The Design and Implementation of the Synchronous Language CÉU

Francisco Sant'Anna, Departamento de Informática e Ciência da Computação, UERJ
Roberto Ierusalimschy, Departamento de Informática, PUC–Rio
Noemi Rodriguez, Departamento de Informática, PUC–Rio
Silvana Rossetto, Departamento de Ciência da Computação, UFRJ
Adriano Branco, Departamento de Informática, PUC–Rio

CÉU is a synchronous language inspired by Esterel with a simpler semantics and more fine-grained control over program execution targeting soft real-time systems. CÉU uses an event-triggered notion of time that enables compile-time checks to detect conflicting concurrent statements, resulting in deterministic and concurrency-safe programs. Using Esterel as a base, we present the particularities of our design, such as stack-based internal events, concurrency detection, safe integration with *C*, and first-class timers. We also present two implementation back ends: one aiming for resource efficiency and interoperability with *C*, and another as a virtual machine that allows remote reprogramming.

Additional Key Words and Phrases: Concurrency, Determinism, Embedded Systems, Esterel, Synchronous, Reactivity

1. INTRODUCTION

An established alternative to *C* in the field of embedded systems is the family of reactive synchronous languages [Benveniste et al. 2003]. Two major styles of synchronous languages have evolved: in the *control–imperative* style, programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style, programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming. Of the control-based languages, Esterel [Boussinot and De Simone 1991] was the first to appear and succeed, influencing a number of embedded languages, such as *Reactive-C* [Boussinot 1991], *OSM* [Kasten and Römer 2005], *Sync-C* [von Hanxleden 2009], and *PRET-C* [Andalam et al. 2010].

Despite its success and influence, Esterel has a complex semantics that requires careful static analysis to detect and refuse programs with *causality* and *schizophrenia* problems [Berry 1999; Shiple et al. 1996; Sentovich 1997; Boussinot 1998; Schneider and Wenz 2001; Tardieu and De Simone 2004; Edwards 2005; Yun et al. 2013]. A complex semantics not only challenges the analysis and compilation of programs, but also affects the programmer's understanding about the code, who, ultimately, has to solve the errors when facing corner cases. Furthermore, Esterel's semantics is non-deterministic for intra-reaction statements, which prevents threads from interacting with stateful system calls safely and makes shared-memory concurrency not as straightforward as reading and writing to shared variables.

In this work, we present CÉU, a new programming language that inherits the synchronous and imperative mindset of Esterel but diverges in some fundamental semantic aspects. CÉU proposes a semantics for soft real-time systems with fine-grained control for intra-reaction execution, which is amenable to concurrency checks that improve safety. The list that follows summarizes the contributions behind the design of CÉU:

— Unique and queue-based external events, which define the notion of time in CÉU.
— Stack-based internal events for intra-reaction communication, which also provides a limited form of coroutines.
— Static concurrency checks to detect suspicious concurrent statements.
— Safe integration with *C* that enforces finalization for external resources.

— First-class timers with dedicated syntax and automatic synchronization.

We also present a lightweight single-threaded implementation of CÉU with two back ends: one aiming for resource efficiency and interoperability with *C*, and another as a virtual machine that allows remote reprogramming. Our implementations target resource-constrained devices, such as *Arduino* and *MICAz* sensor nodes based on 8-bit microcontrollers[1], showing a practical aspect of the proposed semantics.

In previous work [Sant'Anna et al. 2013; Branco et al. 2015], we employed CÉU in the context of wireless sensor networks, developing a number of applications, protocols, and device drivers. We evaluated the expressiveness of CÉU in comparison to event-driven code in *C* and attested a reduction in source code size (around 25%) with a small increase in memory usage (around 5–10% for *text* and *data*) [Sant'Anna et al. 2013]. For the *VM* back end, applications have a bytecode footprint in the order of hundreds of bytes and can be transmitted over the air in a few packets [Branco et al. 2015].

The semantics of CÉU implies an implementation with a runtime stack and a sequential scheduler. In contrast with Esterel's semantics, this prevents programs to be compiled directly and efficiently to hardware [Berry 1999]. It also makes difficult to statically determine worst-case reaction times for hard real-time systems [Li et al. 2005].

The rest of the paper is organized as follows: Section 2 discusses the design of CÉU, focusing on the fundamental differences to Esterel. Section 3 presents the *C* and *VM* implementation back ends. Section 4 discusses other synchronous languages targeting embedded systems. Section 5 concludes the paper.

2. THE DESIGN OF CÉU

CÉU is a synchronous reactive language inspired by Esterel in which programs advance in a sequence of discrete reactions to external events. Like Esterel, CÉU is designed for control-intensive applications, supporting concurrent lines of execution, known as *trails*, and broadcast communication through events. Internal computations within a reaction (e.g. expressions, assignments, and system calls) are considered to take no time in accordance with the synchronous hypothesis [de Simone et al. 2005]. An `await` is the only statement that halts a running reaction and allows a program to advance in this discrete notion of time. To ensure that reactions run in bounded time and programs always progress, loops are statically required to contain at least one `await` statement in all possible paths [Sant'Anna et al. 2013; Berry 2000]. CÉU shares the same limitations with (core) Esterel and synchronous languages in general [Berry 1993]: computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay hypothesis, and cannot be directly implemented.

Figure 1 illustrates the syntactic similarities between the languages, showing side-by-side the implementations in Esterel `[a]` and CÉU `[b]` for the following control specification: *"Emit an output O as soon as inputs A and B occur. Reset this behavior each time input R occurs"* [Berry 2000]. The first phrase of the specification, awaiting and emitting the events, is translated almost identically in the two languages (ln. 5–10, in both implementations), as Esterel's '‖' and CÉU's par/and constructs are equivalent. For the second phrase, the reset behavior, the Esterel version uses an `abort-when` statement (ln. 4–11) which, in this case, serves the same purpose as CÉU's par/or (ln. 4–13): the occurrence of event R aborts the awaiting statements in parallel and restarts the enclosing `loop`.

Figure 2 presents the subset of the concrete syntax of CÉU used in the paper as a quick reference.

---

[1]Both *Arduino* and *MICAz* use the 8-bit *ATmega328* microcontroller with 32K of FLASH and 2K of SRAM.

```
1   input  A, B;
2   output O;
3   loop
4      abort
5         [
6            await A
7         ||
8            await B
9         ];
10        emit O
11     when R
12  end
13
14  .
```

```
1   input  void A, B;
2   output void O;
3   loop do
4      par/or do
5         par/and do
6            await A;
7         with
8            await B;
9         end
10        emit O;
11     with
12        await R;
13     end
14  end
```

[a] Esterel   [b] CÉU

Fig. 1.   A control specification implemented in Esterel and CÉU: *"Emit O after A and B, resetting each R."* A
par/and terminates when both trails in parallel terminate. A par/or terminates when any trail terminates,
aborting the other.

```
// DECLARATIONS

input  <type> <ids>;                 // external input  events
output <type> <ids>;                 // external output events
event  <type> <ids>;                 // internal events
var    <type> <id> = <exp>;          // a variable with initial value

// EVENT HANDLING

<id> = await <id>;                   // await an event and assign received value
<id> = await <time>;                 // await time and assign delayed delta
emit <id> => <exp>;                  // emit an event passing a value

// CONTROL FLOW

<stmt> ; <stmt>                      // sequence
if <exp> then <stmts> else <stmts> end   // conditional
loop do <stmts> end                  // repetition
do <stmts> end                       // lexical block

par/or  do <stmts> with <stmts> end  // aborts when one side terminates
par/and do <stmts> with <stmts> end  // terminates when both sides terminate
par     do <stmts> with <stmts> end  // never terminates

finalize <stmts> with <stmts> end    // block finalization

// INTEGRATION WITH C

<_id>(<exps>)                        // C call (identifier starts with '_')
native do <stmts> end                // declarations in C
native @const <_ids>                 // annotations: constant symbols
native @pure  <_ids>                 //              pure functions
native @safe  <_ids> with <_ids>     //              non—conflicting symbols
```

Fig. 2.   Subset of the concrete syntax of CÉU used in the paper.

In the subsections that follow, we discuss the main differences between CÉU and
Esterel: Unique and queue-based external events (2.1); Stack-based internal events
(2.2); Static concurrency checks (2.3); Safe integration with *C* (2.4); and First-class
synchronized timers (2.5). We finish the section with a summary of our design (2.6).
We present the formal specification of the semantics of CÉU with a simplified abstract
syntax in a separate paper [Sant'Anna 2013].

2.1. Unique and Queue-Based External Events

Esterel defines time as a discrete sequence of logical unit instants or "ticks". At each tick, the program reacts to an arbitrary number of simultaneous input events from the environment. In contrast, CÉU defines time as a discrete sequence of reactions to unique input events. At each input event, which constitutes a logical unit of time, the program reacts exclusively to it. The event-triggered execution of a program in CÉU is as follows [Sant'Anna et al. 2013]:

(1) The program initiates the "boot reaction" in a single trail (but parallel constructs may create new trails).
(2) Active trails execute until they await or terminate, one after the other. This step is named a *reaction chain*, and always runs in bounded time.
(3) The program goes idle and the environment takes control.
(4) On the occurrence of a new external input event, the environment awakes *all* trails awaiting that event. It then goes to step 2.

