SCEst: Sequentially Constructive Esterel

Karsten Rathlev, Steven Smyth,
Christian Motika and Reinhard von Hanxleden
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, 24118 Kiel, Germany
Email: {rvh,krat,ssm,cmot}@informatik.uni-kiel.de

Michael Mendler
Faculty of Information Systems
and Applied Computer Sciences
Otto-Friedrich-Universität Bamberg
An der Weberei 5, 96047 Bamberg, Germany
Email: michael.mendler@uni-bamberg.de

***Abstract*—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, which also provides a novel semantic definition and compilation approach for Esterel.**

I. 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 execution 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* [1] provides just that, determinacy for reactive systems. That is, a reactive system programmed or modeled in a 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.

The Esterel language is a well-studied, textual synchronous language that targets control-oriented systems. It offers concurrency, with the || operator, and numerous forms of preemption and suspension [2]. 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 to decompose a system without having to spread a computation across multiple ticks. However, instantaneous communication also makes the compilation of Esterel rather challenging [2].

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 delay-insensitive circuits. However, this is often considered restrictive by programmers, who are used to a more liberal model of computation that allows, for example, to read a variable in some tick and then write 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.

```
1 module WriteAfterRead:
2 signal s in ...
3   present (not s) then emit s; ... end
4 end
```

(a) Invalid Esterel, but valid SCEst

```
1 WriteAfterRead() {
2   bool s = false; ...
3   if (!s) { s = true; ... }
4 }
```

(b) Equivalent, valid C-like Program

Fig. 1: WriteAfterRead example.

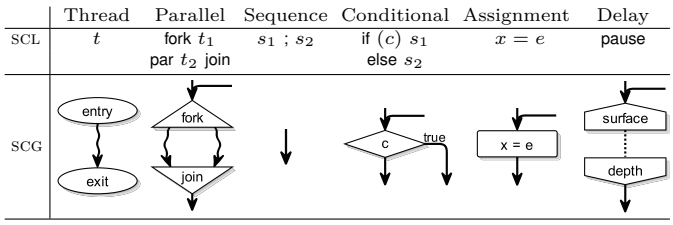


Fig. 2: Overview of SCL and SCG elements.

Contributions and Outline: This paper presents *Sequentially Constructive Esterel* (*SCEst*), which builds on the recently introduced Sequentially Constructive Model of Computation (SC MoC) and the SC Language (SCL) [3], briefly reviewed in Sec. II. As detailed in Sec. III, 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. IV, SCEst thus adds expressiveness compared to both SCL and Esterel. The technical core of the paper is Sec. V, where we define SCEst as a set of transformation rules from SCEst to SCL. We thus leverage the existing formal semantics for SCL and build on the result [4] 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 not only unambiguously defines the semantics of SCEst, but is also the basis for a compilation procedure for SCEst, and thus Esterel as well, as illustrated with experimental results in Sec. VI. We wrap up with related work in Sec. VII and conclusions in Sec. VIII.

II. Sequential Constructiveness

We here give a brief overview 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 [3], [5].

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 || of Esterel/SCEst, we use | for (logical) *or* and similarly & for (logical) *and*. We also assume a type bool.

For purely sequential SCL programs, that is programs without parallel, the semantics is as one might expect from imperative, sequential programs. In particular, 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.

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. We say that two variable accesses are *confluent* if their order of scheduling does not matter. *E.g.*, updates are confluent; likewise, in Esterel, emissions of the same signal 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 [3].) 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+=...), 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.

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.

III. 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			Pure Signals		Signal Values	
	C	Esterel	SCEst/SCL	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	arbitrary	arbitrary	arbitrary	present/absent	present/absent	arbitrary	arbitrary
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 scheduling 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
Determinacy guaranteed	no	yes	yes	yes	yes	yes	yes

TABLE I: Comparison of data handling in C, Esterel and SCEst. Note the syntactic detail that SCEst variable assignments may equivalently be written "=" (C-style, preferred) as well as ":=" (Esterel-style, for backwards compatibility). Likewise the conditional may be written C-style or Esterel-style.

