionally be emitted inside the **suspend**, except that that emission is suspended. Thus we should avoid a situation where for example a **suspend** includes concurrent threads A and B, we are in the depth of the **suspend**,

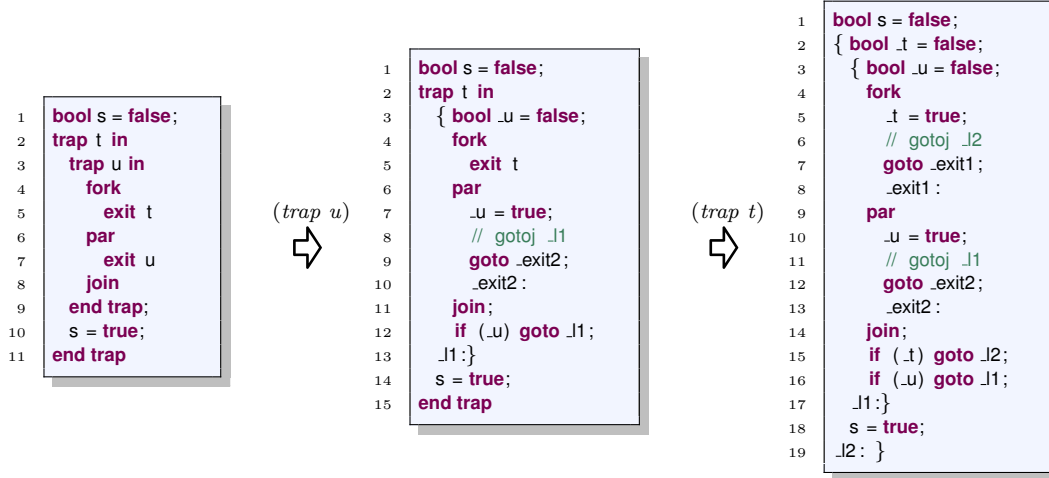


Fig. 10: Illustration of proper handling of nested traps

and thread A is the only emitter of s . This would suspend thread B but not A because A sequentially first checked s and then emitted it. A conservative approach for preventing that situation is to simply forbid emissions of s inside the body of the `suspend`.

A `trap t` preempts its body when the body executes an `exit t`, see Fig. 9. **Notation:** `gotoj l` stands for either (i) `goto l`, if label l is in the immediately enclosing thread, or otherwise (ii) `goto _exit`, where `_exit` is a fresh label added at the end of the current thread.

As `gotos` in SCL cannot jump across `joins`, we must create a chain of jumps from the `exit` through any intermediate `joins` to `_J`. This is implemented by using `gotoj _J` instead of `goto _J`, and by checking not only at each `pause` but also at each `join` whether an exception has occurred. For an illustration of `gotoj`, consider the expansion shown in Fig. 10. For example, `exit u` (line 7) produces a `goto _exit2`, which jumps to the end of the enclosing thread (which is ineffective here since we are already at the end of the thread); then in line 12, after the `join`, we check again the trap condition and jump to the end of the trap scope (`_J1`), which is again ineffective here.

The `exit` statements can only be executed within the `trap` statement scope for which they were defined. Threads that run concurrently in the statement body execute the current tick, even if the `trap` statement is triggered, and are preempted when the control reaches a tick boundary. This is signaled with the flag `_t`. Concurrent `exit` statements result in concurrent initializations of `_t`, which does not pose a scheduling problem for IUR as the initializations are confluent. Before each `pause`, `_t` is checked. As an optimization, it suffices to do so only in `pause` statements for which there is a concurrent `exit T`. If `_t` is `true`, the control jumps to the label `_J` at end of the `trap` statement.

As transformations are applied inside-out, as stated before, this also nicely handles Esterel's trap priorities which require that if nested traps are exited concurrently, the outermost trap should have priority. This is illustrated in the example in Fig. 10, where after the `join`, first the outer trap `u` is tested, and hence `s` is not emitted. The jumps in lines 7/12 in the transformed program result from the `gotoj _J1/_J2` generated for the `exit` statements; these jumps could clearly be eliminated, since the `exit` took place at the end of the enclosing thread. Likewise, the jump in line 16 is superfluous.

Note that the translation rule for traps provided here serves well to define its semantics, but is not necessarily the most efficient one for implementation purposes, both in terms of code size and execution time. Depending on the application characteristics (trap nesting depth, number of `pause` statements within trap scope) and the synthesis target (hardware or

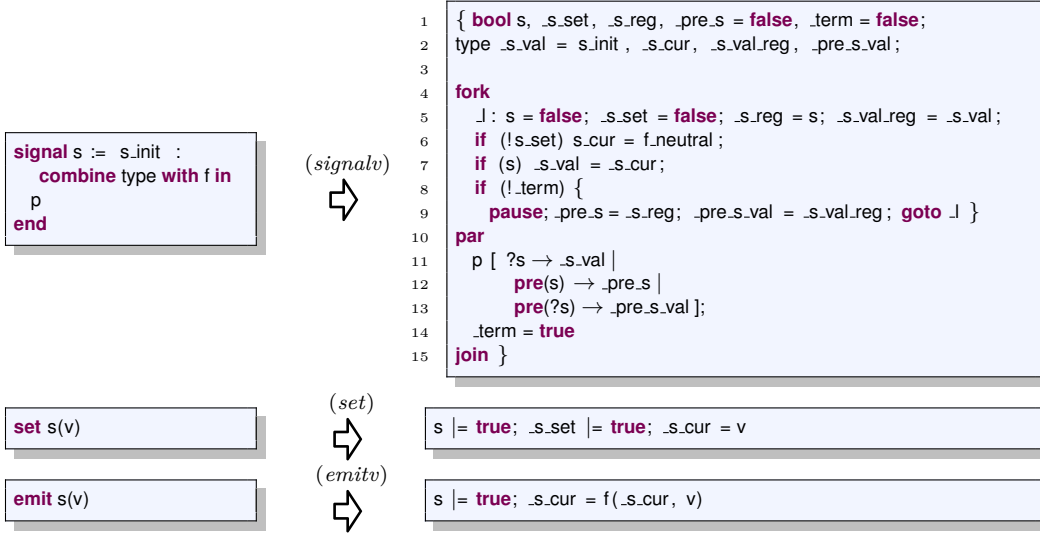


Fig. 11: Transformation for valued signals of some type with initial value `s_init`, combine function `f`, and its neutral element `f_neutral`