A program must react to an event completely before handling the next one. Based on the synchronous hypothesis, a program takes a negligible time on step 2 and is always idle on step 3. In practice, if a new external input event occurs while a reaction chain is running, it is enqueued to occur in a subsequent reaction.
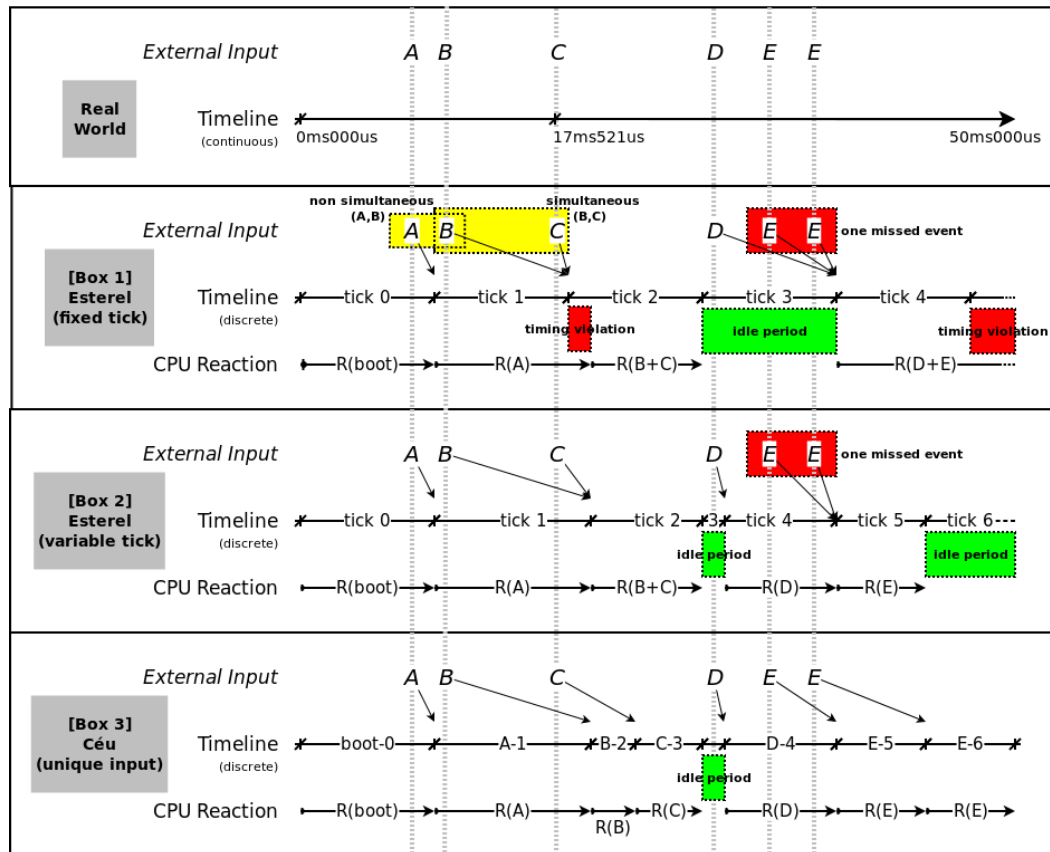

Fig. 3. The discrete notions of time in Esterel and CÉU.

Figure 3 compares the discrete notions of time in two variations of Esterel and in CÉU. The box `Real World` assumes an external observer with an absolute reference clock that timestamps event occurrences over a continuous timeline [Kopetz 2011, Chapter 3] (e.g., event `C` occurs at $17ms521us$). The other boxes show how the same occurring events fit differently in each logical notion of time.

— [`Box-1`]: *Esterel with fixed-length ticks* [Li et al. 2005]. Usually referred as *sample-driven execution* [Benveniste et al. 2003]. We assume a reaction `R(boot)` at `tick-0` which happens before any input. The input `A` "physically" occurs during the boot reaction but, because time is discrete, its corresponding reaction only executes in the next tick. Note that `R(A)` takes more time than `tick-1` and invades `tick-2`, causing a *timing violation* [Li et al. 2005]. The events `B` and `C` occur during `tick-1` and are delayed to happen *simultaneously* at `tick-2` with `R(B+C)`. Since no new events occur during `tick-2`, the CPU stays idle during the whole `tick-3`. Finally, one instance of event `D` and two instances of event `E` occur during the idle `tick-3`. However, only one occurrence of `E` can be considered in `R(D+E)`.

— [`Box-2`]: *Esterel with variable-length ticks* [Roop et al. 2004]. This approach avoids the timing violation for `R(A)` and also results in smaller idle periods because it adjusts the tick lengths to match the CPU times for the reactions. For instance, the occurrence of `D` interrupts the idle `tick-3` to react alone as `R(D)` at `tick-4`. Similarly to the fixed-tick approach, only one of the two simultaneous occurrences of `E` is considered at `tick-5`, now because `R(D)` takes too long.

— [`Box-3`]: CÉU *with unique and queue-based input events*. We also assume a reaction `R(boot)` before any input. Because the occurrence of event `A` is unique during `R(boot)`, the behavior in CÉU is similar to `Box-2` for its first two reactions (`tick-0` and `tick-1`). However, CÉU does not consider the events `B` and `C` as simultaneous, and handles each in subsequent reactions `R(B)` and `R(C)`. We assume the CPU times for `R(B+C)` in Esterel and `R(B)+R(C)` in CÉU to be roughly the same. This way, the first idle periods in `Box-2` and `Box-3` coincide. Finally, CÉU reacts to the two instances of `E` independently, which are handled in sequence.

Sample-driven execution maps directly to hardware implementations and involve no runtime queues. However, considering the soft real-time application domain of CÉU, we decided for the unique and queue-based semantics in CÉU for the reasons that follow:

— **A "tick" is implementation dependent:** Tick lengths are not part of the program specification. For instance, if we log the `Real World` box timeline and reproduce it in implementations with different tick lengths, the behaviors might diverge.

— **Events are never absolutely simultaneous:** We consider that the notion of simultaneity should not be imposed by the language, but defined explicitly for each use case (to be discussed in Section 2.5). In tick-based approaches, simultaneity depends on the length of discrete ticks. For instance, in `box-1` and `box-2` of Figure 3, events `B` and `C` are simultaneous, even though `A` and `B` "physically" happen much closer to one another.

— **Unique input events imply mutual exclusion:** Reactions to exclusive events are atomic and never overlap. Automatic mutual exclusion simplifies reasoning about concurrency and is a prerequisite for the concurrency checks to be discussed in Section 2.3.

Note that Esterel supports declarations for mutually exclusive inputs which cannot be simultaneous in the same reaction (e.g., "`relation A#B#C#D#E;`") [Berry 2000]. Making all inputs mutually exclusive and adopting the variable-length-tick approach is

```
1  input A;    // external          1  input void A;  // external (in uppercase)
2  signal B;   // internal          2  event void b;  // internal (in lowercase)
3  [[                               3  par/and do
4      await A;                     4      await A;
5      emit B;                      5      emit b;
6      call f();                    6      _f();
7  ||                               7  with
8      await B;                     8      await b;
9      call g();                    9      _g();
10 ]]                               10 end
```

            [a] Esterel   [b] CÉU

Fig. 4. Internal signals (events) in Esterel and CÉU: similar syntax, but different semantics.

equivalent to CÉU's notion of time, which can be seen as an imposed restriction on the semantics of Esterel.

The synchronous hypothesis for CÉU holds if the reactions run faster than the rate of incoming input events. Otherwise, the application continuously accumulates delays between the real occurrence and actual reaction of a given event (we discuss implementation considerations in Sections 3.5 and 3.6). This is also the case for the variable-length-tick approach of Esterel, since the more inputs to handle, the longer the reaction takes, and the more inputs accumulate for subsequent ticks. For the fixed-length-tick approach of Esterel, a breach in the synchronous hypothesis causes timing violations, requiring *worst case reaction time* analysis to infer appropriate tick lengths [Li et al. 2005]. For soft real-time systems, accumulating delays and occasionally postponing reactions might be enough. However, hard real-time systems require a more robust approach such as determining fixed-length ticks that can be statically verified.

A limitation of event-triggered execution is that all program behavior is purely reactive, given that no code can execute in the absence of inputs. Tick-triggered execution allows for active behavior, since code can execute regularly on every tick. Although CÉU supports active *asynchronous* execution [Sant'Anna et al. 2012], its synchronous core is still purely reactive.

2.2. Stack-Based Internal Events

In Esterel, the behavior of internal and external signals is equivalent. In CÉU, in contrast with queue-based external events, internal events follow a stack-based execution policy similar to subroutine calls in typical programming languages. Figure 4 illustrates the use of internal signals (events) in Esterel [a] and CÉU [b]. In Esterel, when A occurs, the program emits B (ln. 4–5) and both events become active, resulting in the invocation of f() and g() in no particular order (ln. 6,9). In CÉU, when A occurs, the program behaves as follows:

(1) 1st trail awakes, broadcasts b, and pauses (ln. 4–5).
(2) 2nd trail awakes, calls _g(), and terminates (ln. 8–9). *(No other trails awake to b.)*
(3) 1st trail (on top of the stack) resumes, calls _f(), and terminates (ln. 5–6).
(4) Both trails have terminated, so the par/and rejoins, and the program also terminates.