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 and the like do not guarantee I/O determinacy when concurrency or preemption mechanisms 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. II), which still provides determinacy.

Pure signals in Esterel must follow the EBT discipline (Sec. II), 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.

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, and like in 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 [2, 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.

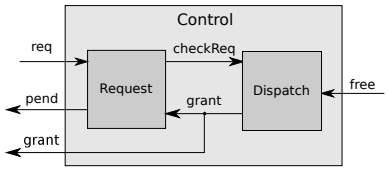
```
1 module ReEmit:
2 signal A : combine boolean with or in
3   emit A(true);
4   if ?A then emit O end;
5   emit A(true)
6 end
```

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.

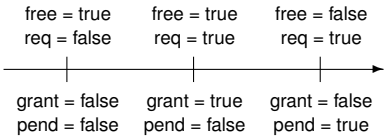
IV. The Control Example

WriteAfterRead and ReEmit provided a first indication of the additional expressiveness of SCEst over Esterel. To illustrate this expressiveness in more detail, we consider the Control example presented in Fig. 4. The original SCL description is shown in Fig. 4f. As discussed elsewhere [3], this example is an abstracted version of a Programmable Logic Controller software used in the railway domain. 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 in [3].

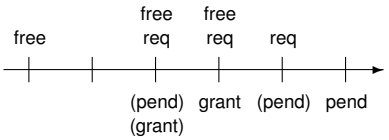
The functionality of Control is as follows. A Request thread takes resource requests, indicated by req, from the environment and internally signals requests with CheckReq to a Dispatch thread. If a resource is available, indicated by the environment with free, the request is granted, signalled to the Request thread and the environment with grant. Otherwise, the request is still pending, indicated by the



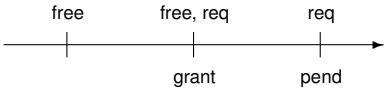
(a) The dataflow view



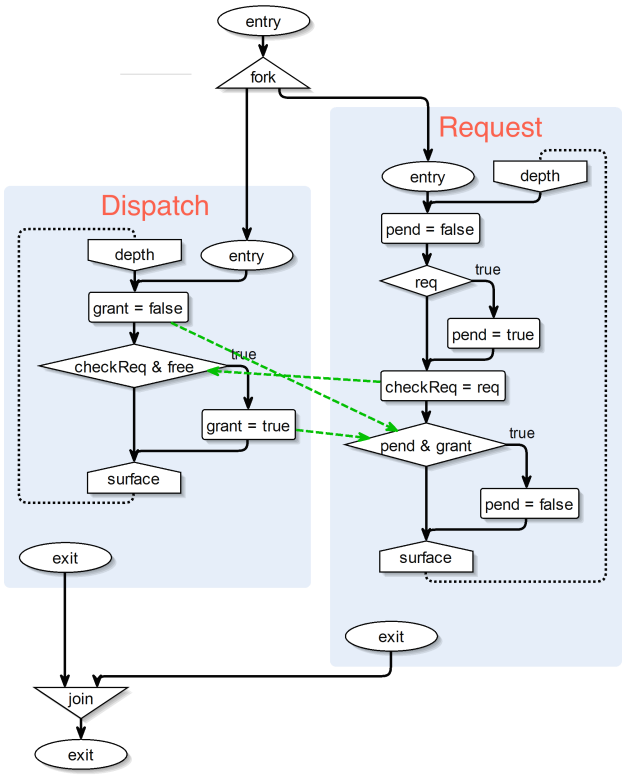
(b) An example trace with three ticks for Control, shown as tick time line. Inputs are above the time line, outputs below.



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



(d) Example trace for ControlSSA/ ControlVar/ControlSCEst.