software), other translation approaches might be preferable, *e.g.*, by encoding trap priorities with *completion codes* as done in Esterel.

5.4. Data Handling Statements

SCEst variables map directly to SCL variables. To emulate SCEst valued signals, we adopt the approach followed in Quartz [Schneider 2001] and Potop-Butucaru’s encoding of Esterel valued signals [Potop-Butucaru et al. 2007] of splitting a valued signal into a pure signal `s` and a signal value `s_val`.

Esterel provides a construct to declare combine functions for signals to allow the concurrent emission of valued signals without compromising determinacy. For example, a signal might be combined with the addition function. We realize this by initializing the signal value to the neutral element of the combine function, denoted by `f_neutral` (which is 0 for addition), at the beginning of each tick by an absolute write. Whenever the signal is emitted with some value `v`, that value is combined (*e.g.*, added) with the current signal value, which constitutes an update. One technical issue is that if the signal is not emitted in a tick, it should keep its value from the previous tick, and should not be overwritten with the neutral element. To resolve this, the transformations shown in Fig. 11 build the new value first, with the (*emitv*) rule, in a temporary value `s_cur`, which is copied to `s_val` by the (*signalv*) rule only if `s` is emitted. The (*signalv*) rule also contains the machinery for accessing the status and value of `s` in the previous tick (`pre(s)` and `pre(?s)`, respectively), using the same logic as in the (*signal*) rule for pure signals.

To allow concurrent emits, the assignment to `s_cur` generated by (*emitv*) must be not an initialization, but an update of `s_cur`. This not only requires that `f` is a valid combination function, but also that `v` does not reference `s_cur`.

The `set` differs from the `emit` in that it uses an absolute write (initialization) to `s_cur` instead of a relative write (update). Thus, IUR ensures that a `set` is scheduled before a concurrent `emit`. As usual in the SC MoC, sequential accesses can be made in any order.

The purpose of the flag `s_set` is to make sure that only one of the initializations of `s_cur` produced by the (*signalv*) and (*set*) takes place, as these are concurrent and in general not confluent; otherwise, these would result in a scheduling conflict under the IUR proto-

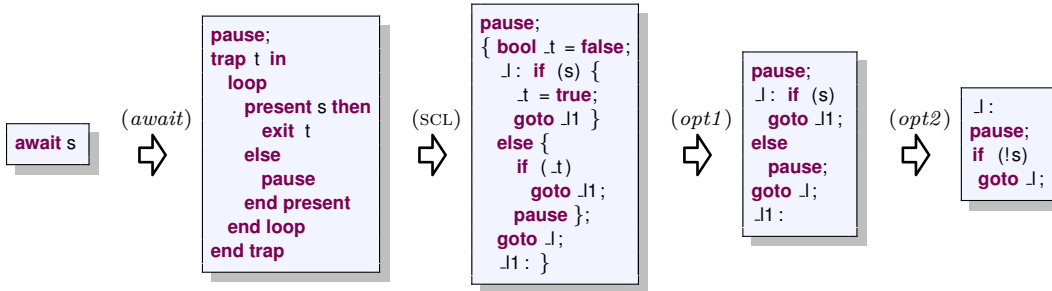


Fig. 12: The `await` transformation: *(await)* is the Esterel definition for `await`, *(SCL)* are the rules for expanding SCEst to SCL, *(opt1)* propagates `false` to the test of `.t`, as control leaves the scope of `.t` after it is set to `true` since the `pause` is not concurrent to the `exit`, *(opt2)* simplifies control-flow.

col. The downstream SCL compiler performs the (fairly straightforward) analysis that the initializations are mutually exclusive at run time because of the guard by `s.set`.

Again, the rule for output valued signals is similar, we only must consider that both the status and value become an output. Input valued signals are again initialized by the environment at each tick.

5.5. Derived Statements

The above definitions are complete in that they allow to map all of SCEst to SCL. For the semantic definition of the part of SCEst not defined yet in the previous sections, we simply adopt the expansion rules already provided for Esterel. This for example applies to the reduction of `await s` shown in Fig. 12, which pauses for a tick and from the next tick on terminates as soon as `s` becomes present. As illustrated there, the grounding in SCL may offer optimization opportunities that can be exploited with transformation rules that map derived SCEst/Esterel statements (such as `await`) directly to efficient SCL.

Fig. 13 shows further derived preemption statement transformations. As for `await`, the transformations are not constructed ad-hoc but based on the SCEst rules for the Esterel kernel statements plus classical Esterel expansion rules and subsequent optimizations. The `await immediate s` differs from (“delayed”) `await s` in that it potentially terminates from the initial tick on.

The immediate form of `suspend` differs from the non-immediate form (*suspend* rule in Fig. 9) in that the statement blocks in the initial tick if the suspension trigger `s` is present.