Internal events provide fine-grained execution control and can express a limited form of subroutines, as depicted in Figure 5. The "subroutine" inc is defined as a loop (ln. 3–6) that continuously awaits its identifying event (ln. 4), incrementing the value passed by reference (ln. 5). A trail in parallel (ln. 8–11) invokes the subroutine through an emit inc (ln. 10) in reaction to some code (ln. 9). Given the stacked execution for internal events, the calling trail pauses, the subroutine awakes (ln. 4), runs its body (yielding v=2), loops, and awaits the next "call" (ln. 4, again). Only after this sequence, the calling trail resumes and passes the assertion test (ln. 10–11).

```
1  event int* inc; // subroutine 'inc'
2  par/or do
3      loop do       // definitions are loops
4          var int* p = await inc;
5          *p = *p + 1;
6      end
7  with
8      var int v = 1;
9      <...>
10     emit inc => &v; // call 'inc'
11     _assert(v==2);  // after return
12 end
```

Fig. 5.   Subroutine inc is defined in a loop (ln. 3–6), in parallel with the caller (ln. 8–11).

```
1  event void e,f;
2  loop do
3      par/or do
4          await e;
5      with
6          emit e;       // w/o delayed awaits, the emit awakes 1st trail
7          await f;       // and restarts the loop instantaneously
8      end
9  end
```

Fig. 6.   Delayed awaits prevents re-execution of statements by design.

CÉU also supports nested emit invocations for internal events. For instance, the body of the subroutine inc in Figure 5 could emit another event after awaking (ln. 4), creating a new level in the stack. The runtime stack constitutes fine-grained micro reactions, with one on top of the other, all inside the same reaction to an external event.

On the one hand, this form of subroutine has a significant limitation that it cannot express recursive calls: an emit to itself is always ignored, given that a running body cannot be awaiting itself. On the other hand, this very same limitation brings some important safety properties to subroutines: first, they are guaranteed to react in bounded time; second, memory for locals is also bounded, not requiring data stacks. Also, this form of subroutine can use the other primitives of CÉU, such as parallel compositions and the await statement. In particular, they await keeping context information such as locals and the program counter, similarly to coroutines [de Moura and Ierusalimschy 2009]. In previous work, we build other advanced control mechanisms on top of internal events, such as resumable exceptions and reactive variables [Sant'Anna et al. 2013].

Another distinction regarding event handling in comparison to CÉU is that Esterel supports same-cycle bi-directional communication [Edwards 1999], i.e., two threads can react to one another during the same cycle due to mutual signal dependencies (a.k.a. *instantaneous dialogue* [Halbwachs 1994]). CÉU takes a different approach, posing a tradeoff that an await is only valid for the next reaction, i.e., if an await and emit occur simultaneously in parallel trails, the await does not awake. This approach avoids any form of graph construction or data dependency analysis as required in Esterel, e.g.: "set or reset of a signal must always precede any test of this signal (signal coherence law and constructivity" [Closse et al. 2002]. Also, these *delayed awaits* avoid corner cases of instantaneous termination and re-execution of statements in the same reaction (known as *schizophrenic statements* [Berry 1999]).

The example in Figure 6 illustrates delayed awaits, which prevents infinite execution by design. Both sides of the par/or have an await statement (ln. 4,7), which characterizes the enclosing loop as non instantaneous (ln 2–9). However, if the emit e

(ln. 6) could awake the `await e` instantaneously (ln. 4), the `par/or` would terminate and restart the `loop` instantaneously, resulting in infinite execution. In atypical scenarios requiring immediate awake, delayed awaits can be circumvented by placing the code to execute before the `await`. On the one hand, we transfer the burden of dealing with these corner cases to the programmer. On the other hand, we simplify the semantics of the language and eliminate the need for complex analysis to deal with schizophrenic statements.

2.3. Static Concurrency Checks

Embedded applications make extensive use of global memory and shared resources, such as through memory-mapped registers and system calls to device drivers. Hence, an important goal of CÉU is to ensure a reliable behavior for programs with concurrent lines of execution sharing memory and interacting with the environment.

Esterel is only deterministic with respect to external behavior: "the same sequence of inputs always produces the same sequence of outputs" [Berry 2000]. However, the execution order for operations within a reaction is non-deterministic: "if there is no control dependency, as in `(call f1() || call f2())`, the order is unspecified and it would be an error to rely on it" [Berry 2000]. For this reason, Esterel, does not support shared-memory concurrency: *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [Berry 2000]. A number of Esterel-inspired synchronous languages enforce an arbitrary execution order for statements in multiple lines of execution to achieve intra-reaction determinism (*Reactive C* [Boussinot 1991], *Protothreads* [Dunkels et al. 2006], *SOL* [Karpinski and Cahill 2007], *SC* [von Hanxleden 2009], and *PRET-C* [Andalam et al. 2010]). CÉU also takes the deterministic approach and, when multiple trails are active during the same reaction, they are scheduled in lexical order, i.e., in the order they appear in the program source code.

Even so, we consider that enforcing an arbitrary execution order can be misleading in some cases. For instance, consider the two examples in Figure 7, both defining a shared variable (ln. 2), and assigning to it in parallel trails (ln. 5, 8). In the example `[a]`, the two assignments to x can only execute in reactions to different events `A` and `B`, which cannot occur simultaneously by definition (Section 2.1). Hence, for the sequence `A->B`, x becomes 4 (`(1+1)*2`), while for `B->A`, x becomes 3 (`(1*2)+1`). In the example `[b]`, the two assignments to y are simultaneous because they execute in reaction to the same event `A`. Since CÉU employs lexical order for intra-reaction statements, the execution is still deterministic, and y always becomes 4 (`(1+1)*2`). However, an (apparently innocuous) change in the order of trails modifies the semantics of the program, which we consider unsafe.

To mitigate this threat, CÉU performs concurrency checks at compile time to detect conflicting accesses to shared variables: if a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable, nor dereference a

```
1  input void A, B;
2  var int x = 1;
3  par/and do
4      await A;
5      x = x + 1;
6  with
7      await B;
8      x = x * 2;
9  end
```

```
1  input void A;
2  var int y = 1;
3  par/and do
4      await A;
5      y = y + 1;
6  with
7      await A;
8      y = y * 2;
9  end
```

[a] Accesses to x are safe   [b] Accesses to y are unsafe

Fig. 7. Shared-memory concurrency in CÉU: example `[a]` is safe because the trails access x atomically in different reactions; example `[b]` is unsafe because both trails access y in the same reaction.

pointer of that variable type. Concurrency in CÉU is characterized when two or more trail segments in parallel react to the same input event. A trail segment is a sequence of statements followed by an `await` (or termination). Considering the examples in Figure 7:

— The assignments to x (`[a]`:2,5) **cannot** be concurrent because they are **not** in parallel trails.
— The assignments to x (`[a]`:5,8) **cannot** be concurrent because they **cannot** execute during the same reaction.
— The assignments to y (`[b]`:5,8) **can** be concurrent because they are in parallel trails and **can** execute during the same reaction.

The detection algorithm, which is depicted in Section 3.1, inspects all possible `await` statements that precede a variable access and keeps a list with all corresponding awaking events. Then, it checks all accesses in parallel trails to see if they share an awaking event. If it is the case, the compiler warns about the suspicious accesses.

The uniqueness of input events within reactions makes this analysis possible, otherwise, any two trail segments in parallel could be concurrent, even if they react to different input events. Note that the static checks are optional and do not affect the semantics of the program.

2.4. Safe Integration with *C*

In CÉU, any identifier prefixed with an underscore is passed unchanged to the *C* compiler that generates the final binary. Therefore, access to *C* is straightforward and syntactically trackable. Similarly to Esterel with the `call` primitive, external calls are assumed to be instantaneous [Berry 2000]. This way, programs should only resort to *C* for asynchronous functionality, such as non-blocking I/O, or simple `struct` accessors, but never for control purposes.[2]

2.4.1. Concurrency Checks. As a safety measure, the concurrency checks of Section 2.3 also consider concurrent calls and accesses to external symbols in *C*. As an example, the program in Figure 8.a defines four external symbols inside a `native` block with standard declarations in *C* (ln. 1–6). During the boot reaction, two trails react concurrently inside the par/and (ln. 7–11): the first trail calls symbol `_f` (ln. 8), while the second calls `_g` and `_id`, and also reads `_NUM` (ln. 10). Since CÉU does not inspect any code in *C*, it complains about suspicious concurrent accesses between `_f` and all symbols in the second trail.

---

[2]In CÉU, it is possible to restrict the available *C* symbols as a compile-time option.

```
1   native do                              native @const _NUM;
2       ##define NUM 10                     native @pure  _id();
3       void f  (void)  { <...> }           native @safe  _f() with _g();
4       void g  (int v) { <...> }
5       int  id (int v) { <...> }
6   end
7   par/and do
8       _f();
9   with
10      _g(_id(_NUM));
11  end
```

    [a] Definitions and uses of symbols   [b] Annotations for the symbols in [a]

Fig. 8. The unsafe program in [a] only compiles with the annotations in [b].

```
par do                                        1   native do
    <...> // animate and redraw "background"  2       #define redraw_non_commutative redraw
    _redraw(background);                       3   end
with                                          4   native @safe _redraw_non_commutative with
    <...> // animate and redraw "foreground"  5           _redraw_non_commutative;
        _redraw(foreground);                   6   par do
end                                           7       <...> // animate and redraw "background"
                                              8           _redraw_non_commutative(background);
                                              9   with
                                              10      <...> // animate and redraw "fore"
.                                             11          _redraw_non_commutative(foreground);
                                              12  end
```

[a] _redraw **cannot** be concurrent   [b] _redraw_non_commutative **can** be concurrent

Fig. 9.   Making the non-commutative redrawing calls from [a] to compile in [b].