(e) The SC Graph (SCG), indicating sequential flow (continuous arrows), concurrent data dependencies (dashed, green arrows), and the tick delimiter edges (dotted lines).

```
module Control
input bool free, req;
output bool grant, pend;
{
  bool checkReq;

  fork {
    // Thread "Request"
    Request_entry:
      pend = false;
      if (req)
        pend = true;
      checkReq = req;
      if (pend & grant)
        pend = false;
      pause;
    goto Request_entry;
  }
  par {
    // Thread "Dispatch"
    Dispatch_entry:
      grant = false;
      if (checkReq & free)
        grant = true;
      pause;
    goto Dispatch_entry;
  }
  join;
}
```

(f) The SCL program

```
module ControlPause:
input free, req;
output grant, pend;
signal checkReq in
  % Thread "Request"
  loop
    present req then
      emit pend;
      emit checkReq;
    end present;
    pause; % Extra delay
    present pre(pend)
        and pre(grant)
      else emit pend;
    end present;
    present pre(grant)
      then emit grant;
    end present;
    pause;
  end loop
||
  % Thread "Dispatch"
  loop
    present checkReq
        and free then
      emit grant;
    end present;
    pause;
    pause % Extra delay
  end loop
end signal
end module
```

(g) Esterel with extra delay

```
module ControlSSA:
input free, req;
output grant, pend;
signal checkReq, pend2 in
  % Thread "Request"
  loop
    present req then
      emit pend2;
      emit checkReq
    end present;
    present pend2 and
        not grant then
      emit pend
    end present;
    pause
  end loop
||
  % Thread "Dispatch"
  loop
    present checkReq
        and free then
      emit grant
    end present;
    pause
  end loop
end signal
end module
```

(h) SSA-style Esterel

```
module ControlVar:
input free, req;
output grant, pend;
signal checkReq in
  % Thread "Request"
  loop
    var pendv := false:
        boolean in
      present req then
        pendv := true;
        emit checkReq
      end present;
      if pendv then
        present grant
          then pendv := false
        end present
      end if;
      if pendv then emit pend
      end if
    end var;
    pause
  end loop
||
  % Thread "Dispatch"
  loop
    present checkReq
        and free then
      emit grant
    end present;
    pause
  end loop
end signal
end module
```

(i) Esterel with local variable

```
module ControlSCEstVar:
input bool free, req;
output bool grant, pend;
var checkReq: boolean in
  % Thread "Request"
  loop
    pend := false;
    if req then
      pend := true
    end;
    checkReq := req;
    if pend and grant then
      pend := false
    end if;
    pause
  end loop
||
  % Thread "Dispatch"
  loop
    grant := false;
    if checkReq and free then
      grant := true
    end;
    pause
  end loop
end signal
end module
```

(j) SCEst with variables

```
module ControlSCEstSig:
input free, req;
output grant, pend;
signal checkReq in
[
  % Thread "Request"
  loop
    present req then
      emit pend;
      emit checkReq
    end present;
    present pend and
        grant then
      unemit pend
    end present;
    pause
  end loop
||
  % Thread "Dispatch"
  loop
    present checkReq
        and free then
      emit grant
    end present;
    pause
  end loop
]
end signal
end module
```

(k) SCEst with signals

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

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.

We first 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 with extra delay: One approach to resolve (2) is to break up 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. 4g. In the initial tick, pend is emitted if req is present; in the next tick, pend will be emitted *unless* pend and grant 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. 4c. This possible solution 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 [6], 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. 4h, 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. 4i (lines 18/19).

Sequentially Constructive Esterel: In contrast to Esterel, SCEst has no difficulties reconciling (1) and (2). As discussed in Sec. III, SCEst provides variables with the same capabilities as SCL. The SCEst-equivalent of Control based on variables is shown in Fig. 4j. In addition, SCEst provides signals that can be used as in Esterel, but with fewer restrictions. Specifically,

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 to model 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. 4k. 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.