Similar to Esterel, a signal can be qualified by an integer `n` to form a *count delay*. The `await` will wait for `n` occurrences of the signal before it terminates. Therefore the transformation generates a new scope and counts the emissions of a signal. As long as the value of the counter is below the desired amount of signal occurrences a `pause` is iterated.

Like Esterel, SCEst provides four forms of `abort`, which differ along two independent dimensions. (1) If the abort is triggered in some tick, *weak* aborts let their body still execute in that tick, whereas *strong* aborts do not give their body control in that tick. Thus, the transformations for the weak aborts check the abort condition at the end of a tick, *i.e.*, before a `pause`, whereas the strong aborts check the abort at the beginning of a tick/after a `pause`. (2) *Immediate* aborts are sensitive to their trigger immediately, whereas *delayed* aborts are insensitive to their trigger in the initial tick that they are entered, and become sensitive to their trigger only from the subsequent tick onwards. If `s` holds in the initial tick when the abort is started, it is ignored by the delayed abort. Thus, and this is often a cause for confusion, if `s` is present in the initial tick, this does not mean that the abort is simply delayed by one tick. Instead, the presence of `s` in the initial tick is ignored all



Fig. 13: Selected derived statement transformations.

together. The delayed aborts therefore do not test the abort condition directly at each join, but use an auxiliary flag `_t` instead. The flag `_t` is set if the abort condition `S` holds *after* a `pause` statement. Thus `_t` never holds in the initial tick, which enforces the desired delayed semantics. We cannot use the abort condition `S` directly, in particular when propagating the delay after the `join` statements, since the `join` statement may in general be executed in the initial tick as well. Furthermore, the weak delayed abort also uses a `_depth` flag, which indicates that we are in the depth of the abort.

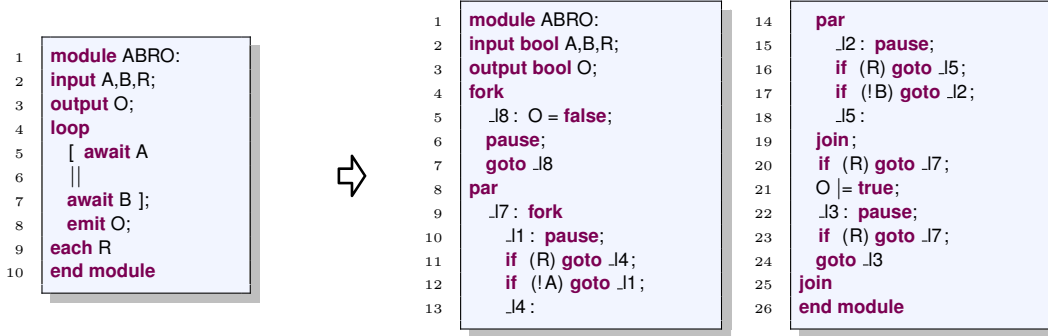


Fig. 14: Transforming ABRO from Esterel/SCEst to SCL.

The `loop p each s` restarts `p` whenever `s` holds. As an optimization, one can check whether no join is potentially reachable in the initial tick of the abort; if so, then one can do without the `.t` flag. This leads to the optimized *loope-opt* rule. Similar optimized rules for the delayed aborts are omitted here.

A `sustain` emits a specific signal infinitely resulting in an `emit` in a paused loop.

Fig. 14 illustrates the transformation (including *loope-opt*) applied to ABRO, the “hello world” program of Esterel. Lines 5–7 of the transformed version contain the thread initializing output signal `O` to absent (`false`). Lines 9–24 correspond to the original program. Lines 10–13 are the thread awaiting `A` (line 5 of the original program), lines 15–18 await `B`; see also the *await*-rule in Fig. 12. After the initial tick, the reset signal `R` restarts the loop, according to the *loope-opt*-rule in Fig. 13.

6. EXPERIMENTAL VALIDATION

To validate that the definition of SCEst is indeed conservative with respect to Esterel, we have implemented an SCCharts compiler in the open-source, Eclipse-based KIELER framework¹. The SCEst to SCL translation rules are implemented with Xtend². For downstream compilation of SCL to C, we reused the KIELER’s SCCharts compiler [von Hanxleden et al. 2014a], which also (conservatively) checks for sequential constructiveness by statically analyzing the SCG. For numerous benchmarks that tested individual language features, Eclipse unit tests compared the outputs of the generated code with what was expected. The test cases that were valid Esterel programs, *i.e.*, that did not use any new feature of SCEst, were also compared with the Columbia Esterel Compiler (CEC, version 0.4) [Edwards 2003].

6.1. SCL Compilation

To complete the picture for how SCEst can be synthesized into hardware or software, we show how our running `Control` example is compiled into a tick function using a data-flow-based code generation approach. Essentially, the initial core constructs depicted in Fig. 2 can be mapped directly to data-flow code and netlists as illustrated in Fig. 15. The data-flow equations compute a *guard* for each statement; for example, in the conditional, *g* indicates whether the conditional is executed, *g_{true}* is set if the true-branch is taken, and *g_{false}* is set when the false-branch is taken. These guards correspond to clocks in data-flow synchronous programming. For further details on this and an alternative, priority-based compilation approach we refer the reader elsewhere [von Hanxleden et al. 2014a; Smyth

¹<http://www.informatik.uni-kiel.de/rtsys/kieler>

²<http://www.eclipse.org/xtend>

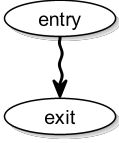
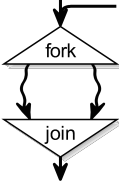
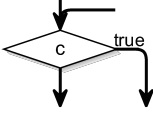
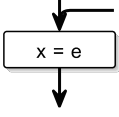
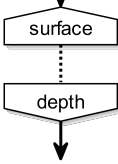
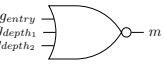
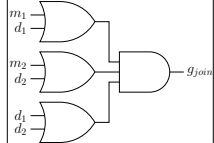
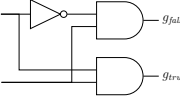
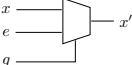