2.4.2. Annotations. The annotations in Figure 8.b provide hints to the compiler about the semantics of the *C* symbols in program [a], which now compiles successfully:

— _NUM is a constant symbol, meaning that it is safe to use it concurrently with any other symbol in the program.
— _id is a pure function, also meaning that it is safe to call it concurrently with any other symbol in the program.
— Both _f and _g are impure, but have non-conflicting commutative effects, and can be safely called concurrently.

From our experience, however, we find that programs often need non-commutative concurrent calls. This is the case for logging (e.g., calls to _printf in trails in parallel) and for redrawing objects in the screen. Figure 9.a shows an abstract code to animate and redraw the objects background and foreground in trails in parallel. In typical graphical APIs, consecutive calls to _redraw overwrites conflicting pixels, which makes the calls non commutative and prevents the code to compile. However, in this case we want to rely on lexical order to always redraw the background object before the foreground object. Therefore, in Figure 9.b, we redefine _redraw as _redraw_non_commutative (ln. 2) and annotate it as safe (ln. 4–5) to make the code compile successfully. The new function name is explicit about its effect, improving code readability.

2.4.3. Finalization. Esterel's abort and CÉU's par/or statements provide orthogonal abortion of lines of execution, which is a distinctive feature of synchronous languages in comparison to asynchronous languages [Berry 1993]. However, aborting lines of execution that deal with external resources may lead to inconsistencies. For this reason, Esterel and CÉU provide a finalize construct to unconditionally execute a series of statements even if the enclosing block is aborted and does not terminate normally.

CÉU also enforces the use of finalize for system calls that deal with pointers representing resources, as illustrated in the two examples of Figure 10:

— If CÉU **passes** a pointer to a system call (ln. [a]:5), the pointer represents a **local** resource (ln. [a]:2) that requires finalization (ln. [a]:7).
— If CÉU **receives** a pointer from a system call return (ln. [b]:4), the pointer represents an **external** resource (ln. [b]:2) that requires finalization (ln. [b]:6).

CÉU tracks the interaction of system calls with pointers and requires finalization clauses to accompany them. In the example in Figure 10.a, the local variable msg (ln. 2) is an internal resource passed as a pointer to _send_request (ln. 5), which is an asynchronous call that transmits the buffer in the background. If the block aborts (ln. 11) before receiving an acknowledge from the environment (ln. 9), the local msg goes out of scope and the external transmission now holds a *dangling pointer*. The finalization

```
1   par/or do
2       var _buffer_t msg;
3       <...> // prepare msg
4       finalize
5           _send_request(&msg);
6       with
7           _send_cancel(&msg);
8       end
9       await SEND_ACK;
10  with
11      <...>
12  end
13  .
```

```
1   par/or do
2       var _FILE* f;
3       finalize
4           f = _fopen(...);
5       with
6           _fclose(f);
7       end
8       _fwrite(..., f);
9       await A;
10      _fwrite(..., f);
11  with
12      <...>
13  end
```

[a] Local resource finalization   [b] External resource finalization

Fig. 10.   CÉU enforces the use of finalization to prevent *dangling pointers* for local resources and *memory leaks* for external resources.

```
1   var int v;
2   await 10ms;
3   v = 1;
4   await 1ms;
5   v = 2;
6
7
8
9   .
```

```
1   par/or do
2       await 10ms;
3       <...>        // any non—awaiting sequence
4       await  1ms;
5       v = 1;
6   with
7       await 12ms;
8       v = 2;
9   end
```

[a]   [b]

Fig. 11.   First-class timers in CÉU.

ensures that the transmission also aborts (ln. 7). In the example in Figure 10.b, the call to _fopen (ln. 4) returns an external file resource as a pointer. If the block aborts (ln. 12) during the await A (ln. 9), the file remains open as a *memory leak*. The finalization ensures that the file closes properly (ln. 6). In both cases, the code does not compile without the finalize construct.

2.5. First-Class Timers

Activities that involve reactions to *wall-clock time*[3] appear in typical patterns of embedded development [Bourke and Sowmya 2009], such as timeout watchdogs and sensor samplings. However, the interaction between system clocks and programs is not absolutely precise, a fact that is usually ignored in the development process. We define the difference between a requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in a row (or in a loop) may accumulate a considerable amount of deltas that can lead to incorrect behavior in programs.

The await statement of CÉU supports wall-clock time and handles deltas automatically, resulting in more robust applications. Due to platform overheads and occasional slow reactions, an await 10ms suspends a line of execution for *at least* 10 milliseconds. In the example in Figure 11.a, suppose that after the first await request, the underlying system gets busy and takes 15ms to notify CÉU. The scheduler will notice that the await 10ms (ln. 2) has not only already expired, but is delayed with delta=5ms. Then, the awaiting trail awakes, sets v=1 (ln. 3), and invokes await 1ms (ln. 4). As the current delta is still higher than the requested timeout (i.e. $5ms > 1ms$), the trail is rescheduled for execution, now with delta=4ms.

---

[3]By wall-clock time we mean the passage of time from the real world, measured in hours, minutes, etc.

```
1   input void LEFT_CLICK;                        13  #define AWAIT_AND(e1, e2)   \
2   input void RIGHT_CLICK;                       14      par/and do              \
3   event void middle_click;                      15          await e1;           \
4   loop do                                       16      with                    \
5       par/or do                                 17          await e2;           \
6           AWAIT_AND(LEFT_CLICK, RIGHT_CLICK);   18      end                     \
7           emit middle_click;                    19  #define AWAIT_OR(e1, e2)    \
8       with                                      20      par/or do               \
9           AWAIT_OR(LEFT_CLICK, RIGHT_CLICK);    21          await e1;           \
10          await 200 ms;                         22      with                    \
11      end                                       23          await e2;           \
12  end                                           24      end
```

Fig. 12. Application defines that a middle_click event occurs whenever both LEFT_CLICK and RIGHT_CLICK occur within 200 milliseconds. The macros AWAIT_AND (ln. 13–18) and AWAIT_OR (ln. 19–24) are simple expansions to a par/and and par/or for better readability.

Delta compensation also implies that timers can be added and compared when reasoning about time as a physical quantity. For instance, in the example in Figure 11.b, although the scheduler cannot guarantee that the first trail terminates exactly in 11ms (ln. 2,4), it can at least ensure that the program always terminates with v=1. Given that any non-awaiting sequence is considered to take no time in the synchronous model, the first trail (ln. 2–5) is guaranteed to awake first and terminate the par/or before the second trail executes (ln. 7–8), because $10 + 1 < 12$. A similar program in a language without first-class support for timers would depend on the execution timings for the code marked as <...>, making the reasoning about the execution behavior more difficult.

In Section 2.1, we argue that event occurrences are infinitesimal and can never be absolutely simultaneous. However, the "sensation of simultaneity" is not infinitesimal, but actually increases with the inaccuracy of the observer (e.g., a human being). Therefore, we consider that simultaneity should be defined case by case, and should not be imposed by the language. First-class timers simplify the implementation of application-defined simultaneity. Figure 12 emulates a middle_click event (ln. 3) in terms of "simultaneous" occurrences of LEFT_CLICK and RIGHT_CLICK (ln. 1–2). If both events occur, we emit the internal event middle_click (ln. 6–7). However, if one of them occurs and the *200ms* timer expires (ln. 9–10), we abort the whole behavior with the par/or (ln. 5) and try again with the enclosing loop (ln. 4). In this specification, "simultaneous" means "within 200 milliseconds", which might be a huge amount of time for a language-defined tick. For instance, an implementation of this specification in Esterel would be similar, since it would not rely on the tick notion of simultaneity either.

2.6. Summary

CÉU aims to offer a simpler semantics than Esterel with more determinism and fine-grained control over program execution. The following list summarizes the contributions of our design in this direction:

— *Event-triggered notion of time bound to the semantics of the language.* Event-driven programming is popular in many domains, such as server and GUI development. We believe that programmers are more familiar with dealing with events in isolation, which simplifies the reasoning about concurrency. In addition, the uniqueness of external events is a prerequisite for the concurrency checks of CÉU.
— *Deterministic intra-reaction execution and communication.* Determinism in CÉU does not depend on additional levels of static analysis. It encompasses the whole language, including memory accesses, system calls, and stack-based internal events.

Programmers can figure out which statement executes next, helping runtime analysis and debugging.

—*Static concurrency checks*. Although execution is deterministic, the CÉU compiler still advises about suspicious statements that can react concurrently to the same event.

—*Safe integration with* C. When dealing with concurrent system calls, programmers can provide annotations to reduce false positives in the static checks, or to force non commutative concurrent behavior. CÉU also requires finalization clauses to handle pointers representing resources.

—*First-class timers with dedicated syntax and automatic synchronization.* Given the omnipresence of timers in embedded systems, a dedicated syntax can simplify the development and readability of programs. Furthermore, automatic synchronization releases the programmer from the burden of adjusting timers in sequence and in parallel.

Our synchronous and deterministic approach also leads to some limitations as follows:

—*Execution is purely reactive as result of event-triggered reactions.* Since only event occurrences can start reactions, programs cannot execute proactively in the absence of events. In addition, `await` statements cannot awake in the same reaction they are reached.

—*Reactions must execute in bounded time due to the synchronous hypothesis.* As a synchronous language, CÉU requires CPU times for reactions to be negligible in comparison to the rate of incoming events.

—*Execution is sequential because of intra-reaction determinism.* The deterministic semantics of CÉU does not make implicit parallelization easy (to be discussed in Section 3.5).