V. 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. III, and as new control flow statement a restricted form of goto.

```
module ControlSCEstSig
input bool free;
input bool req;
output bool grant;
output bool pend;

fork
  _l7: grant = false;
  pend = false;
  pause;
  goto _l7;
par
  bool checkReq;
  fork
    fork
      _l3: if (req) {
        pend |= true;
        checkReq |= true };
      if (pend & grant) {
        pend = false };
      pause;
      goto _l3
    par
      _l5: if (checkReq & free) {
        grant |= true };
      pause;
      goto _l5
    join
  par
    _l6: checkReq = false;
    pause;
    goto _l6;
  join
join
```

Fig. 5: ControlSCEstSig (Fig. 4k) transformed 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 other *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 [2] 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, $p \,||\, q$, the *signal handling statements* signal S in p end, emit S, present S then p else q end, and the *preemption statements* trap T, exit T, and suspend p when S.

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 [3].

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.

A. 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. 6.

Notation: p, q are arbitrary (possibly compound) statements. Some transformations produce fresh variables or labels, indicated in their names with an underscore "_".

Fig. 6: Transforming loop and parallel from SCEst to SCL.

B. 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.

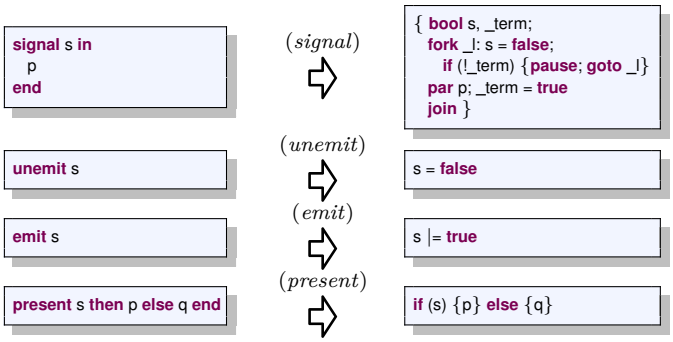


Fig. 7: Transformations for pure signals.

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." The resulting transformation rules are shown in Fig. 7. The flag _term indicates when the signal scope is left and the signal initialization thread should terminate. Braces are used to delineate 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 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.

C. Data Handling Statements

SCEst variables map directly to SCL variables. To emulate SCEst valued signals, we adopt the approach

followed in Quartz [7] and Potop-Butucaru's encoding of Esterel valued signals [2] of splitting a valued signal into a pure signal s and a signal value s_val.

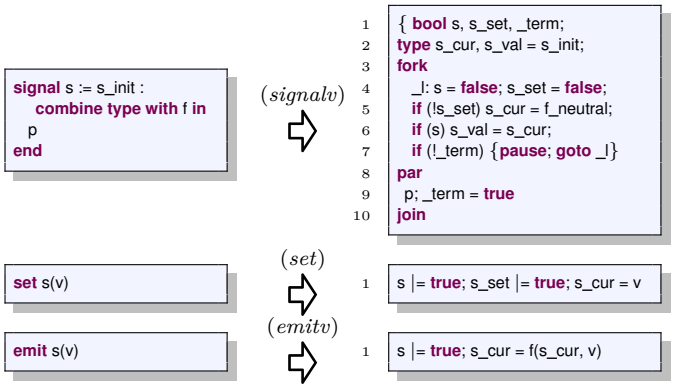


Fig. 8: Transformation for valued signals of some type with initial value s_init, combine function f and its neutral element f_neutral.

Esterel provides a basic construct to declare combine functions for signals. We realize this by initializing the signal value to the neutral element of the combine function, denoted by f_neutral, at the beginning of each tick by an absolute write. Whenever the signal is emitted with some value v, that value is combined with the current signal value. 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. 8 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 if s is emitted. References "?s" to the value of s are simply replaced by s_val.

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. Our SCEst implementation generates updates as augmented assignments, such as, *e.g.*, s_cur += v if f is addition; the downstream compilation of SCL then recognizes this as an update of s_cur provided that v does not contain 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 protocol. The downstream SCL compiler performs the (fairly straightforward) analysis that the initializations are mutually exclusive at run time.