	Thread	Parallel	Conditional	Assignment	Delay
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
SCG					
Data-Flow	$d = g_{exit}$ $m = \neg \left(g_{entry} \vee \bigvee g_{depth} \right)$	$g_{join} =$ $(d_1 \vee m_1) \wedge$ $(d_2 \vee m_2) \wedge$ $(d_1 \vee d_2)$	$g = \bigvee g_{in}$ $g_{true} = g \wedge c$ $g_{false} = g \wedge \neg c$	$g = \bigvee g_{in}$ $x' = g ? e : x$	$g_{surf} = \bigvee g_{in}$ $g_{depth} =$ $pre(g_{surf})$
Netlist					

Fig. 15: Conceptual matrix (adopted from [von Hanxleden et al. 2014a]) showing the mapping between the SCG and the data-flow code leading to a synthesized netlist.

et al. 2015]. Subsequently, each schedulable SCL program w.r.t. the IUR protocol can be directly translated to hardware or software.

Fig. 4d in Sec. 4 already illustrates the first analysis step after the mapping of an SCL program to its corresponding SCG, the dependency analysis. The three write-read dependencies between the two concurrent threads are displayed as green, dashed arrows. Hence, the execution order is determined by the IUR discipline. In the control example, `grant = false` and `grant = true` in `Dispatch` are initializations and must be scheduled before the read `pend & grant` in `Request`. Similarly, `checkReq = req` in `Request` must be scheduled before the read `checkReq & free` in `Dispatch`. As explained in Sec. 2, if the concurrent dependencies form a dependency cycle, the program is not constructive and must be rejected.

After dependency analysis, the SCG is divided in *basic blocks* and a *guard* is added to each block determining the activity state of its block. This is depicted in Fig. 16. The purple rectangles enclosing the SCG nodes represent the basic blocks. They are annotated with their guard at the upper left. Below the guard variables, there are the *guard expressions* defining the guards. *E.g.*, the first `pend = false` node of the `Request` thread is enclosed in the basic block guarded by `g2`. The corresponding guard expression is `g7 || _GO` meaning that this block becomes active, if either the `_GO` signal, which indicates a program restart, or the guard `g7` are true. *Dead code* is illustrated as blocks with gray background in our tooling.

Subsequently, the scheduler creates a schedule for the SCG w.r.t. its control-flow and data dependencies. The order of the schedule is colored as the purple path in Fig. 16. In the `Control` example the schedule starts at the beginning of the program and proceeds with thread `Request` after the `fork` until it reaches `checkReq = true`. Then the schedule switches to the thread `Dispatch` because `Request` cannot continue with the read access `pend & grant` due to the IUR discipline. Hence, the schedule proceeds with `Dispatch`. Here, the read access `checkReq & free` can be executed immediately since it follows the ordering enforced by the IUR protocol. After the assignment to `grant` thread `Request` resumes execution. Then, the remaining blocks are scheduled.

One might want to reduce the number of context switches between threads, especially when performing timing analyses or measurements [Fuhrmann et al. 2014]. Since there are

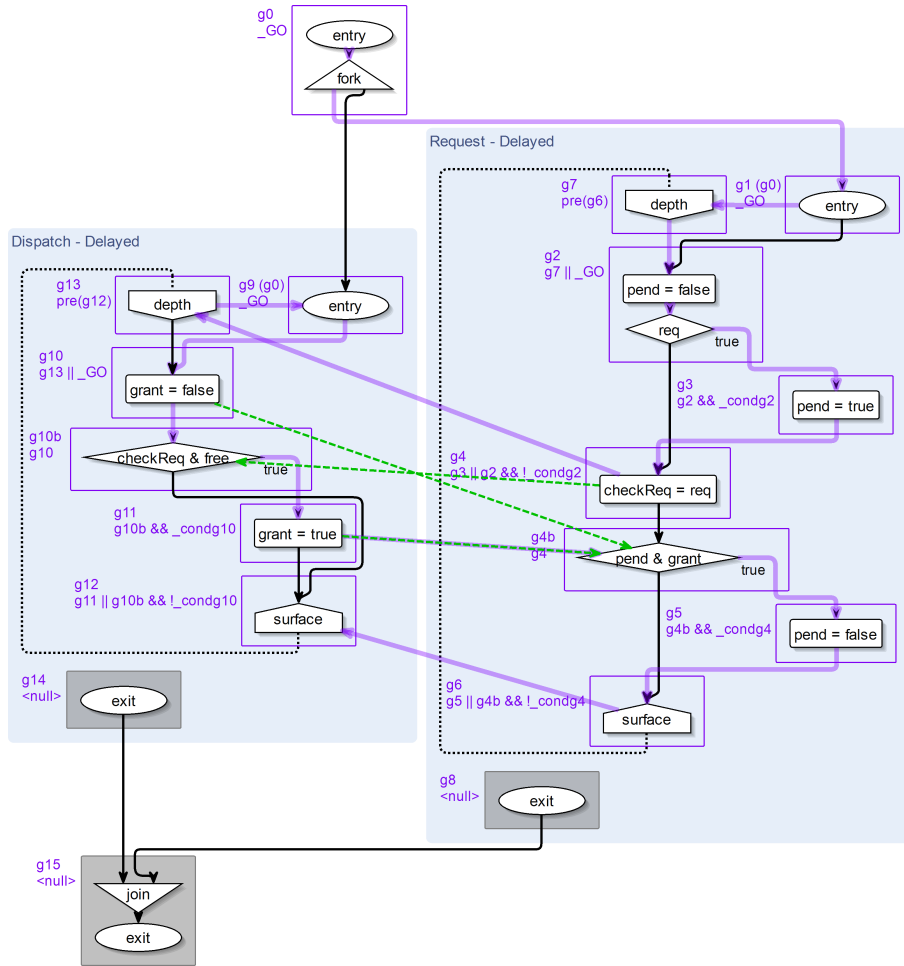


Fig. 16: The SCG for the **Control** example (Fig. 4d) after analysis for data-flow code synthesis, with scheduling path (purple edges). Basic blocks (purple rectangles) are annotated with guards (upper left) and guard expressions. Basic blocks with gray background indicate dead code.