Nonetheless, we advocate keeping a tractable synchronous reactive core with support for shared memory concurrency and deterministic execution. To deal with the limitations above, we recommend memory-isolated parallelizable asynchronous primitives as separate extensions to the synchronous core [Berry et al. 1993] (which are not in the scope of this paper).

3. IMPLEMENTATION

The compilation process of programs in CÉU is composed of three main phases: the *parsing phase* converts the source code in CÉU to an abstract syntax tree (*AST*); the *concurrency checks phase* detects inconsistencies in programs, such as unbounded loops and suspicious concurrent statements; the *code generation phase* converts the *AST* to standard *C* code and augments it with platform-dependent functionality (e.g., system calls) and the runtime of CÉU, compiling everything with *gcc* to generate the final binary.

In the subsections that follow, we discuss implementation details specific to CÉU: concurrency checks for determinism (Section 3.1), static memory allocation for data and trails (Sections 3.2 and 3.3), static scheduling and trail finalization (Section 3.4), single-threaded dispatching (Section 3.5), interaction with the environment (Section 3.6), and the *VM* back end (Section 3.7),

3.1. Concurrency Checks

The compile-time concurrency checks phase detects inconsistencies in CÉU programs. Here, we focus on the algorithm that detects suspicious concurrent statements, such as accesses to shared variables, as discussed in Section 2.3.

```
1   input void A, B;            1   Stmts I={boot} O={A}
2   var int y;                  2       Dcl_y I={boot} O={boot}
3   par/or do                   3       ParOr I={boot} O={A,B}
4      await A;                 4           Stmts I={boot} O={A}
5      y = 1;                   5               Await_A I={boot} O={A}
6   with                        6               Set_y I={A} O={A}
7      await B;                 7           Stmts I={boot} O={B}
8      y = 2;                   8               Await_B I={boot} O={B}
9   end                         9               Set_y I={B} O={B}
10  await A;                    10      Await_A I={A,B} O={A}
11  y = 3;                      11      Set_y I={A} O={A}
```

[a] A program in CÉU...   [b] ...with corresponding sets I and O.

Fig. 13. A program with a corresponding AST describing the sets I and O. The program is safe because accesses to y in parallel have no intersections for I.

For each node representing a statement in the program AST, we keep the set of input events I (*incoming*) that can start the execution of the node, and also the set of input events O (*outgoing*) that can terminate the node. As an example, for the single-statement program await A, we have $I = \{boot\}$ and $O = \{A\}$.

A node inherits the set I from its immediate parent and calculates O according to its type, as follows:

— Nodes that represent expressions, assignments, C calls, and declarations simply reproduce $O = I$, as they do not await;
— An await E statement, where E is an external input event, has $O = \{E\}$ (see also internal events below).
— A break statement has $O = \{\}$ as it escapes the innermost loop and never proceeds to the statement immediately following it (see also loop below);
— A *sequence node (;)* modifies each of its children to have $I_n = O_{n-1}$, except for $n = 1$ (which inherits I from the parent node). The set O for the whole node is copied from its last child, i.e., $O = O_n$.
— A loop node includes the output of its body on its own I ($I = I \cup O_{body}$), as the loop is also reached from its own body. The union of all O from nested break statements forms the set O for a loop.
— An if node has $O = O_{true} \cup O_{false}$, where true and false are the two if branches.
— A parallel composition may terminate from any of its branches, hence $O = O_1 \cup ... \cup O_n$.
— For internal events, an await awakes from any input that leads to any matching emit in a trail in parallel:
    — An await e has $O = I_{e1} \cup ... \cup I_{eN}$, where $e1...eN$ are emit e statements in trails in parallel.
    — An emit e terminates in the same reaction, having $O = I$.

With all sets calculated, we take all pairs of nodes that perform side effects and are in parallel branches, and compare their sets I for intersections. For each pair, if the intersection is not empty, we mark both nodes as suspicious.

The code in Figure 13.a has its corresponding AST and sets I and O in Figure 13.b. The assignments to y in parallel (ln. [a]:5,8) have an empty intersection of I (ln. [b]:6,9), hence, they do not conflict. Note that, although the accesses to y in sequence (ln. [a]:5,11) do have an intersection (ln. [b]:6,11), they are not in parallel branches and are also safe.

```
 1  input int A, B, C;
 2  do
 3      var int a = await A;
 4  end
 5  do
 6      var int b = await B;
 7  end
 8  par/and do
 9      await B;
10  with
11      await C;
12  end
```

```
 1  union {              // sequence
 2      int a;           //   do_1
 3      int b;           //   do_2
 4      struct {         //   par/and
 5          int _and_1: 1;
 6          int _and_2: 1;
 7      };
 8  } MEM ;
 9
10
11
12  .
```

[a] A program in CÉU...   [b] ...with corresponding memory layout

Fig. 14. A program with blocks in sequence and in parallel, with corresponding memory layout generated by the compiler.

3.2. Static Memory Layout

CÉU promotes a fine-grained use of trails: it is common to use trails that await a single event and terminate. For this reason, CÉU does not allocate per-trail stacks; instead, all data resides in fixed memory slots—this is true for the program variables as well as for temporary values and runtime flags. Memory for trails in parallel must coexist, while statements in sequence can reuse it. Translating this idea to *C* is straightforward [Kasten and Römer 2005]: memory for blocks in sequence are packed into a union, while blocks in parallel are packed into a struct. CÉU reserves a single static block of memory to hold all memory slots, whose size is the maximum the program uses at any given time. A position in the memory may hold different data (with variable sizes) during runtime. As an example, Figure 14 shows a program with its corresponding memory layout. The do-end blocks and par/and in sequence (ln. [a]:2–4,5–7,8–12) are packed in a union (ln. [b]:2,3,4–7), given that their variables cannot be in scope at the same time, e.g., a and b can safely share the same memory slot. The example also illustrates the presence of runtime flags (ln. [b]:4–7) related to the par/and termination (ln. [a]:8–12), which also reside in reusable slots in the static memory.

3.3. Static and Lightweight Trail Allocation

Each line of execution in CÉU needs to carry associated data, such as which event it is currently awaiting and which code to execute when it awakes. The compiler statically infers the maximum number of trails a program can have at the same time and creates a static vector to hold the runtime information about them. Like normal variables, trails that cannot be active at the same time share slots in the static memory vector.

At any given moment, a trail can be awaiting in one of the following states: INACTIVE, STACKED, FINALIZE, or in any of the events defined in the program:

```
enum {
    INACTIVE = 0,
    STACKED,
    FINALIZE,
    EVT_A,       // input void A;
    EVT_e,       // event int e;
    <...>        // other events
}
```

All terminated or not-yet-started trails stay in the INACTIVE state and are ignored by the scheduler. A STACKED trail holds an associated numeric stack level and can only execute when scheduler runtime drops to that level. A FINALIZE trail represents a pending finalization block which is scheduled only when its corresponding block goes out of scope. A trail waiting for an event stays in that event, also holding the minimum

sequence reaction number (*seqno*) in which it can awake (to respect *delayed awaits*). In concrete terms, a trail is represented by the following `struct`:

```c
struct trail_t {
    state_t evt;                    // awaiting event
    label_t lbl;                    // awaking execution label
    union {
        unsigned char seqno;        // if evt=EVT_*
        stack_t       stk;          // if evt=STACKED
    };
};
```

The field `evt` holds the state of the trail (or the event it is awaiting); the field `lbl` holds the entry point in the code to execute when the trail segment is scheduled; the third field depends on the `evt` field and may hold the `seqno` for an event, or the stack level `stk` for a `STACKED` trail.

The size of `state_t` depends on the number of events in an application; for an application with less than 253 events (plus the 3 states), one byte is enough. The size of `label_t` depends primarily on the number of `await` statements in the application—each `await` splits the code into two segments and requires a unique entry point in the code for its continuation. Additionally, split & join points for parallel compositions, `emit` continuations, and finalization blocks also require labels. The fields `seqno` requires only 2 bits because the scheduler adjusts them while traversing all trails. The size of `stack_t` depends on the maximum depth of nested emissions but is bounded by the maximum number of trails: in the worst case, a trail emits an event that awakes another trail, which emits an event that awakes another trail, and so on; the last trail cannot awake any other trail, because they are all blocked in the `STACKED` state.

In the context of embedded systems, the size of `trail_t` is typically only 3 bytes (1 byte for each field), imposing a negligible memory overhead even for trails that only await a single event and terminate. For instance, the *CTP* collection protocol ported to CÉU reaches eight simultaneous lines of execution but has a memory overhead of only 2% in comparison to the original single-threaded version in *C* [Sant'Anna et al. 2013].

3.4. Static Scheduling and Trail Finalization

In the final generated code in *C*, each trail segment label representing an entry point becomes a *switch case* with the associated code to execute. Figure 15 illustrates the generation process. For the program in [a], the compiler extracts the entry points and associated trails, e.g., the label `Awake_e` will execute on TRAIL-0 (ln. [a]:7 and [b]:3). For each yielding statement (e.g., `emit`, `await`, `par/and`, etc.), the compiler splits the trail into two segments with associated entry points. The entry points translate to an `enum` in the generated code (ln. [b]:1–10). The state of trails translates to a vector of type `trail_t` with the maximum number of simultaneous trails (ln. [b]:12–15). On initialization, TRAIL-0 is set to execute the `Main` entry point (ln. [b]:13), while all others are set to INACTIVE (ln. [b]:14).