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.

D. 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. 10. 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.

Notation: p $[s_1 \to s_2 \mid \ldots]$ replaces all s_1 in p by s_2, for an arbitrary number of replacement patterns.

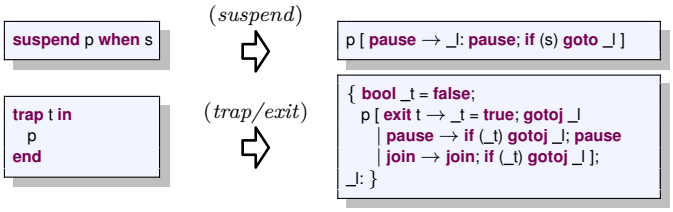


Fig. 10: Transformations for preemption statements.

A trap t preempts its body when the body executes an exit t, see Fig. 10. **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 _l. This is implemented by using gotoj _l instead of goto _l, and by checking not only at each pause but also at each join whether an exception has occurred.

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 signalled 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 _l 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. 11, where after the join, first the outer trap u is tested, and hence s is not emitted. The jumps in lines 4/6 result from the gotoj _l1/_l2 generated for the exit statements; these jumps can clearly be eliminated.

Note that the translation rule for traps provided here serves well to define its semantics, but is not necessarily

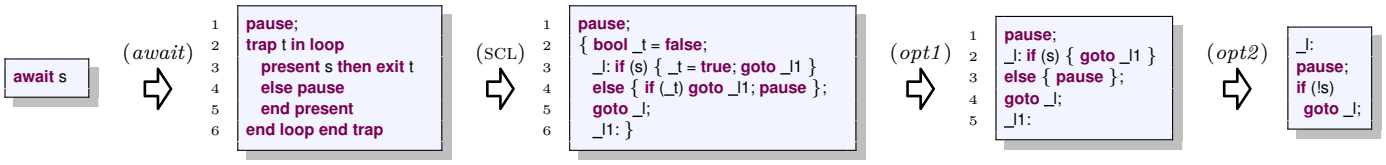


Fig. 9: await transformation: (*await*) is the Esterel definition for await, (SCL) are the rules for expanding SCEst to SCL, (*opt1*) propagates false to the test of _t, as control leaves the scope of _t after it is set to true (the pause is not concurrent to the exit), (*opt2*) simplifies control flow.

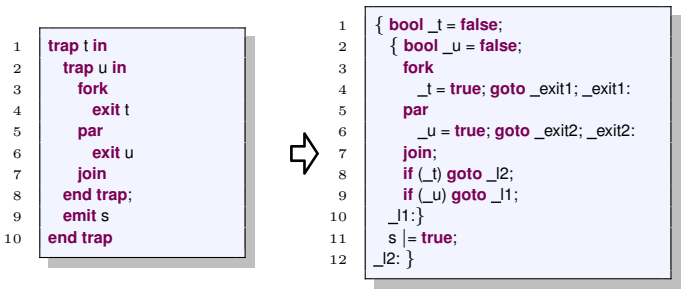


Fig. 11: Illustration of proper handling of nested traps.

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

E. Derived Statements

The above definitions are complete in that all of SCEst can be mapped to SCL. For the semantic definition of the remainder of SCEst, which includes all of Esterel, we simply adopt the expansion rules already provided for Esterel. This for example applies to the reduction of await s shown in Fig. 9, 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. 12 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. 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