no further interfering dependencies between the two threads subsequent to **g11**, **Dispatch** would also be eligible to finish its execution before rescheduling to the **Request** thread.

Once the program is scheduled, a new sequentialized SCG representing the data-flow netlist is created. Fig. 17 depicts the generated sequentialized SCG for the **Control** example. The assignments compute the guards and the conditionals guard actions of the corresponding basic blocks. The new SCG can directly be synthesized into a software tick function.

For hardware synthesis, one must in addition resolve multiple assignments to the same variable into different variable instances (wires), akin to SSA. This, however, is now straightforward, as all variable accesses are sequentially ordered. Furthermore, one typically wants to separate registers from combinational logic. Here, the registers are the guards that correspond to the surface nodes of the **pause** statements, as indicated by the **pre**-operators that refer to the previous tick. In the **Control** example, these are the guards **g6** and **g12**.

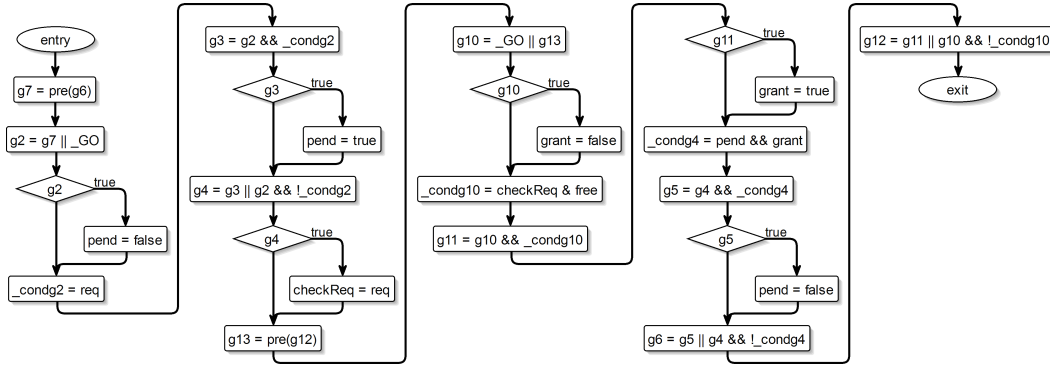


Fig. 17: The SCG of the **Control** example during the data-flow code synthesis, derived by sequentializing the SCG from Fig. 16.

6.2. Regression Testing

As stated at the beginning of Sec. 6, the tests used for validation and benchmarking were performed with the KIELER Automated Regression Testing (KART) system. Fig. 18 shows the setup of the testing system. Each Esterel test case is compiled with SCEst (upper region) and the CEC (lower region). SCEst compiles to C via the SCL. The CEC produces C code directly. Then, the gcc generates two executables and the test systems runs a simulation for each using identical input data. The output is compared to validate the results.

6.3. Experimental Results

Even though the development of a new Esterel compiler was not our primary objective, a natural question to ask is how competitive the transformation from SCEst/Esterel to SCL and further to C and ultimately executable code is with existing compilation approaches for Esterel. To that end, we evaluated our data-flow-based compilation approach by comparing sizes and reaction times of the generated code from the SCEst compiler with the CEC. We used a system with an Intel Core 2 Duo T9800 (2.93GHz) architecture. Fig. 19 shows the evaluation results, where the reaction times were averaged for 1000 ticks. These benchmarks are admittedly rather small (29 lines of code for the **Control-Sig**, to 93/171 lines of code before/after module expansion for **MsrcsFlipFlop**), and we did not extensively try the compile options offered by the CEC. Thus the results should be treated with care. However, the trend so far is that the reaction time of the SCEst is quite competitive (on average about 30% faster than CEC), but code size so far is not (on average about double). At this point, we suspect that the main culprit regarding code size is the downstream compilation from SCL to C, since for example the generation of synchronizers for the join statements so far induces unnecessary redundancies for nested threads. We see these preliminary results as indication that the approach of compiling SCEst (Esterel) by applying source-level transformations down to a very elementary language (SCL), which, for example, does not have any kind of built-in preemption or signal initialization mechanism, can be competitive with existing, sophisticated Esterel compilers. Furthermore, the fact that SCEst accepts more programs than Esterel by allowing sequential variable accesses does not seem to be an impediment to efficient compilation; put another way, we would not expect significant improvement potential if we were to forbid sequential accesses. A detailed comparison, or ideally an open evaluation platform with a benchmark suite, sample data, and an interface where compiler developers can compete, is still future work.