The scheduler executes in two passes: in the *broadcast* pass, it sets all trails that are waiting for the current event to the state `STACKED` in the current numeric stack level; in the *dispatch* pass, it executes each trail that is `STACKED` to run in the current level, setting it immediately to INACTIVE. Since the trail space is flat and carries no information about the program hierarchy (ln. [b]:12–15), all trails (active or inactive) are traversed in all passes. We believe that this makes reaction times more predictable and improves worst-case reaction times (e.g., with all trails active). However, we did not investigate these assumptions.

During the dispatch pass, if a trail executes and emits an internal event, the scheduler increments the stack level and re-executes the two passes. After all trails are properly dispatched, the scheduler decrements the stack level and resumes the previ-

```
input void A;                    1    enum {                          1    void dispatch (trail_t* t) {
event void e;                    2      Main = 1,    // ln 3          2      switch (t->lbl) {
// TRAIL 0 — lbl Main            3      Awake_e,     // ln 7          3        case Main:
par/and do                       4      And_chk,     // ln 8,15       4          // activate TRAIL 1
  // TRAIL 0 — lbl Main          5      And_sub_2,   // ln 10         5          TRLS[1].evt = STACKED;
  await e;                       6      Awake_A_1,   // ln 12         6          TRLS[1].lbl = And_sub_2;
  // TRAIL 0 — lbl Awake_e       7      Emit_cont,   // ln 14         7          TRLS[1].stk = cur_stack;
  // TRAIL 0 — lbl And_chk       8      And_out,     // ln 17         8
with                             9      Awake_A_2    // ln 19         9          // code in the 1st trail
  // TRAIL 1 — lbl And_sub_2    10    };                             10          // await e;
  await A;                      11                                   11          TRLS[0].evt = EVT_e;
  // TRAIL 1 — lbl Awake_A_1    12    trail_t TRLS[2] = {            12          TRLS[0].lbl = Awake_e;
  emit e;                       13      { STACKED,  Main, 0 };       13          TRLS[0].seq = cur_seqno;
  // TRAIL 1 — lbl Emit_cont    14      { INACTIVE, 0,    0 };       14          break;
  // TRAIL 1 — lbl And_chk      15    };                             15
end                             16                                   16        case And_sub_2:
// TRAIL 0 — lbl And_out        17                                   17          // await A;
await A;                        18                                   18          TRLS[1].evt = EVT_A;
// TRAIL 0 — lbl Awake_A_2      19                                   19          TRLS[1].lbl = Awake_A_1;
                                20                                   20          TRLS[1].seq = cur_seqno;
                                21                                   21          break;
                                22                                   22
                                23                                   23        <...>  // other labels
                                24                                   24      }
.                               25    .                             25    }

        [a]                              [b]                                    [c]
```

Fig. 15. [a] Static allocation of trails: the comments identify the trail indexes inferred by the compiler; [b] Entry-point labels: each trail segment has an associated numeric identifier generated by the compiler. [c] Dispatch function: uses a `switch` to associate each segment identifier with the corresponding code to execute.

ous execution. For the boot reaction, the scheduler starts from the *dispatch* pass, given that the `Main` label is the only one that can be active at the stack level 0 (ln. [b]: 13).

The code in [c] dispatches a trail segment according to the current label to execute. For the first reaction, it executes the `Main` label in TRAIL-0 (ln. 3–14). When the `Main` label reaches the par/and (ln. [a]:4), it first stacks TRAIL-1 (ln. [c]:4–7) and then executes the `await e` (ln. [a]:6) in TRAIL-0 (ln. [c]:9–14), respecting lexical execution order. The dispatcher sets the running TRAIL-0 to await EVT_e on label `Awake_e`, and then halts with a `break`. Then, it switches to TRAIL-1 and executes label `And_sub_2` (ln. [c]:6,16–21), which sets TRAIL-1 to await EVT_A and also halts.

Regarding abortion and finalization, when a par/or terminates, the scheduler makes a *broadcast* pass for the FINALIZE event, but limited to the range of trails covered by the terminating par/or. Trails that do not match the FINALIZE are set to INACTIVE, as they have to be aborted. Given that trails in parallel are allocated in subsequent slots in the static vector TRLS (ln. [b]:12–15), this pass only aborts the desirable trails. The subsequent *dispatch* pass executes the finalization code properly. Escaping a `loop` that contains parallel compositions also triggers the same abortion process.

3.5. Single-Threaded Dispatching

The implementation of CÉU dispatches active trails sequentially in a single thread, taking no advantage of multi-core CPUs. This decision comes not only from the fact that CÉU targets constrained single-CPU embedded systems, but also because CÉU imposes deterministic execution for intra-reaction statements.

Note that, as discussed in Section 2.3, the concurrency checks of CÉU infer precisely trails that are concurrent and yet do not share resources. Hence, these non-conflicting trails could potentially execute with real parallelism in multiple cores. However, our experiments with multi-core execution are actually slower than single-core execution in the same system. Considering that we use CÉU primarily in control-dominated ap-

plications, this result is not surprising and also appears in related work [Yuan et al. 2011; Haribi 2012]. One reason is the overhead from continuous fork-and-rejoin in small reactions. Another reason is contention from excessive locality of data in stack-less trails sharing contiguous static memory.

If we consider data-intensive applications, multi-core implementations can offer considerable speedups. However, data-intensive computations do not typically require a disciplined step-wise execution and can actually execute in isolated asynchronous calls. Esterel provides a `task` primitive for this purpose [Berry 2000], while CÉU provides an equivalent `async/thread` primitive. Asynchronous execution is out of the scope of this paper.

In previous work [Sant'Anna et al. 2013], we evaluate our implementation by rewriting some components of the *TinyOS* code base[4] from *nesC* (a *C* variant) [Hill et al. 2000; Gay et al. 2003] to CÉU. TinyOS provides open-source industrial-level applications and serve as basis for a considerable amount of research in the Wireless Sensor Network academic community. The generated code has been tested in real hardware and also in congested networks simulated in sofware [Sant'Anna et al. 2013]. Figure 16 compares source size (*number of tokens*), binary size (*ROM*), and memory usage (*RAM*) for a number of standardized network protocols and a radio driver. The small overhead in resource usage shows that the gains in productivity and safety with CÉU make it a viable alternative to *C* in the context of constrained embedded systems. Single-threaded dispatching may not be suitable for hard real-time activities. We also measure how synchronous lengthy computations in *C* (e.g., hashing and compression) can block the scheduler and affect higher-priority activities such as a radio driver. In such cases, the system requires careful testing to avoid undersized hardware deployment. For instance, we currently do not perform any worst-case reaction times analysis [Boldt et al. 2008; Li et al. 2005].

In future work, we plan to perform an equivalent throughout analysis in comparison to industrial-quality code bases in Esterel. An Esterel implementation that compiles to sequential C code with similar techniques also generates "code less than 10% larger the original manually-sequenced code" for an industrial-size application [Weil et al. 2000].

3.6. Interaction with the Environment

As a reactive language, the execution of programs in CÉU is guided entirely by the occurrence of external input events. The binding for a specific platform (environment)

-----

[4]TinyOS repository: http://github.com/tinyos/tinyos-release/

Application	Language	tokens	Céu vs nesC	ROM	Céu vs nesC	RAM	Céu vs nesC
CTP	nesC	383	-23%	18896	9%	1295	2%
	Céu	295		20542		1319	
SRP	nesC	418	-30%	12266	5%	1252	-3%
	Céu	291		12836		1215	
DRIP	nesC	342	-25%	12708	8%	393	4%
	Céu	258		13726		407	
CC2420	nesC	519	-27%	10546	2%	283	3%
	Céu	380		10782		291	

Fig. 16.   Resource usage for CÉU and *nesC* in the domain of sensor networks.

```
1   implementation
2   {
3       #include "ceu_app.h"
4       #include "ceu_app.c"
5
6       event void Boot.booted () {
7           ceu_init();
8   #ifdef CEU_WCLOCKS
9           call Timer.startPeriodic(10);
10  #endif
11      }
12
13  #ifdef CEU_WCLOCKS
14      event void Timer.fired () {
15          ceu_wclock(10000);
16      }
17  #endif
18
19  #ifdef EVT_PHOTO_READDONE
20      event void Photo.readDone (int val) {
21          ceu_go(EVT_PHOTO_READDONE, &val);
22      }
23  #endif
24
25  #ifdef EVT_RADIO_SENDDONE
26      event void RadioSend.sendDone (message_t* msg) {
27          ceu_go(EVT_RADIO_SENDDONE, &msg);
28      }
29  #endif
30
31  #ifdef EVT_RADIO_RECEIVE
32      event message_t* RadioReceive.receive (message_t* msg) {
33          ceu_go(EVT_RADIO_RECEIVE, &msg);
34          return msg;
35      }
36  #endif
37
38      <...>   // other events
39  }
```

Fig. 17.   The *TinyOS* binding for CÉU. This platform-dependent template includes the *C* files generated from the original application in CÉU (ceu_app.h and ceu_app.c) for the *code generation phase*.

calls hook functions in the API of the CÉU runtime whenever an external event occurs. These calls must never interleave or parallelize execution in order to preserve the sequential/discrete notion of time in CÉU.

Figure 17 shows our binding for *TinyOS* [Hill et al. 2000], which maps system call-backs to input events in CÉU. The file ceu_app.h (ln. 3) contains all definitions for the compiled CÉU program, which are further queried through #ifdef's. The file ceu_app.c (ln. 4) contains the runtime of CÉU with the scheduler and dispatcher pointing to the labels defined in the program. The callback Boot.booted (ln. 6–11) is called by TinyOS on startup, so we initialize CÉU inside it (ln. 7). If the CÉU program uses timers, we also start a periodic timer (ln. 8–10) that triggers callback Timer.fired (ln. 13–17) every 10 milliseconds to advance the wall-clock time of CÉU (ln. 15)[5]. The remaining lines map pre-defined TinyOS events that can be used in CÉU programs, such as the light sensor (ln. 19–23) and the radio transceiver (ln. 25–36). The scheduler of TinyOS is already synchronous by default and always executes event handlers atomically, hence, the API calls to CÉU are properly serialized.