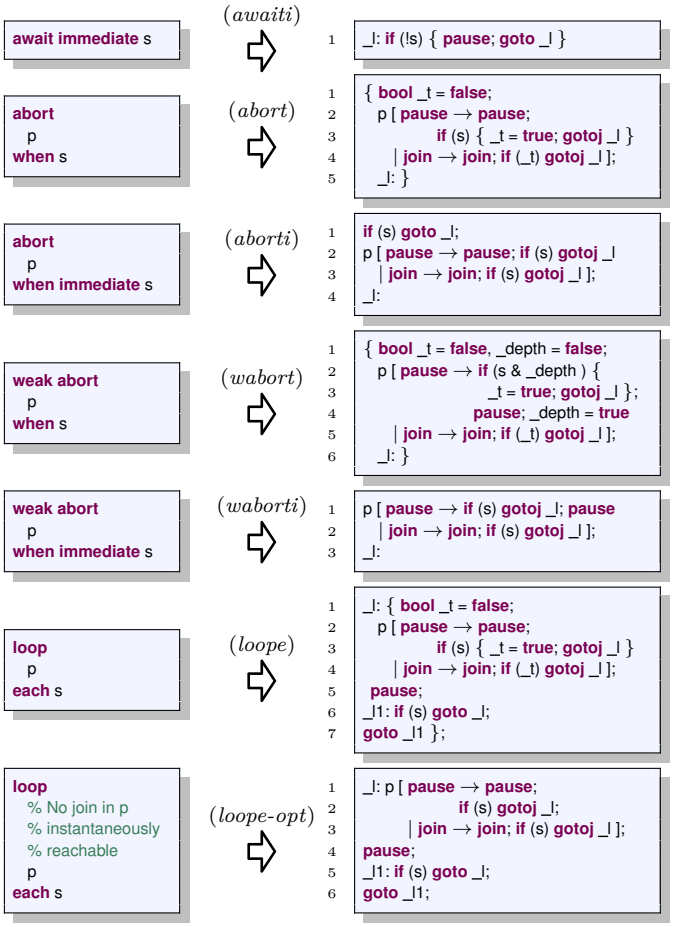


Fig. 12: Derived preemption statement transformations.

beginning of a tick/after a pause. (2) *Immediate* aborts test their trigger immediately, whereas *delayed* aborts always pause for a tick, and *then* start testing their trigger s; if s holds in the initial tick when the abort is started, it is simply ignored by the delayed abort. The delayed aborts therefore do not test the abort condition directly at each join, but use an auxiliary flag _t instead; furthermore, the weak delayed abort also uses a _depth flag.

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.)

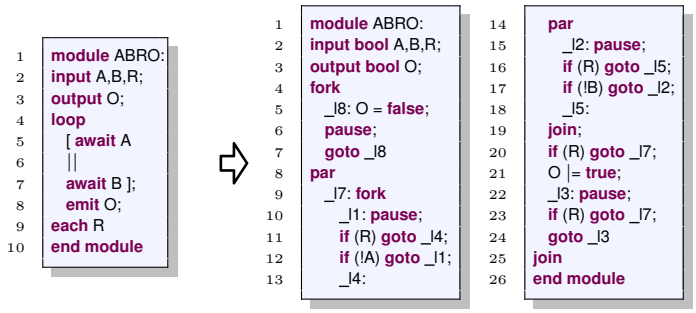


Fig. 13: Transforming ABRO from Esterel/SCEst to SCL.

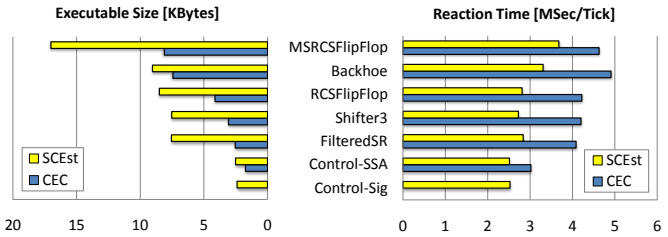


Fig. 14: Size and speed comparison of SCEst compilation and CEC. The Control-Sig example is not Berry constructive (see Sec. IV) and hence rejected by the CEC.

Fig. 13 illustrates the transformation (including *loope-opt*) applied to ABRO, the "hello world" program of Esterel.

VI. 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 framework[1]. The SCEst to SCL translation rules are implemented with Xtend[2]. For downstream compilation of SCL to C, we reused the KIELER's SCCharts compiler [5], 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) [8].

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 dataflow-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. 14 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. 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.