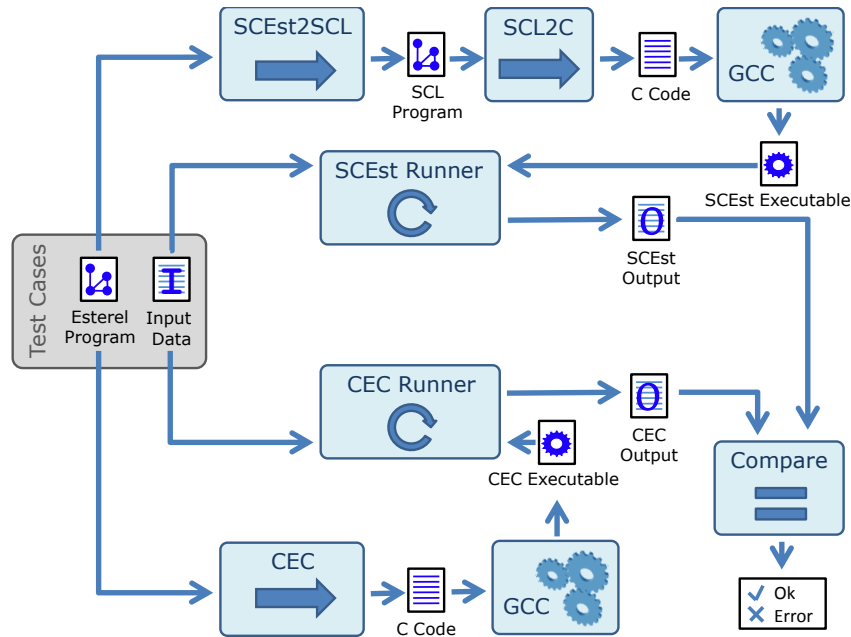


Fig. 18: Test cases setup depicting the compilation, simulation, and comparison steps between SCEst and the CEC.

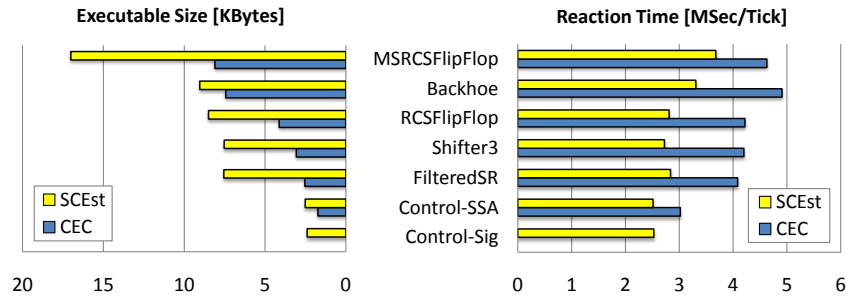


Fig. 19: Size and speed comparison of SCEst and CEC compilations. The Control-Sig example is not Berry constructive (see Sec. 4) and hence rejected by the CEC.

7. RELATED WORK

The proper handling of concurrency has a long tradition in computer science, yet, as argued by Lee [2006], still has not found its way into mainstream programming languages such as Java. Synchronous languages were largely motivated by the desire to bring determinacy to reactive control-flow, which covers concurrency and preemption [Benveniste et al. 2003].

There have been several proposals that extend C or Java with concurrency constructs. However, these typically induce race conditions for shared variables, as addressed for example by Yu et al. [2012] for OpenMP. Some of these proposed concurrency extensions avoid race conditions by building on synchronous programming principles. ECL extends C as well with Esterel-like reactive constructs [Lavagno and Sentovich 1999]; ECL programs are compiled into a reactive part (Esterel) and a data-part (C), plus some “glue logic”. FairThreads [Boussinot 2006] are an extension introducing concurrency via native

threads. Synchronous C, a.k.a. SyncCharts in C [von Hanxleden 2009], augments C with synchronous, determinate concurrency and preemption. It provides a coroutine-like thread scheduling mechanism, with thread priorities that have to be explicitly set by the programmer. PRET-C [Andalam et al. 2009] also provides determinate reactive control-flow; however, PRET-C assumes fixed priorities per thread, and thus could not execute SC programs that require back-and-forth context switching between threads. Even more restrictive is the synchronous approach ForeC [Yip et al. 2013] for multi-core execution which does not permit any communication during a tick at all. SHIM [Tardieu and Edwards 2006] provides concurrent Kahn process networks with CSP-like rendezvous communication [Hoare 1985] and exception handling. None of these language proposals embeds the concept of Esterel-style constructiveness into shared variables as we do here. As far as these language proposals include signals, they come as “closed packages” that do not, for example, allow to separate initialisations from updates.

Concerning extensions of Esterel, Tardieu and Edwards have presented two control-flow constructs, namely **gotopause** [Tardieu 2004] and **goto** [Tardieu and Edwards 2007]. These have been motivated in part by the desire to synthesize SyncCharts [André 2003], where state transitions correspond to jumps that are difficult to map to the structured control-flow offered by Esterel. They are also helpful for handling signal reincarnation. SCEst’s **goto** is more restricted in that it only allows jumps within the same thread. However, the SCEst **goto** is still capable of implementing SyncChart/SCChart transitions (which have the same restriction) and, together with the IUR protocol, of handling signal reincarnation.

Caspi et al. [Caspi et al. 2009] have extended Lustre with a shared memory model. However, they adhere to the current synchronous model of execution in that they forbid multiple writes even when they are sequentially ordered. Potop-Butucaru [Potop-Butucaru et al. 2007] introduced *data-handling kernel statements* to Esterel and with them formalized Esterel variables and valued signals. The SCEst realization of valued signals follows the approach of splitting it into a pure signal and a value, as also done in Quartz.

The functional synchronous Lucid Synchronone [Colaço et al. 2006] allows the definition of local names, which can be used to encode sequential orderings, as in `let x = ... in x = x + 1`; the same effect can be achieved by converting a program into SSA form [Appel 1998]. In Lucid Synchronone, this is motivated also by the desire to sequentialize external function calls with side effects, such as “`print`.” Unlike these Lustre variants, SCEst is not restricted to constructiveness in Berry’s sense [Berry 2000], but relaxes this requirement to sequential constructiveness (SC).

Edwards [Edwards 2003] and Potop-Butucaru et al. [Potop-Butucaru et al. 2007] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages. We present an alternative compilation approach that handles most constructs that are challenging for a synchronous languages compiler by transformation into the minimal SCL language. This applies in particular to aborts in combination with concurrency, which we, as part of the high-level compilation phase, reduce to (normal) terminations. Compared to existing approaches, this significantly simplifies down-stream compilation. Our SCG, which results from the compilation from SCEst to SCL, is closely related to the concurrent control-flow graph (CCFG) used by the Columbia Esterel Compiler (CEC) as its intermediate representation [Edwards 2003], the main difference being that we permit arbitrary control-flow including loops and that we have more refined types of data dependencies. The SCG is also related to the GRaph Code (GRC) [Potop-Butucaru et al. 2007] that, like the CCFG, has a separate structure to keep state across tick boundaries (“reconstruction tree”). In comparison, the SCG has a rather simple means to express state, namely registers that correspond to the non-transient SCChart states.

Sequentially Constructive Charts (SCCharts) [von Hanxleden et al. 2014a] is a graphical language that relates to SCEst as SyncCharts relate to Esterel. Both SCCharts and SCEst build on the SC MoC which relaxes Berry’s notion of constructiveness. Yet, like Esterel, they

still assume a locally asynchronous, globally synchronous (LAGS) model of computation. A different, largely orthogonal approach is the notion of *constructive polychrony* by Talpin et.al. [Talpin et al. 2014] to extend Esterel for multi-clocked data-flow in order to capture globally asynchronous locally synchronous (GALS) systems.

8. CONCLUSIONS AND OUTLOOK

SCEst is a new language for designing reactive systems with determinate behavior that combines the rich feature set of Esterel, in particular concerning reactive control-flow, with a data handling flexibility familiar from standard imperative languages. SCEst resolves “spurious” causality issues introduced by sequential control-flow, thus accepting a larger class of programs than Esterel. SCEst makes use of the recently proposed Sequentially Constructive model of computation, and we have defined the semantics of SCEst by providing translation rules from SCEst to SCL. Our extension of Esterel is “minimally invasive” in that we provide translation rules for the Esterel kernel statements plus three new SCEst statements (**unemit**, **set**, **goto**). We reuse existing derivations of non-kernel Esterel statements. However, for some derived statements, their expansion into kernel statements and further to SCL opens optimization possibilities. As illustrated with the **await** statement, this is made possible by having elementary features such as the **goto** available directly in SCL.

There are several issues that we did not have the space to discuss here, but for which we refer the reader to Rathlev’s thesis on SCEst [Rathlev 2015]. In particular, when compiling SCL with a data-flow-based approach that demands *acyclic SC schedulability* [von Hanxleden et al. 2014b], care has to be taken to not inadvertently introduce cycles in the resulting SCG. Rathlev also discusses the—still largely open— inclusion of **weak suspend**, as introduced by Esterel v7 and Quartz, which would facilitate for example multi-clock hardware design.

While the translation rules from SCEst to SCL primarily serve to unambiguously define the semantics of SCEst, first experiments indicate that they can also be a basis for synthesis to executable code. However, we expect that there is still room for improvement.

Experience will show how significant the extensions of SCEst over Esterel really are, both in terms of increased expressiveness and ease of use. Experienced Esterel programmers might prefer to keep sequential modifications (Esterel variables) and shared accesses (Esterel signals) separate, which is of course still possible in SCEst. However, experience in the classroom both with SCEst and its graphical counterpart SCCharts indicates that the SC MoC eases the learning curve for getting into synchronous programming, without giving up the advantages offered by Esterel and its synchronous language cousins. An interesting question, currently under investigation, is whether every SCEst program can be “easily” translated into equivalent, SSA-style Esterel, as presented for the **Control** example in Sec. 4. For purely sequential programs this seems fairly straightforward, using standard techniques. For concurrent programs, preliminary results indicate that at least in most cases a concurrent SSA is also feasible, building on Esterel’s valued signal mechanism. This could also lead to an alternative, slightly more restrictive definition of sequential constructiveness that forbids *speculation* [Potop-Butucaru et al. 2007]; the idea would be to define such programs as sequentially constructive where the equivalent, SSA-style program is constructive in the sense of Esterel.

Apart from the language extensions of SCEst, we consider the “unification” of shared signals and sequential variables as valuable, even in the context of Esterel alone. As explained elsewhere [Aguado et al. 2014], this allows, for example, a more efficient handling of schizophrenic signals at source code level, with worst-case linear instead of quadratic code increase. This is made possible by exposing the initialization of signals part at the language level, rather than hiding it within a pre-packaged signal semantics.

More generally, one of the lessons learned—or reinforced—from this work is that when used carefully, “low-level features” for the handling of data, control and state can add significantly to the power of a language. SCEst has explored this mostly in the data direction

and, to a lesser extent, regarding control with its restricted form of `goto`. We currently experiment with more flexible control and state handling mechanisms, in part following the work by Tardieu and Edwards [Tardieu 2004; Tardieu and Edwards 2007]. For example, the `pause` statement may seem very elementary, but is in fact a high-level, pre-packaged control/state construct: it causes one thread to deactivate itself in the current tick at a certain continuation point, and in the next tick causes the same thread to activate itself again at the same continuation point. Tardieu’s `gotopause` already makes this a bit more flexible by allowing that thread to activate itself again at a different continuation point, however, for some applications (such as the `weak suspend`) one might wish even more flexibility.

Finally, we expect that the further work on SCEst—and SCCharts—will also lead to new insights and variations on the underlying SC MoC. *E.g.*, adding a “pre-init” stage to IUR would facilitate to distinguish local and global initializations, as would be suitable for multi-core implementations. More generally, an interesting question is how to exploit the concurrent nature of SCEst for truly parallel/distributed execution. Girault conducted a nice survey on the distribution of synchronous programs [Girault 2005], and some of the techniques presented there should be applicable for SCEst as well. One starting point might be the data-flow-based code generation technique (see Sec. 6.1), since the resulting data-flow equations and the corresponding netlist are already highly parallel.

ACKNOWLEDGMENTS

We thank the participants of the SYNCHRON 2014 workshop and the MEMOCODE 2015 conference, in particular Gérard Berry, for the valuable feedback on the initial ideas regarding SCEst. We also thank Nis Wechselberg and Insa Fuhrmann for their suggestions on a draft of this paper, as well as the anonymous reviewers for their very helpful comments.

REFERENCES

- Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. 2014. Grounding Synchronous Deterministic Concurrency in Sequential Programming. In *Proceedings of the 23rd European Symposium on Programming (ESOP’14)*, LNCS 8410. Springer, Grenoble, France, 229–248. Long version: Technical Report 94, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, Bamberg University, August 2014, ISSN 0937-3349.
- Sidharta Andalarn, Partha Roop, Alain Girault, and Claus Traulsen. 2009. PRET-C: A new language for programming precision timed architectures. *Workshop on Reconciling Performance with Predictability (RePP’09)*, *Embedded Systems Week*, Grenoble, France.
- Charles André. 2003. *Semantics of SyncCharts*. Technical Report ISRN I3S/RR–2003–24–FR. I3S Laboratory, Sophia-Antipolis, France.
- Andrew W. Appel. 1998. SSA is functional programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20.
- Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, Vol. 91. IEEE, Piscataway, NJ, USA, 64–83.
- Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press, Cambridge, MA, USA, 425–454.
- Frédéric Boussinot. 2006. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience* 18, 5 (April 2006), 445–469.
- Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. 2009. Synchronous Objects with Scheduling Policies: Introducing Safe Shared Memory in Lustre. In *ACM Int’l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES’09)*. ACM, Dublin, Ireland, 11–20.
- Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT’06)*. ACM, Seoul, South Korea, 73–82.
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991), 451–490.

- Stephen A. Edwards. 2003. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems* 8, 2 (April 2003), 141–187.
- M. Fuchs and M. Mendler. 1995. A functional semantics for delta-delay VHDL based on FOCUS. In *Formal Semantics for VHDL*, C. D. Kloos and P. Breuer (Eds.). Kluwer, 9–42.
- Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. 2014. Towards Interactive Timing Analysis for Designing Reactive Systems. Reconciling Performance and Predictability (RePP'14), satellite event of ETAPS'14. (April 2014).
- Alain Girault. 2005. A Survey of Automatic Distribution Method for Synchronous Programs. In *International Workshop on Synchronous Languages, Applications and Programs (SLAP '05) (Electronic Notes in Theoretical Computer Science)*, F. Maraninchi, M. Pouzet, and V. Roy (Eds.). Elsevier Science, Edinburgh, UK.
- C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ.
- Luciano Lavagno and Ellen Sentovich. 1999. ECL: a specification environment for system-level design. In *Proc. 36th ACM/IEEE Conf. on Design Automation (DAC'99)*. ACM, 511–516.
- Edward A. Lee. 2006. The Problem with Threads. *IEEE Computer* 39, 5 (2006), 33–42.
- Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel*. Springer.
- Karsten Rathlev. 2015. *From Esterel to SCL*. Master thesis. Kiel University, Department of Computer Science. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf>.
- Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. 2015. 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.
- K. Schneider. 2001. Embedding Imperative Synchronous Languages in Interactive Theorem Provers. In *Conference on Application of Concurrency to System Design (ACSD'01)*. IEEE Computer Society, Newcastle upon Tyne, UK, 143–156.
- P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. 2010. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *CACM* (2010), 89–97.
- Steven Smyth, Christian Motika, and Reinhard von Hanxleden. 2015. A Data-Flow Approach for Compiling the Sequentially Constructive Language (SCL). In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörschach, Austria.
- J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S.K. Shukla. 2014. Constructive polychronous systems. *Science of Computer Programming* 96, 3 (Dec. 2014), 377–394.
- Olivier Tardieu. 2004. Goto and Concurrency—Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'04)*. Barcelona, Spain.
- Olivier Tardieu and Stephen A. Edwards. 2006. Scheduling-Independent Threads and Exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (EMSOFT'06)*. ACM, Seoul, South Korea, 142–151.
- Olivier Tardieu and Stephen A. Edwards. 2007. Instantaneous Transitions in Esterel. In *Proc. Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*. Braga, Portugal.
- Reinhard von Hanxleden. 2009. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT'09)*. ACM, Grenoble, France, 225–234.
- Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014a. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, Edinburgh, UK.
- Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. 2014b. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13, 4s (July 2014), 144:1–144:26.
- Eugene Yip, Partha S. Roop, Morteza Biglari-Abhari, and Alain Girault. 2013. Programming and Timing Analysis of Parallel Programs on Multicores. In *13th International Conference on Application of Concurrency to System Design (ACSD'13), Barcelona, Spain, 8-10 July, 2013*. 160–169.
- Fang Yu, Shun-Ching Yang, Farn Wang, Guan-Cheng Chen, and Che-Chang Chan. 2012. Symbolic Consistency Checking of OpenMp Parallel Programs. In *Proceedings of the 13th ACM Int'l Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*. ACM, 139–148.

Received January 2016; revised September 2016; accepted March 2017