_____

[5]We also offer a mechanism to start the underlying timer on demand to avoid the "battery unfriendly" 10ms polling.

3.7. The Terra Virtual Machine

Terra is a system for programming wireless sensor network applications which uses CÉU as its scripting language [Branco et al. 2015]. Figure 18 shows the three basic elements of Terra: CÉU as the scripting language, a set of customized pre-built components, and the embedded virtual-machine engine which can disseminate and install bytecode images dynamically. This approach aims to combine the flexibility of remotely uploading code with the expressiveness and safety guarantees of CÉU.
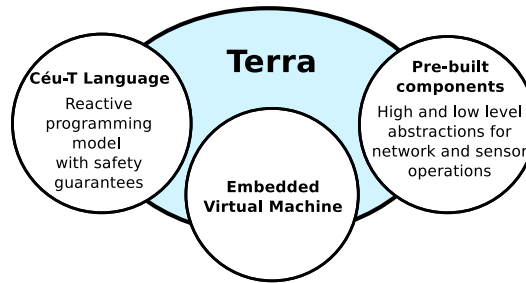
Fig. 18.   Terra programming system basic elements.

The main difference between the standard *C* back end and the Terra *VM* is the *code generation phase*, which here outputs assembly instructions for the *VM* (instead of statements in *C*). To reduce the memory footprint of applications, the *VM* includes special instructions for complex and recurrent operations from the runtime of CÉU, such as for handling events and trails.

In Terra, CÉU scripts cannot execute arbitrary *C* code, instead, they rely on pre-built components that can be customized for different application domains. In the domain of sensor networks, Terra already provides components organized in four areas: radio communication, group management, data aggregation, and local operations (e.g., access to sensors and actuators). When creating an instance of the *VM*, the programmer can choose whether or not to include each component, setting different abstraction boundaries for scripts. The generated *VM* has to be preloaded into the embedded devices before they are physically distributed.

The communication between scripts in CÉU and the components in the *VM* is mostly through events: scripts `emit` requests through `output` events and `await` answers through `input` events. Terra also provides system calls for initialization and configuration of components (e.g., *getters* and *setters*). Figure 19.a shows a CÉU interface with the available functionality for a customized *VM* (with temperature and radio components). Figure 19.b shows the associated bindings for output events (ln. 1–8), input events (ln. 10–14), and system calls (ln. 16–22). Note that all applications for the customized *VM* must comply with the same interface. In contrast, the template-based *C* back end (illustrated in Figure 17) allows applications to choose all possible combinations of functionalities from the underlying platform at compile time.

4. RELATED WORK

CÉU has a strong influence from Esterel but differ in the most fundamental aspect of the notion of time. In CÉU, instead of clock ticks, atomic external event occurrences that define time units. The event-driven approach of CÉU is widespread [Ousterhout 1996] and popular in many software communities, such as web frameworks (e.g., *jQuery* [Chaffer 2009] and *Node.js* [Tilkov and Vinoski 2010]), GUI toolkits (e.g.,

```
// Output events
output void REQUEST_TEMPERATURE;
output int  REQUEST_SEND;    // sends int value

// Input events
input int  TEMPERATURE_DONE; // recvs int value
input void SEND_DONE;

// System calls
function int getRadioID (void);




.
```

```
1  // Output events
2  void VM.out(int evt_id, void* args) {
3      switch (id){
4          case O_REQUEST_TEMPERATURE:
5              call TINYOS_TEMP.read();
6          <...>; // O_REQUEST_SEND
7      }
8  }
9
10 // Input events
11 event TINYOS_TEMP.done (int val) {
12     VM.enqueue(I_TEMPERATURE_DONE, &val);
13 }
14 <...> // TINYOS_SEND.done
15
16 // System calls
17 void VM.function(int id, void* params) {
18     switch (id) {
19         case F_GET_RADIO_ID:
20             VM.push(TINYOS_NODE_ID);
21     }
22 }
```

[a]   [b]

Fig. 19. [a] CÉU interface with customized *VM*. [b] The routine VM.out redirects all output events to the corresponding OS calls (ln. 1–8). Each *TinyOS* event callback calls VM.enqueue for the corresponding input event (ln 10–14). System calls use VM.push for immediate return values (ln. 16–22).

Tcl/Tk [Ousterhout 1991] and *Java Swing* [Eckstein et al. 1998]), and Games [Nystrom 2014]. As far as we know, CÉU is the first to encrust this event-triggered notion of time in the core of the language, which is a prerequisite for the concurrency checks that enable safe shared-memory concurrency.

In CÉU, internal events support stack-based micro reactions within external reactions, providing more fine-grained control for intra-reaction execution. Some variants of Statecharts [**?**] also distinguish internal from external events [**?**]. In Statemate, "reactions to external and internal events (...) can be sensed only after completion of the step" [Harel and Naamad 1996], implying queue-based execution. In Stateflow, "the receiving state (of the event) acts here as a function" [Hamon and Rushby 2007], which is similar to CÉU's stack-based execution. To avoid recursion, CÉU adopts delayed awaits, while in Statemate, "loops in the broadcasting of events (between states) are forbidden".

Like CÉU, many synchronous languages rely on deterministic scheduling to preserve intra-reaction determinism (*Reactive C* [Boussinot 1991], *Protothreads* [Dunkels et al. 2006], *SOL* [Karpinski and Cahill 2007], *SC* [von Hanxleden 2009], and *PRET-C* [Andalam et al. 2010]). CÉU also performs concurrency checks to detect trails that, when reordered, change the observable behavior of the program, i.e., trails that actually rely on deterministic scheduling. *SCCharts* [von Hanxleden et al. 2014a] proposes *sequentially constructive programs* that allow "shared variables to have multiple values per tick as long as these values are explicitly ordered by sequential statements" [von Hanxleden et al. 2014b]. This relaxes Esterel's restriction for *read-after-write* accesses only (e.g., a signal must be emitted to be present) and permits *write-after-read* accesses typical in imperative programs (e.g., test and set). Although this approach still restricts concurrent accesses, it does not impose strict deterministic scheduling such as in CÉU.

Esterel+Delay [Bourke and Sowmya 2009] is a non-intrusive extension to Esterel to program in terms of physical time with delay statements. The global transformation relies on a *platform statement* the specifies the available timers in the system (e.g., sample-driven or interrupt-driven). It then expands delay statements into existing Es-

terel statements in a way that the desired semantics can be realised in the platform. CÉU makes physical time a special input event that feeds the runtime with an associated time to elapse which is decremented from all awaiting trails. The compensation scheme of CÉU guarantees that trails awake in the correct order and that errors are not propagated on subsequent awaits. Interrupt-driven timers are also supported with a hook callback that the runtime calls whenever the program awaits an earlier timer.

Regarding resource management, Esterel supports a finalization mechanism to unconditionally execute a series of statements on abortion. In addition, CÉU also tracks pointers representing resources that cross *C* boundaries and forces the programmer to provide associated finalizers.

ReactiveML [Mandel and Pouzet 2005] and *URBI* [Baillie 2005] extend the synchronous model with dynamic lines of execution. The implementations use coroutines or CPS transformations and rely on heap allocation and/or garbage collection, diverging from our goals regarding resource efficiency and static bounds for memory and execution time. We discuss dynamic abstractions in CÉU in previous work [Sant'Anna et al. 2015].

Esterel has different compilation back ends that synthesizes to software and also to hardware circuits [Dayaratne et al. 2005; Edwards 2003]. Among the software-based approaches, *SAXO–RT* [Closse et al. 2002; Weil et al. 2000] is the closest to our implementation with respect to trail allocation and scheduling: the compiler slices programs into "control points" (analogous to our "entry points") and rearranges them into a directed acyclic graph. Then, it flattens the graph into sequential code in *C* suitable for static scheduling. Esterel compilation is more sophisticated since it involves solving data dependency relations to comply with its constructive semantics.

A number of virtual machines have been proposed for embedded systems. *Darjeeling* [Brouwers et al. 2008] and *TakaTuka* [Aslam et al. 2010] are complete *Java VMs* targeting constrained embedded systems with support for multithreading and garbage collection. Java has antagonistic design choices in comparison to CÉU: it does not impose static bounds on memory usage and execution time, and provides preemptive multithreading which requires synchronization primitives for accessing shared memory. Plummer et al. [Plummer et al. 2006] propose a Esterel-based *VM* with similar design choices to our work. To reduce code size, the *VM* has a specialized instruction set to deal with events and concurrency constructs that are particular to Esterel. However, the proposed *VM* is only a proof of concept, with no support for arithmetic operations, external system calls, or remote reprogramming.

5. CONCLUSION

We present the design and implementation of CÉU, a synchronous reactive language inspired by Esterel with event-driven semantics and more fine-grained control for intra-reaction execution. On the one hand, this approach is familiar to programmers in general, abstracting tick sampling with reactions to unique events from their application domain. On the other hand, this level of abstraction does not suit systems with hard-real time requirements in which interacting directly with a concrete notion of tick is more robust.

CÉU is a concurrency-safe language, employing static checks to ensure that the high degree of concurrency in embedded systems does not pose safety threats to applications. As a summary, the following safety properties hold for all programs that successfully compile in CÉU: time and memory-bounded reactions to the environment (except for system calls), no race conditions in shared memory, reliable abortion for activities handling resources, and automatic synchronization for timers. These properties are usually desirable in embedded applications and are guaranteed as preconditions in CÉU by design.

CÉU is a resource-efficient language suitable for constrained embedded systems. The reference implementation compiles to portable event-driven code in *C*, with no special requirements for OS threads or per-trail data stacks. The *VM* implementation uses the same front end and imposes no extra restrictions, being equally suitable for constrained systems.

CÉU is a practical language with expressive control constructs, such as lexically scoped parallel compositions, convenient first-class timers, and a unique stack-based mechanism for internal signalling. Programs interoperate seamlessly with *C*, and can take advantage of existing libraries, lowering the entry barrier for adoption. CÉU has an open source implementation and bindings for *TinyOS*, *Arduino*, and the *SDL* graphical library.[6]

For the past three years, we have been teaching CÉU to undergraduate and graduate students in courses on *distributed systems* and *reactive programming*. Our experience shows that students take advantage of the sequential-imperative style of CÉU and can implement non-trivial concurrent applications in a few weeks. More recently, a company specialized in embedded systems (not related to our research group) released a product based on CÉU.

REFERENCES

Andalam, S., Roop, P., and Girault, A. 2010. Predictable multithreading of embedded applications using PRET-C. In *Proceeding of MEMOCODE'10*. IEEE, 159–168.

Aslam, F., Fennell, L., Schindelhauer, C., Thiemann, P., Ernst, G., Haussmann, E., Rührup, S., and Uzmi, Z. A. 2010. Optimized Java binary and virtual machine for tiny motes. In *Proceedings of DCOSS'10*. Springer, 15–30.

Baillie, J.-C. 2005. Urbi: Towards a universal robotic low-level programming language. In *International Conference on Intelligent Robots and Systems*. IEEE, 820–825.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and Simone, R. D. 2003. The synchronous languages twelve years later. In *Proceedings of the IEEE*, Vol. 91. 64–83.

Berry, G. 1993. Preemption in Concurrent Systems.. In *FSTTCS (LNCS)*, Vol. 761. Springer, 72–93.

Berry, G. 1999. *The Constructive Semantics of Pure Esterel (draft version 3)*. Ecole des Mines de Paris and INRIA.

Berry, G. 2000. *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France. Version 5.10, Release 2.0.

Berry, G., Ramesh, S., and Shyamasundar, R. 1993. Communicating reactive processes. In *Proceedings of POPL'93*. ACM, 85–98.

Boldt, M., Traulsen, C., and von Hanxleden, R. 2008. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science* 203, 4 (2008), 65–79.

Bourke, T. and Sowmya, A. 2009. Delays in Esterel. *SYNCHRON09* (2009), 55.

Boussinot, F. 1991. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience* 21, 4 (1991), 401–428.

Boussinot, F. 1998. SugarCubes implementation of causality. (1998).

Boussinot, F. and De Simone, R. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.

Branco, A., Sant'anna, F., Ierusalimschy, R., Rodriguez, N., and Rossetto, S. 2015. Terra: Flexibility and Safety in Wireless Sensor Networks. *ACM Trans. Sen. Netw.* 11, 4, Article 59 (Sept. 2015), 27 pages. DOI:http://dx.doi.org/10.1145/2811267

Brouwers, N., Corke, P., and Langendoen, K. 2008. Darjeeling, a Java compatible virtual machine for microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. ACM, 18–23.

Chaffer, J. 2009. *Learning JQuery 1.3: Better Interaction and Web Development with Simple JavaScript Techniques*. Packt Publishing Ltd.

Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. 2002. Saxo–RT: Interpreting Esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science* 65, 5 (2002), 80–94.

---

[6]Website of CÉU: http://www.ceu-lang.org/

Dayaratne, M. W. S., Roop, P. S., and Salcic, Z. 2005. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of SLAP'05*.

de Moura, A. L. and Ierusalimschy, R. 2009. Revisiting coroutines. *ACM TOPLAS* 31, 2 (Feb. 2009), 6:1–6:31.

de Simone, R., Talpin, J.-P., and Potop-Butucaru, D. 2005. The Synchronous Hypothesis and Synchronous Languages. In *Embedded Systems Handbook*, R. Zurawski (Ed.).

Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*. ACM, 29–42.

Eckstein, R., Loy, M., and Wood, D. 1998. *Java swing*. O'Reilly & Associates, Inc.

Edwards, S. A. 1999. Compiling Esterel into sequential code. In *7th International Workshop on Hardware/-Software Codesign*. ACM, 147–151.

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

Edwards, S. A. 2005. Using and Compiling Esterel. MEMOCODE'05 Tutorial. (July 2005).

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of PLDI'03*. 1–11.

Halbwachs, N. 1994. *Synchronous programming of reactive systems*. Vol. 215. Springer Science & Business Media.

Hamon, G. and Rushby, J. 2007. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 447–456.

Harel, D. and Naamad, A. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5, 4 (1996), 293–333.

Haribi, W. 2012. Compiling Esterel for Multi-Core Execution. *Synchrone Sprachen* (2012), 45.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGPLAN Notices* 35 (November 2000), 93–104. Issue 11.

Karpinski, M. and Cahill, V. 2007. High-Level Application Development is Realistic for Wireless Sensor Networks. In *Proceedings of SECON'07*. 610–619.

Kasten, O. and Römer, K. 2005. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. In *Proceedings of IPSN '05*. 45–52.

Kopetz, H. 2011. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media.

Li, X., Lukoschus, J., Boldt, M., Harder, M., and von Hanxleden, R. 2005. An Esterel processor with full preemption support and its worst case reaction time analysis. In *Proceedings of CASES'05*. ACM, 225–236.

Mandel, L. and Pouzet, M. 2005. ReactiveML: a reactive extension to ML. In *Proceedings of PPDP'05*. ACM, 82–93.

Nystrom, R. 2014. *Game Programming Patterns*. Genever Benning.

Ousterhout, J. K. 1991. An X11 Toolkit Based on the Tcl Language.. In *USENIX Winter*. 105–116.

Ousterhout, J. K. 1996. Why Threads Are A Bad Idea (for most purposes). (January 1996).

Plummer, B., Khajanchi, M., and Edwards, S. A. 2006. An Esterel virtual machine for embedded systems. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*. Citeseer, Vienna, Austria.

Roop, P. S., Salcic, Z., and Dayaratne, M. 2004. Towards direct execution of Esterel programs on reactive processors. In *Proceedings of EMSOFT'04*. ACM, 240–248.

Sant'Anna, F. 2013. *Safe System-level Concurrency on Resource-Constrained Nodes with Céu*. Ph.D. Dissertation. PUC–Rio.

Sant'Anna, F., Rodriguez, N., and Ierusalimschy, R. 2012. *Céu: Embedded, Safe, and Reactive Programming*. Technical Report 12/12. PUC-Rio.

Sant'Anna, F., Rodriguez, N., and Ierusalimschy, R. 2013. Advanced Control Reactivity for Embedded Systems. Workshop on Reactivity, Events and Modularity (REM'13). (2013).

Sant'Anna, F., Rodriguez, N., and Ierusalimschy, R. 2015. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*.

Sant'Anna, F., Rodriguez, N., Ierusalimschy, R., Landsiedel, O., and Tsigas, P. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

Schneider, K. and Wenz, M. 2001. A new method for compiling schizophrenic synchronous programs. In *Proceedings of CASES'01*. ACM, 49–58.

Sentovich, E. M. 1997. Quick conservative causality analysis. In *System Synthesis, 1997. Proceedings., Tenth International Symposium on*. IEEE, 2–8.

Shiple, T. R., Berry, G., and Touati, H. 1996. Constructive analysis of cyclic circuits. In *European Design and Test Conference, 1996. ED&TC 96. Proceedings*. IEEE, 328–333.

Tardieu, O. and De Simone, R. 2004. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of MEMOCODE'04*. IEEE, 39–48.

Tilkov, S. and Vinoski, S. 2010. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 6 (2010), 80–83.

von Hanxleden, R. 2009. SyncCharts in C: a proposal for light-weight, deterministic concurrency. In *Proceedings EMSOFT'09*. ACM, 225–234.

von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., and O'Brien, O. 2014a. SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications: HW/SW-synthesis for a Conservative Extension of Synchronous Statecharts. In *Proceedings of PLDI'14 (PLDI'14)*. ACM, New York, NY, USA, 372–383. DOI:http://dx.doi.org/10.1145/2594291.2594310

von Hanxleden, R., Mendler, M., Aguado, J., Duderstadt, B., Fuhrmann, I., Motika, C., Mercer, S., O'brien, O., and Roop, P. 2014b. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 144 (July 2014), 26 pages. DOI:http://dx.doi.org/10.1145/2627350

Weil, D., Bertin, V., Closse, E., Poize, M., Venier, P., and Pulou, J. 2000. Efficient Compilation of ESTEREL for Real-time Embedded Systems. In *Proceedings of CASES'00*. ACM, New York, NY, USA, 2–8.

Yuan, S., Yoong, L. H., and Roop, P. S. 2011. Compiling Esterel for multi-core execution. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 727–735.

Yun, J.-H., Kim, C.-J., Kim, S., Choe, K.-M., and Han, T. 2013. Detection of harmful schizophrenic statements in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 3 (2013), 80.