[1]http://www.informatik.uni-kiel.de/rtsys/kieler/
[2]http://www.eclipse.org/xtend/

VII. Related Work

There have been several proposals that extend C or Java with concurrency constructs. 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 [9]; ECL programs are compiled into a reactive part (Esterel) and a data-part (C), plus some "glue logic". FairThreads [10] are an extension introducing concurrency via native threads. PRET-C [11] also provides determinate reactive control flow; however, PRET-C assumes fixed priorities per thread, thus could not execute SC programs that require back-and-forth context switching between threads. None of these language proposals embeds the concept of Esterel-style constructiveness into shared variables as we do here.

Concerning extensions of Esterel, Tardieu and Edwards have presented two control flow constructs, namely go-topause [12] and goto [13]. These have been motivated in part by the desire to synthesize SyncCharts [14], 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. [15] 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.

Sequentially Constructive Charts (SCCharts) [5] is a graphical language that relate 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. [16] to extend Esterel

for multi-clocked data-flow in order to capture globally asynchronous locally synchronous (GALS) systems.

VIII. 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.

Experience will show how significant the extensions of SCEst over Esterel really are, both in terms of increased expressiveness and ease of use, in particular for language novices. However, 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 [4], this allows for example a more efficient handling of schizophrenic signals at source code level, with worst-case linear instead of quadratic code increase.

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 the derivation of non-kernel Esterel statements concerning their semantics. For some derived statements, their expansion into kernel statements and further to SCL opens optimization possibilities that should be exploited in a compiler. As illustrated with the await statement, this is made possible by having rather 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 [17]. In particular, when compiling SCL with a data-flow based approach that demands *acyclic SC schedulability* [3], 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.

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.

Acknowledgements

We thank the participants of the SYNCHRON 2014 workshop, 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, and the reviewers for their very helpful comments.

References

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

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

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

[4] J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann, "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, long version: Technical Report 94, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, Bamberg University, August 2014, ISSN 0937-3349.

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

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, October 1991.

[7] K. Schneider, "Embedding imperative synchronous languages in interactive theorem provers," in *Conference on Application of Concurrency to System Design (ACSD'01)*. Newcastle upon Tyne, UK: IEEE Computer Society, June 2001, pp. 143–156.

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

[9] L. Lavagno and E. Sentovich, "ECL: a specification environment for system-level design," in *Proc. 36th ACM/IEEE Conf. on Design Automation (DAC'99)*. New York, NY, USA: ACM Press, 1999, pp. 511–516.

[10] F. Boussinot, "Fairthreads: mixing cooperative and preemptive threads in C," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 5, pp. 445–469, Apr. 2006.

[11] S. Andalam, P. Roop, A. Girault, and C. Traulsen, "PRET-C: A new language for programming precision timed architectures," *Workshop on Reconciling Performance with Predictability (RePP'09), Embedded Systems Week*, Grenoble, France, Oct. 2009.

[12] O. Tardieu, "Goto and concurrency—introducing safe jumps in Esterel," in *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'04)*, Barcelona, Spain, Mar. 2004.

[13] O. Tardieu and S. A. Edwards, "Instantaneous transitions in Esterel," in *Proc. Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, Mar. 2007.

[14] C. André, "Semantics of SyncCharts," I3S Laboratory, Sophia-Antipolis, France, Tech. Rep. ISRN I3S/RR–2003–24–FR, April 2003.

[15] P. Caspi, J.-L. Colaço, L. Gérard, M. Pouzet, and P. Raymond, "Synchronous Objects with Scheduling Policies: Introducing safe shared memory in Lustre," in *ACM Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. Dublin, Ireland: ACM, Jun. 2009, pp. 11–20.

[16] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla, "Constructive polychronous systems," *Science of Computer Programming*, vol. 96, no. 3, pp. 377–394, Dec. 2014.

[17] K. Rathlev, "From Esterel to SCL," Master thesis, Kiel University, Department of Computer Science, Mar. 2015, http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf.