# The Design, Semantics, and Implementation of CÉU: a Synchronous Reactive Language based on Esterel

Francisco Sant'Anna, Departamento de Informática, PUC–Rio
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 reactive language based on Esterel that targets constrained embedded platforms and ensures safe concurrency by handling threats at compile time. Based on the synchronous programming model, our design allows for a simple reasoning about concurrency that enables compile-time analysis and results in deterministic and memory-safe programs. We discuss the design of CÉU and propose a formal semantics for its particular control mechanisms, such as parallel compositions, finalization, and internal events. We also present two implementation back ends: one aiming for resource efficiency and interoperability with *C*, and another based on 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. Considering the control-based languages, Esterel [Boussinot and de Simone 1991] was the first to appear and influenced 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].

CÉU is another Esterel-based language targeting embedded and (soft) real-time systems with novel functionalities:

— Stack-based execution for internal events, which provide a limited form of coroutines.
— A static temporal analysis and deterministic execution semantics that allows programs to safely share memory.
— A finalization mechanism for safe abortion of lines of execution holding external resources.
— First-class synchronized timers.

In this work, we discuss the design of CÉU and propose a formal semantics for a small synchronous kernel that represents a subset of the language covering these new functionalities. We also present an implementation of CÉU with two back ends: one aiming for resource efficiency and interoperability with *C*, and another based on 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]

[1]Arduino: https://www.arduino.cc/en/Main/arduinoBoardUno
MICAz: http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf
(Both use the *ATmega328* microcontroller with 32 Kbytes of FLASH and 2 Kbytes of SRAM.)

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 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, a simple application that blinks three LEDs periodically occupies less than 100 bytes and can be completely transmitted in 4 radio messages. In contrast, the full image of a *C*-compiled version occupies more than 2 Kbytes [Branco et al. 2015].[2]

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 a formal semantics for the control primitives of CÉU. Section 4 presents the *C* and *VM* implementation back ends. Section 5 discusses other synchronous languages targeting embedded systems. Section 6 concludes the paper.

## 2. THE DESIGN OF CÉU

CÉU is a synchronous reactive language based on Esterel with support for multiple concurrent lines of execution known as *trails*. By reactive, we mean that programs are stimulated by the environment through input events that are broadcast to all awaiting trails. By synchronous, we mean that all trails at any given moment are either reacting to the current event or are awaiting another event; in other words, trails are never reacting to different events.

In the sections that follow, we discuss the main differences between CÉU and Esterel: queue-based external events and stack-based internal events (Section 2.1), shared-memory concurrency and determinism (Section 2.2), safe abortion with finalization (Section 2.3), and first-class timers (Section 2.4).

Regarding the similarities, Figure 1 shows side-by-side the implementations in Esterel and CÉU for the following control specification [Berry 2000]: *"Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs"*. The first phrase of the specification, awaiting and emitting the events, is translated almost identically in the two languages (ln. 4–9, in both implementations), given that Esterel's '||' and CÉU's `par/and` constructs are equivalent. For the second phrase, the reset behavior, the Esterel version uses a `abort-when` (ln. 3–10), which serves the same purpose of CÉU's `par/or` (ln. 3–12): the occurrence of event `R` aborts the awaiting statements in parallel and restarts the enclosing `loop`.

[2]The *VM* is preloaded in the devices with some *I*/*O* functionality, whereas *C*-based binaries are full images.

```
1  // ESTEREL
2  loop
3     abort
4        [
5           await A
6        ||
7           await B
8        ];
9        emit O
10    when R
11 end
```

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

Fig. 1. A control specification implemented in Esterel and CÉU: *"Emit `O` after `A` and `B`, resetting each `R`."*

CÉU has a strong imperative flavor, with explicit control flow through sequences, loops, parallels, and also assignments. Being designed for control-intensive applications, it provides support for concurrent lines of execution and broadcast communication through events. In addition, CÉU also employs the synchronous model, in which programs advance in a sequence of discrete reactions to external 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 [Potop-Butucaru et al. 2005]. The `await` statements are the only ones that halt a running reaction and allow a program to advance in this 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 Esterel and synchronous languages in general: computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay hypothesis, and cannot be elegantly implemented.

### 2.1. Queue-Based External Events and Stack-Based Internal Events

Esterel makes no semantic distinctions between internal and external signals, both having only the notion of either presence or absence during an entire reaction [Berry 1993]. In CÉU, external input events are unique within reactions and programs cannot emit them, resulting in an intrinsic queue-based handling. In contrast, programs can emit internal events but these follow a stack-based execution policy, similar to subroutine calls in typical programming languages. Figure 2 illustrates the use of internal signals (events) in Esterel and CÉU. In the version in Esterel, when `A` occurs, `B` is emitted (ln. 5–6) and both events become active, resulting in the invocation of `f()` and `g()` in no particular order. In the version in CÉU, the occurrence of `A` makes the program behave as follows (with the stack contents in italics):

(1) 1st trail awakes (ln. 5), emits `b`, and pauses.
*stack: [1st-trail]*
(2) 2nd trail awakes (ln. 9), calls `_g()`, and terminates.
*stack: [1st-trail]*
(3) 1st trail (on top of the stack) resumes, calls `_f()`, and terminates.
*stack: []*
(4) Both trails have terminated, so the `par/and` rejoins, and the program also terminates;

Internal events bring support for a limited form of subroutines, as depicted in Figure 3. The subroutine `inc` is defined as a loop (ln. 3–6) that continuously awaits its identifying event (ln. 4), incrementing the value passed as reference (ln. 5). A trail in parallel (ln. 8–11) invokes the subroutine in reaction to event `A` through an `emit` (ln. 10). 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.

```
1  // ESTEREL
2  input A;    // external
3  signal B;   // internal
4  [[
5      await A;
6      emit B;
7      call f();
8  ||
9      await B;
10     call g();
11 ]]
```

```
1  // CEU
2  input void A;  // external (in uppercase)
3  event void b;  // internal (in lowercase)
4  par/and do
5      await A;
6      emit b;
7      _f();
8  with
9      await b;
10     _g();
11 end
```

Fig. 2. Internal signals (events) in Esterel and CÉU: similar syntax, but different semantics.

```
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      await A;
10     emit inc => &v; // call 'inc'
11     _assert(v==2);  // after return
12 end
```

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

4, again). Only after this sequence the calling trail resumes and passes the assertion test (ln. 11).

On the one hand, this form of subroutines 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 subroutines 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 [Moura and Ierusalimschy 2009].

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 dependency. CÉU imposes a restriction 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. These *delayed awaits* avoid corner cases of instantaneous termination and re-execution of statements in the same reaction (known as *schizophrenic statements* [Tardieu and De Simone 2003; Yun et al. 2013]). The example that follows relies on this restriction to avoid infinite execution:

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

Both sides of the `par/or` have an `await` statement to avoid instantaneous termination (ln. 4,7). However, if the `emit` (ln. 6) could awake the `await` (ln. 4) in the same reaction that reaches them, 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 manually copying or transforming the code to execute on awake. From our experience, in some cases we need to execute a block of code periodically from internal event requests, *including the current reaction*, as illustrated in the left of Figure 4. In this case, moving the `await` to the end of the loop (ln. 10) makes the periodic code to also execute immediately (ln. 9), and then in reactions to each `emit` request (ln. 5). If the periodic `emit` depends on a condition, as illustrated in the right of Figure 4, the code becomes more intricate because we need a state variable (ln. 2) and also to copy the

```
event void e;
par do
    loop do
        <...>     // code that awaits
        emit e;   // periodic request
    end
with
    loop do
        <...>     // code to execute
        await e;  // await after
    end
end
```

```
1  event void e;
2  var bool should_execute = false;
3  par do
4      loop do
5          <...>         // code that awaits
6          if <...> then // some conditon
7              should_execute = true;
8              emit e;
9          end
10     end
11 with
12     loop do
13         if should_execute then
14             <...>     // code to execute
15         end
16         await e;
17     end
18 end
```

Fig. 4. Examples that circumvent the *delayed await* restriction by post-fixing the `await` inside the `loop` (in the left), and by copying the condition test (in the right).

condition test to the periodic code (ln. 13). 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 analysis to deal with schizophrenic statements.

### 2.2. Shared-Memory Concurrency and Determinism

Embedded applications make extensive use of global memory, such as for accessing resources through memory-mapped registers. Hence, an important goal of CÉU is to ensure a reliable behavior for programs with concurrent lines of execution sharing memory. Esterel is only deterministic with respect to reactive control: "the same sequence of inputs always produces the same sequence of outputs" [Berry 2000]. However, the execution order for operations with side-effects 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]. Therefore, Esterel forbids sharing memory between lines of execution: *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [Berry 2000].

Concurrency in CÉU is characterized when two or more trail segments in parallel execute during the same reaction. A trail segment is a sequence of statements followed by an `await` (or termination). In the program in the left of Figure 5, the two assignments to `x` (ln. 5,8) can never run concurrently, because each trail segment reacts to a different input event (ln. 4,7) and, according to the semantics of CÉU, cannot occur simultaneously. However, the program in the right is non-deterministic, because the two assignments to `y` (ln. 5,8) occur in the same reaction to input `A` (ln. 4,7).

```
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
```

Fig. 5. Shared-memory concurrency in CÉU: the code in the left is safe because the trails access `x` atomically in different reactions; the code in the right is unsafe because both trails access `y` in the same reaction.

```
// ESTEREL
input A;
[
    await A;
    call f1();
||
    await A;
    call f2();
];
```

```
// CEU
input void A;
par/and do
    await A;
    _f1();
with
    await A;
    _f2();
end
```

Fig. 6. In Esterel, the execution order between `f1` and `f2` is unspecified, whereas in CÉU, `_f1` executes before `_f2` due to deterministic scheduling based on lexical order.

CÉU performs a temporal analysis at compile time and detects concurrent accesses to shared variables, as follows: *if a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable, nor dereference a pointer of that variable type.* An analogous policy is applied for pointers *vs* variables and pointers *vs* pointers, as well as for system calls with side effects (e.g., `printf`).

Regardless of the temporal analysis of CÉU, when multiple trails are active during the same reaction, they are scheduled in the order they appear in the program source code. Therefore, even though the program in the right of Figure 5 is suspicious, the assignments to `y` are both atomic and deterministic, i.e., after the reaction to `A` terminates, the value of `y` is 4 (`(1+1)*2`). On the one hand, enforcing an execution order for concurrent operations may seen arbitrary and also precludes true parallelism. On the other hand, it provides a priority scheme for trails, and makes shared-memory concurrency more tractable. For constrained embedded development, we believe that deterministic shared-memory concurrency is beneficial, given the extensive use of memory mapped ports for $I/O$ and the lack of hardware support for real parallelism. Other synchronous embedded languages, such as *SOL* [Karpinski and Cahill 2007] and *PRET-C* [Andalam et al. 2010], made a similar design choice.

Figure 6 compares the two syntactically equivalent code fragments in Esterel and CÉU to summarize the semantic difference regarding (non-)determinism. Even though the program in CÉU executes deterministically, the compiler still issues a warning, because an apparently innocuous reordering of trails modifies the semantics of the program. Note that in Esterel multiple external events can coexist in the same reaction, which disallows a similar temporal analysis.

### 2.3. Safe Abortion with Finalization

The introductory example of Figure 1 illustrates how synchronous languages can abort awaiting lines of execution without tweaking them with synchronization primitives. In contrast, traditional (asynchronous) multi-threaded languages cannot express thread termination safely [Berry 1993; ORACLE 2011]. Still, handling abortion when dealing with external resources is challenging because they are not subject to the same synchronous execution discipline.

To illustrate threats related to abortion, consider the unsafe example in the left of Figure 7, which does not compile in CÉU. It describes the state machine of a data collection protocol for sensor networks [Gnawali et al. 2009; Sant'Anna et al. 2013]. The input and output events represent the external interface of the protocol (ln. 1–2). The protocol has to transmit a packet every minute (with `SEND_ENQUEUE`), unless it receives a `RETRANSMIT` request to immediately re-transmit it, or a `STOP` request to terminate. The protocol is composed of two main trails: one simply monitors the stopping event (ln. 4); the other periodically transmits the packet (ln. 6–19). The periodic transmission is a loop that starts two other trails (ln. 7–18): one handles the immediate retransmission request (ln. 8); the other transmits the packet and waits for a confirmation (ln. 10–

```
1  input void STOP, RETRANSMIT, SEND_ACK;
2  output _pkt_t* SEND_ENQUEUE, SEND_CANCEL;
3  par/or do
4      await STOP;
5  with
6      loop do
7          par/or do
8              await RETRANSMIT;
9          with
10             par/and do
11                 await 1min;
12             with
13                 var _pkt_t buffer;
14                 <fill-buffer-info>
15                 emit SEND_ENQUEUE => &buffer;
16                 await SEND_ACK;
17             end
18         end
19     end
20 end
```

```
12 <...>
13     var _pkt_t buffer;
14     <fill-buffer-info>
15     finalize
16         emit SEND_ENQUEUE => &buffer;
17     with
18         emit SEND_CANCEL => &buffer;
19     end
20     await SEND_ACK;
21 <...>
```

Fig. 7. The unsafe network protocol in the left, which does not compile, is extended with a finalization clause in the right to handle abortion properly.

17). The actual transmission (ln. 13–16) is enclosed with a `par/and` that takes at least one minute before looping (ln. 11), in accordance with the specification. Note that the `emit SEND_ENQUEUE` (ln. 15) is *asynchronous*, handing to the radio driver a pointer to the lexically-scoped packet (ln. 13). The driver makes the transmission in the background, holding the packet until it signals the application with the `SEND_ACK` to acknowledge completion (ln. 16). At any time, the client may request a retransmission (ln. 8), which terminates the `par/or` (ln. 7), aborts the ongoing transmission (ln. 15, if not idle), and restarts the loop (ln. 6). The client may also request to stop the whole protocol at any time (ln. 4). Therefore, if the sending trail is aborted by the `STOP` or `RETRANSMIT` requests, the packet buffer goes out of scope (ln. 13), leaving behind a *dangling pointer* in the radio driver, which will possibly transmit corrupted data.

The unsafe example in the left of Figure 7 does not compile because CÉU tracks the interaction of `par/or` compositions with local variables and stateful output events calls in order to preserve safe abortion of trails. In fact, CÉU enforces the programmer to write a *finalization* clause to accompany the output request. The code in the right of Figure 7 properly cancels the packet transmission when the block of `buffer` goes out of scope, i.e., the finalization clause (after the `with`, ln. 18) executes automatically on external abortion.[3]

### 2.4. First-Class Timers

Activities that involve reactions to *wall-clock time*[4] appear in typical patterns of embedded development, such as timeout watchdogs and sensor samplings. However, support for wall-clock time is somewhat low-level in existing languages, usually through timer callbacks or "sleep" blocking calls. Furthermore, in any concrete timer implementation, a requested timeout does not expire precisely without delays, a fact that is usually ignored in the development process. We define the difference between the requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in sequence (or

[3]Note that the compiler only enforces the programmer to write the finalization clause, but cannot check if it actually handles the resource properly.
[4]By wall-clock time we mean the passage of time from the real world, measured in hours, minutes, etc.

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

```
par/or do
    await 10ms;
    <...>         // any non—awaiting sequence
    await  1ms;
    v = 1;
with
    await 12ms;
    v = 2;
end
```

Fig. 8. First-class timers in CÉU.

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. For the example in the left of Figure 8, suppose that after the first `await` request, the underlying system gets busy and takes 15ms to check for expiring awaits. The CÉU scheduler will notice that the `await 10ms` has not only already expired, but is delayed with `delta=5ms`. Then, the awaiting trail awakes, sets `v=1`, and invokes `await 1ms`. As the current delta is higher than the requested timeout (i.e. $5ms > 1ms$), the trail is rescheduled for execution, now with `delta=4ms`.

CÉU also takes into account the fact that time is a physical quantity that can be added and compared. For instance, for the program in the right of Figure 8, although the scheduler cannot guarantee that the first trail terminates exactly in 11ms, 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 is guaranteed to terminate before the second trail, 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.

## 3. FORMAL SEMANTICS

In this section, we introduce a reduced syntax of CÉU and propose an operational semantics to formally describe the language. We describe a small synchronous kernel with broadcast communication highlighting the peculiarities of CÉU, in particular the stack-based execution for internal events. For the sake of simplicity, we focus on the control aspects of the language, leaving out side effects and *C* calls (which behave like in conventional imperative languages).

### 3.1. Abstract Syntax

Figure 9 shows the BNF-like syntax for a subset of CÉU that is sufficient to describe all semantic peculiarities of the language. Except for $fin$ and all semantic-derived expressions (i.e., $awating$, $emitting$, and $p \ @ \ loop; p$), which are discussed further, all other expressions are equivalent to their counterparts in the concrete language.

The $mem(id)$ primitive represents all accesses, assignments, *C* function calls, and output events that affect a memory location identified by $id$. As the challenging parts of CÉU reside on its control structures, we are not concerned here with a precise semantics for side effects, but only with their occurrences in programs. The special notation $nop$ is used to represent an innocuous $mem$ expression (it can be thought as a synonym for $mem(\epsilon)$, where $\epsilon$ is an unused identifier). Note that $mem$ and $await$/$emit$ expressions do not share identifiers, i.e., an identifier is either a variable or an event.

```
                                  // primary expressions
p ::= mem(id)                    (any memory access to 'id')
    | await(id)                  (await event 'id')
    | emit(id)                   (emit event 'id')
    | break                      (loop escape)
                                  // compound expressions
    | if mem(id) then p else p   (conditional)
    | p ; p                      (sequence)
    | loop p                     (repetition)
    | p and p                    (par/and)
    | p or p                     (par/or)
    | fin p                      (finalization)
                                  // derived by semantic rules
    | awaiting(id,n)             (awaiting 'id' since sequence number 'n')
    | emitting(n)                (emitting on stack level 'n')
    | p @ loop p                 (unwinded loop)
```

Fig. 9. Reduced syntax of CÉU.

### 3.2. Operational Semantics

The core of our semantics is a relation that, given a sequence number $n$ identifying the current reaction, maps a program $p$ and a stack of events $S$ in a single step to a modified program and stack:

$$\langle S, p \rangle \underset{n}{\longrightarrow} \langle S', p' \rangle \qquad \textbf{(relation-inner)}$$

where

$$S, S' \in id^* \qquad (sequence\ of\ event\ identifiers : [id_{top}, ..., id_{bottom}])$$
$$p, p' \in P \qquad (program\ as\ described\ in\ Figure\ 9)$$
$$n \in \mathbb{N} \qquad (unique\ identifier\ for\ the\ reaction)$$

At the beginning of a reaction, the stack is initialized with the occurring external event $ext$ ($S = [ext]$), but $emit$ expressions can push new events on top of it (we discuss how they are popped further). The ever-increasing sequence number $n$ prevents that awaiting expressions awake in the same reaction they are reached (the *delayed awaits* as explained in Section 2.1).

We describe this relation with a set of small-step structural semantics rules, which are built in such a way that at most one transition is possible at any time, resulting in deterministic reaction chains. The transition rules for the primary expressions are as follows:

$$\langle S,\ await(id) \rangle \underset{n}{\longrightarrow} \langle S,\ awaiting(id, n+1) \rangle \qquad \textbf{(await)}$$

$$\langle id : S,\ awaiting(id, m) \rangle \underset{n}{\longrightarrow} \langle id : S,\ nop \rangle, \quad m <= n \qquad \textbf{(awake)}$$

$$\langle S,\ emit(id) \rangle \underset{n}{\longrightarrow} \langle id : S,\ emitting(|S|) \rangle \qquad \textbf{(emit)}$$

$$\langle S,\ emitting(|S|) \rangle \underset{n}{\longrightarrow} \langle S,\ nop \rangle \qquad \textbf{(pop)}$$

An $await$ is simply transformed into an $awaiting$ (rule **await**) as an artifice to remember the external sequence number $n+1$ it can awake: an $awaiting$ can only transit to a $nop$ (rule **awake**) if its referred event $id$ matches the top of the stack and it was reached in a previous reaction (i.e., sequence number $m <= n$). An $emit$ transits to an $emitting$ holding the current stack level ($|S|$ stands for the stack length), and pushing the referred event on the stack (rule **emit**). With the new stack level $|S| + 1$, the $emitting(|S|)$ itself cannot transit, as rule **pop** expects its parameter to match the current stack level. This trick provides the desired stack-based semantics for internal events.

Proceeding to compound expressions, the rules for conditionals and sequences are straightforward:

$$\frac{val(id,n) \neq 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, p\rangle} \quad \textbf{(if-true)}$$

$$\frac{val(id,n) = 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle} \quad \textbf{(if-false)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ ;\ q)\rangle} \quad \textbf{(seq-adv)}$$

$$\langle S, (mem(id)\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle \quad \textbf{(seq-nop)}$$

$$\langle S, (break\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, break\rangle \quad \textbf{(seq-brk)}$$

Given that our semantics focuses on control, rules **if-true** and **if-false** are the only to query $mem$ expressions. The "magical" function $val$ receives a memory identifier and the current reaction sequence number, returning the current memory value. Although the value here is arbitrary, it is unique in a reaction, because a given expression can execute only once within it (remember that $loops$ must contain $awaits$ which, from rule **await**, cannot awake in the same reaction they are reached).

The rules for loops are analogous to sequences, but use '`@`' as separators to properly bind breaks to their enclosing loops:

$$\langle S, (loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, (p\ @\ loop\ p)\rangle \quad \textbf{(loop-expd)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ @\ loop\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ @\ loop\ q)\rangle} \quad \textbf{(loop-adv)}$$

$$\langle S, (mem(id)\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, loop\ p\rangle \quad \textbf{(loop-nop)}$$

$$\langle S, (break\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, nop\rangle \quad \textbf{(loop-brk)}$$

When a program encounters a $loop$, it first expands its body in sequence with itself (rule **loop-expd**). Rules **loop-adv** and **loop-nop** are similar to rules **seq-adv** and **seq-nop**, advancing the loop until they reach a $mem(id)$. However, what follows the

$$
\begin{aligned}
isBlocked(n, a:S, awaiting(b,m)) &= (a \neq b \ \vee \ m > n)\\
isBlocked(n, S, emitting(s)) &= (|S| \neq s)\\
isBlocked(n, S, (p \ ; \ q)) &= isBlocked(n, S, p)\\
isBlocked(n, S, (p \ @ \ loop \ q)) &= isBlocked(n, S, p)\\
isBlocked(n, S, (p \ and \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q)\\
isBlocked(n, S, (p \ or \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q)\\
isBlocked(n, S, \_) &= false \ \ (mem, await,\\
&\qquad\qquad emit, break, if, loop)
\end{aligned}
$$

Fig. 10. The recursive predicate $isBlocked$ is true only if all branches in parallel are hanged in $awaiting$ or $emitting$ expressions that cannot transit.

loop is the loop itself (rule **loop-nop**). Note that if we used '`;`' as a separator in loops, rules **loop-brk** and **seq-brk** would conflict. Rule **loop-brk** escapes the enclosing loop, transforming everything into a $nop$.

Proceeding to parallel compositions, the semantic rules for $and$ and $or$ always force transitions on their left branches $p$ to occur before their right branches $q$:

$$
\frac{\langle S, p \rangle \ \underset{n}{\longrightarrow} \ \langle S', p' \rangle}{\langle S, (p \ and \ q) \rangle \ \underset{n}{\longrightarrow} \ \langle S', (p' \ and \ q) \rangle} \quad \textbf{(and-adv1)}
$$

$$
\frac{isBlocked(n, S, p) \ , \ \ \langle S, q \rangle \ \underset{n}{\longrightarrow} \ \langle S', q' \rangle}{\langle S, (p \ and \ q) \rangle \ \underset{n}{\longrightarrow} \ \langle S', (p \ and \ q') \rangle} \quad \textbf{(and-adv2)}
$$

$$
\frac{\langle S, p \rangle \ \underset{n}{\longrightarrow} \ \langle S', p' \rangle}{\langle S, (p \ or \ q) \rangle \ \underset{n}{\longrightarrow} \ \langle S', (p' \ or \ q) \rangle} \quad \textbf{(or-adv1)}
$$

$$
\frac{isBlocked(n, S, p) \ , \ \ \langle S, q \rangle \ \underset{n}{\longrightarrow} \ \langle S', q' \rangle}{\langle S(p \ or \ q) \rangle \ \underset{n}{\longrightarrow} \ \langle S', (p \ or \ q') \rangle} \quad \textbf{(or-adv2)}
$$

The deterministic behavior of the semantics relies on the *isBlocked* predicate, which is defined in Figure 10 and used in rules **and-adv2** and **or-adv2**. These rules require the left branch $p$ to be blocked in order to allow the right transition from $q$ to $q'$. Basically, the *isBlocked* predicate determines that an expression becomes blocked when all of its trails in parallel hang in $awaiting$ and $emitting$ expressions that cannot advance.

For a parallel $and$, if one of the sides terminates, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**, as follows). For a parallel $or$, if one of the sides terminates, the whole composition terminates, also applying the $clear$

function to properly finalize the aborted side (rules **or-nop1** and **or-nop2**):

$$\langle S, (mem(id)\ and\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle \qquad \textbf{(and-nop1)}$$

$$\langle S, (p\ and\ mem(id))\rangle \underset{n}{\longrightarrow} \langle S, p\rangle \qquad \textbf{(and-nop2)}$$

$$\langle S, (mem(id)\ or\ q)\rangle \underset{n}{\longrightarrow} \langle S, clear(q)\rangle \quad \textbf{(or-nop1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ or\ mem(id))\rangle \underset{n}{\longrightarrow} \langle S, clear(p)\rangle} \quad \textbf{(or-nop2)}$$

The $clear$ function, defined in Figure 11, concatenates all active $fin$ bodies of the side being aborted, so that they execute before the composition rejoins. Note that there are no transition rules for $fin$ expressions. This is because once reached, a $fin$ expression halts and will only execute when it is aborted by a trail in parallel and is expanded by the $clear$ function. In Section 3.3.3, we show how to map a finalization block in the concrete language to a $fin$ in the formal semantics. Note that there is a syntactic restriction that a $fin$ body can only contain $mem$ expressions, i.e., they are guaranteed to execute entirely within a reaction.

Finally, a $break$ in one of the sides in parallel escapes the closest enclosing $loop$, properly aborting the other side by applying the $clear$ function:

$$\langle S, (break\ and\ q)\rangle \underset{n}{\longrightarrow} \langle S, (clear(q)\ ;\ break)\rangle \quad \textbf{(and-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ and\ break)\rangle \underset{n}{\longrightarrow} \langle S, (clear(p)\ ;\ break)\rangle} \quad \textbf{(and-brk2)}$$

$$\langle S, (break\ or\ q) \underset{n}{\longrightarrow} \langle S, (clear(q)\ ;\ break)\rangle \quad \textbf{(or-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ or\ break)\rangle \underset{n}{\longrightarrow} \langle S, (clear(p)\ ;\ break)\rangle} \quad \textbf{(or-brk2)}$$

A reaction eventually blocks in $awaiting$ and $emitting$ expressions in parallel trails. If all trails hangs only in $awaiting$ expressions, it means that the program cannot

$$\begin{aligned}
clear(fin\ p) &= p \\
clear(p\ ;\ q) &= clear(p) \\
clear(p\ @\ loop\ q)) &= clear(p) \\
clear(p\ and\ q) &= clear(p)\ ;\ clear(q) \\
clear(p\ or\ q) &= clear(p)\ ;\ clear(q) \\
clear(_) &= nop
\end{aligned}$$

Fig. 11. The function $clear$ extracts $fin$ expressions in parallel and put their bodies in sequence.

advance in the current reaction. However, $emitting$ expressions are pending in lower stack indexes and should eventualy resume in the ongoing reaction (see rule **pop**). Therefore, we define another relation that behaves as **relation-inner** (presented above) if the program can advance, and, otherwise, pops the stack:

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, p\rangle \underset{n}{\Longrightarrow} \langle S', p'\rangle} \qquad \frac{isBlocked(n,\ s:S,\ p)}{\langle s:S, p\rangle \underset{n}{\Longrightarrow} \langle S, p\rangle} \qquad \textbf{(relation-outer)}$$

To describe a *reaction* in CÉU, i.e., how a program behaves in reaction to a single external event, we use the reflexive transitive closure of **relation-outer**:

$$\langle S, p\rangle \overset{*}{\underset{n}{\Longrightarrow}} \langle S', p'\rangle$$

Finally, to describe the complete execution of a program, we trigger multiple "invocations" of reaction chains in sequence:

$$\langle [e1], p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle$$
$$\langle [e2], p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle$$
$$\langle [e3], p''\rangle \overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle$$
$$\ldots$$

Each invocation starts with an external event at the top of the stack and finishes with a modified program and an empty stack. After each invocation, we increment the sequence number.

### 3.3. Concrete Language Mapping

Most statements from CÉU ("concrete CÉU") map directly to those presented in the reduced syntax of Figure 9 ("abstract CÉU"). For instance, the `if` in the concrete language behaves exactly like the $if$ in the formal semantics. However, there are some significant mismatches between the concrete and abstract CÉU, and we (informally) propose appropriate mappings in this section. Again, we are not considering side-effects, which are all mapped to the $mem$ semantic construct.

*3.3.1. await and emit.* The concrete `await` and `emit` primitives support communication of values between them. In the two-step translation of Figure 12, we start with the concrete program in CÉU, which communicates the value 1 between the `emit` and `await` in parallel (left-most code). In the intermediate translation, we include the shared variable `e_` to hold the value being communicated between the two trails in order to simplify the `emit`. Finally, we convert the program into the equivalent in the abstract syntax, translating side-effect statements into $mem$ expressions. External events require a similar translation, i.e., each external event has a corresponding variable that is explicitly set by the environment before each reaction.

*3.3.2. First-class Timers.* To encompass first-class timers, we introduce a special external event `DT` that is intercalated with each other event occurrence in an application (e.g. *e1, e2*):

```
par/or do
  <...>
  emit e => 1;
with
  v = await e;
  _printf("%d\n",v);
end
```

```
par/or do
  <...>
  e_ = 1;
  emit e;
with
  await e;
  v = e_;
  _printf("%d\n",v);
end
```

```
<...> ; mem ; emit(e)
        or
await(e) ; mem ; mem
```

Fig. 12. Two-step translation from concrete to abstract `emit` and `await` expressions. The concrete code in the left communicates the value `1` from the `emit` to the `await`. The abstract code uses a shared variable to hold the value.

$$\langle [DT], p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle$$
$$\langle [e1], p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle$$
$$\langle [DT], p''\rangle \overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle$$
$$\langle [e2], p'''\rangle \overset{*}{\underset{4}{\Longrightarrow}} \langle [], p''''\rangle$$
$$\ldots$$

The event `DT` has an associated variable `DT_` carrying the wall-clock time elapsed between two occurrences in sequence, as depicted by the two-step translation of Figure 13. In the concrete program in the left, the variable `dt` holds the residual delta time (as described in Section 2.4) after awaking from the timer. In the first step of the translation, we expand the `await 10ms` to a `loop` that decrements the elapsed number of microseconds for each occurrence of `DT`. When the variable `tot` reaches zero, we escape the `loop` setting the variable `dt` to contain the appropriate delta. In the last step, we convert the program to the abstract syntax.

*3.3.3. Finalization Blocks.* The biggest mismatch between concrete and abstract CÉU is regarding the `finalize` blocks, which require more complex modifications in the program for a proper mapping using $fin$ expressions. In the three-step translation of Figure 14, we start with a concrete program (`CODE-1`) that uses a `finalize` to safely `_release` the reference to `ptr` kept after the call to `_hold`. In the translation, we first need to catch the outermost `do-end` termination to run the finalization code. For this, we translate the block into a `par/or` (`CODE-2`) with the original body in parallel with a $fin$ expression to run the finalization code. Note that the $fin$ has no transition rules in the semantics, keeping the `par/or` alive. This way, the $fin$ body only executes when the `par/or` terminates either normally (after the `await B`), or aborted from an outer composition. However, the $fin$ still (incorrectly) executes even if the call to `_hold` is not

```
dt = await 10ms;
```

```
var int tot = 10000; // 10ms
loop do
    await DT;
    tot = tot - DT_;
    if tot <= 0 then
        dt = -tot;
        break;
    end
end
```

```
mem;
loop(
    await(DT);
    mem;
    if mem then
        mem;
        break
    else
        nop
)
```

Fig. 13. Two-step translation from concrete to abstract timer.

```
do
  var int* ptr = <...>;
  await A;
  finalize
    _hold(ptr);
  with
    _release(ptr);
  end
  await B;
end



// CODE-1
```

```
par/or do
  var int* ptr = <...>;
  await A;
  _hold(ptr);
  await B;
with
  { fin
      _release(ptr); }
end





// CODE-2
```

```
f_ = 0;
par/or do
  var int* ptr = <...>;
  await A;
  _hold(ptr);
  f_ = 1;
  await B;
with
  { fin
      if f_ then
        _release(ptr);
      end }
end

// CODE-3
```

```
mem;
(
   mem;
   await(A);
   mem;
   mem;
   await(B);
or
   fin
     if mem then
       mem
     else
       nop
)
// CODE-4
```

Fig. 14. Three-step translation from concrete to abstract finalization.

reached in the body due to an abort before awaking from the `await A`. To deal with this issue, for each $fin$ we need a corresponding flag to keep track of code that needs to be finalized (CODE-3). The flag is initially set to false, avoiding the finalization code to execute. Only after the call to `_hold` that we set the flag to true and enable the $fin$ body to execute. The complete translation substitutes the side-effect operations with $mem$ expressions (CODE-4).

## 4. IMPLEMENTATION

The compilation process of a program in CÉU is composed of three main phases, as illustrated in Figure 15:

*Parsing.* The parser of CÉU is written in *LPeg* [Ierusalimschy 2009], a pattern matching library that also recognize grammars, making it possible to write the tokenizer and grammar with the same tool. The source code is then converted to an *abstract syntax tree (AST)* to be used in further phases. This phase may be aborted due to syntax errors in the CÉU source file.

*Temporal Analysis.* This phase detects inconsistencies in CÉU programs, such as unbounded loops, suspicious accesses to shared memory, and also performs "classical" semantic analysis, such as building a symbol table for checking variable declarations. For the (standard) *C* back end, this phase outputs code in *C*. Some type checking is delayed to the last phase to take advantage of `gcc`'s error handling. Therefore, we annotate the *C* output file with `#line` pragmas matching the original file in CÉU.

*Final Generation.* The final phase packs the generated *C* file with the CÉU runtime and platform-dependent functionality, compiling them with *gcc* and generating the final binary. The CÉU runtime comprehends the scheduler, timer management, and the external *C* API. The platform files include libraries for I/O and bindings to invoke the CÉU scheduler on external events.

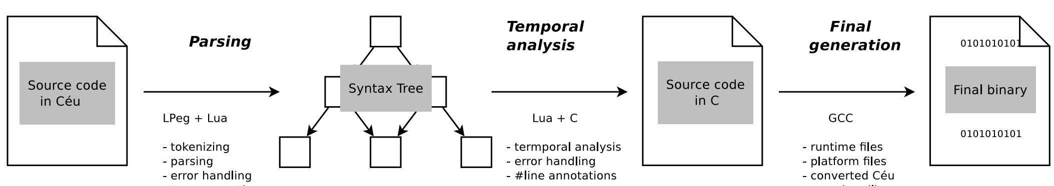


Fig. 15. Compilation process: from the source code in CÉU to the final binary.

```
1  input void A, B;
2  var int y;
3  par/or do
4    await A;
5    y = 1;
6  with
7    await B;
8    y = 2;
9  end
10 await A;
11 y = 3;
```

```
1  Stmts I={.} O={A}
2      Dcl_y I={.} O={.}
3      ParOr I={.} O={A,B}
4          Stmts I={.} O={A}
5              Await_A I={.} O={A}
6              Set_y I={A} O={A}
7          Stmts I={.} O={B}
8              Await_B I={.} O={B}
9              Set_y I={B} O={B}
10     Await_A I={A,B} O={A}
11     Set_y I={A} O={A}
```

Fig. 16. 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 $I$.

### 4.1. Temporal Analysis for Shared-Memory Concurrency

The compile-time *temporal analysis* phase detects inconsistencies in CÉU programs. Here, we focus on the algorithm that detects suspicious access to shared variables, as discussed in Section 2.2.

For each node representing a statement in the program AST, we keep the set of events $I$ (for *incoming*) that can lead to the execution of the node, and also the set of events $O$ (for *outgoing*) that can terminate the node.

A node inherits the set $I$ from its direct parent and calculates $O$ according to its type:

— Nodes that represent expressions, assignments, *C* calls, and declarations simply reproduce $O = I$, as they do not await;
— An `await e` statement has $O = \{e\}$.
— A `break` statement has $O = \{\}$ as it escapes the innermost `loop` and never terminates, i.e., never proceeds to the statement immediately following it (see also `loop` below);
— A *sequence node (;)* modifies each of its children to have $I_n = O_{n-1}$. The first child inherits $I$ from its parent node, and the set $O$ for the sequence node is copied from its last child, i.e., $O = O_n$.
— A `loop` node includes its body's $O$ on its own $I$ ($I = I \cup O_{body}$), as the loop is also reached from its own body. The union of all `break` statements' $O$ forms the set $O$ for a `loop`.
— An `if` node has $O = O_{true} \cup O_{false}$.
— A parallel composition may terminate from any of its branches, hence $O = O_1 \cup ... \cup O_n$.

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

The example in the left of Figure 16 has a corresponding $AST$, in the right of the figure, with the sets $I$ and $O$ for each node. The event . (dot) represents the "boot" reaction. The assignments to `y` in parallel (ln. 5,8 in the code) have an empty intersection of $I$ (ln. 6,9 in the AST), hence, they do not conflict. Note that although the accesses in ln. 5,11 in the code (ln. 6,11 in the AST) do have an intersection, they are not in parallel and are also safe.

### 4.2. Memory Layout

CÉU favors a fine-grained use of trails, being common to use trails that await a single event. 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

```
input int A, B, C;
do
    var int a = await A;
end
do
    var int b = await B;
end
par/and do
    await B;
with
    await C;
end
```

```
union {              // sequence
    int a_1;         //   do_1
    int b_2;         //   do_2
    struct {         //   par/and
        int _and_3: 1;
        int _and_4: 1;
    };
} MEM ;
```

Fig. 17. A program with blocks in sequence and in parallel, with corresponding memory layout generated by the compiler.

Römer 2005; Bernauer and Römer 2013]: memory for blocks in sequence are packed in a `struct`, while blocks in parallel, in a `union`. CÉU reserves a single static block of memory to hold all memory slots, whose size is the maximum the program uses at a given time. A given position in the memory may hold different data (with variable sizes) during runtime. As an example, Figure 17 shows a program with corresponding memory layout. Each variable is assigned a unique *id* (e.g. `a_1`) so that variables with the same name can be distinguished. The `do-end` blocks in sequence are packed in a `union`, given that their variables cannot be in scope at the same time, e.g., `MEM.a_1` and `MEM.b_2` can safely share the same memory slot. The example also illustrates the presence of runtime flags related to the parallel composition, which also reside in reusable slots in the static memory.

### 4.3. Trail Allocation

Each line of execution in CÉU needs to carry associated data, such as which event it is 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 can 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 stack level and is delayed until the scheduler runtime reaches that level again. A `FINALIZE` trail represents a hanged finalization block which is only scheduled when its corresponding block goes out of scope. A trail waiting for an event stays in the state of the corresponding event, also holding the minimum sequence number (*seqno*) in which it can awake. In concrete terms, a trail is represented by the following `struct`:

```
struct trail_t {
    state_t evt;
    label_t lbl;
    union {
        unsigned char seqno;
        stack_t       stk;
```

```
    };
};
```

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` state.

The size of `state_t` depends on the number of events in the 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 in 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 `seqno` could eventually overflow during execution (i.e., every 256 reactions). However, given that the scheduler traverses all trails on every reaction, it can adjust them to properly handle overflows (actually, 2 bits to hold the `seqno` is already enough). The size of `stack_t` depends on the maximum depth of nested emissions and is bounded to 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 trail, because they are all hanged 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 with an overhead of 2% in comparison to the original version in *nesC* [Gay et al. 2003] (a dialect of *C* for event-driven programming) [Sant'Anna et al. 2013].

### 4.4. Code Generation and Scheduling

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 18 illustrates the generation process. For the program in the left of the figure, the compiler extracts the entry points and associated trails, e.g., the label `Awake_e` will execute on `TRAIL-0` (ln. 7). For each statement that pauses (`emit` and `await`), resumes (`par/and`, `par/or`, and `finalize`), or aborts (`par/or` and `break`), the compiler splits the trail into segments with associated entry points. The entry points translate to an `enum` in the generated code (ln. 1–10, in the middle of the figure). The state of trails translates to a vector of type `trail_t` with the maximum number of simultaneous trails (ln. 12–15). On initialization, `TRAIL-0` is set to execute the `Main` entry point (ln. 13), while all others are set to `INACTIVE` (in the example, only one, in ln. 14).

The scheduler executes in two passes: in the *broadcast* pass, it sets all trails that are waiting for the current event to `STACKED` in the current stack level; in the *dispatch* pass, it executes each trail that is `STACKED` to run in the current level, setting it immediately to `INACTIVE` (the trail segment may reset it in sequence if it doesn't terminate).

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 previous execution. For the first 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. 13, in the middle of Figure 18).

The code in the right of the Figure 18 dispatches a trail segment according to the current label to execute. For the first reaction, it executes the `Main` label in `TRAIL-0`. When the `Main` label reaches the `par/and`, it stacks `TRAIL-1` (ln. 4–7) and proceeds to the code in `TRAIL-0` (ln. 9–14), respecting the deterministic execution order. The code sets

```
1  input void A;
2  event void e;
3  // TRAIL 0 — lbl Main
4  par/and do
5    // TRAIL 0 — lbl Main
6    await e;
7    // TRAIL 0 — lbl Awake_e
8    // TRAIL 0 — lbl And_chk
9  with
10   // TRAIL 1 — lbl And_sub_2
11   await A;
12   // TRAIL 1 — lbl Awake_A_1
13   emit e;
14   // TRAIL 1 — lbl Emit_cont
15   // TRAIL 1 — lbl And_chk
16 end
17 // TRAIL 0 — lbl And_out
18 await A;
19 // TRAIL 0 — lbl Awake_A_2
20
21
22
23
24
25 .
```

```
1  enum {
2    Main = 1,  // ln 3
3    Awake_e,   // ln 7
4    And_chk,   // ln 8,15
5    And_sub_2, // ln 10
6    Awake_A_1, // ln 12
7    Emit_cont, // ln 14
8    And_out,   // ln 17
9    Awake_A_2  // ln 19
10 };
11
12 trail_t TRLS[2] = {
13   { STACKED,  Main, 0 };
14   { INACTIVE, 0,    0 };
15 };
16
17
18
19
20
21
22
23
24
25 .
```

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

Fig. 18. From left to right: static allocation of trails, entry-point labels, and dispatch function. In the left, the comments identify the trail indexes inferred by the compiler. In the middle, each trail segment has an associated numeric identifier generated by the compiler. In the right, the dispatcher uses a `switch` to associate each segment identifier with the corresponding code to execute.

the running `TRAIL-0` to await `EVT_e` on label `Awake_e`, and then halts with a `break`. The next iteration of `dispatch` takes `TRAIL-1` and executes its registered label `And_sub_2` (ln. 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`, this pass only aborts the desirable trails. The subsequent *dispatch* pass executes the finalization code. Escaping a `loop` that contains parallel compositions also triggers the same abortion process.

### 4.5. The External *C* API

As a reactive language, the execution of a program in CÉU is guided entirely by the occurrence of external events. From the implementation perspective, there are three external sources of input into programs, which are all exposed as functions in a *C* API:

*ceu_go_init():*. initializes the program (e.g., the initial state of trails) and executes the "boot" reaction (i.e., the `Main` label).
*ceu_go_event(id,param):*. executes the reaction for the received external event id and associated parameter.
*ceu_go_wclock(us):*. increments the current time in microseconds and runs a reaction if any timer expires.

Given the semantics of CÉU, the functions are guaranteed to take a bounded time to execute. They also return a status code that says if the CÉU program has terminated after the reactions. Further calls to the API have no effect on terminated programs.

The bindings for the specific platforms are responsible for calling the functions in the API in the order that better suit their requirements. As an example, it is possible to set

```
1  implementation
2  {
3      #include "ceu_app.h"
4      #include "ceu_app.c"
5
6      event void Boot.booted () {
7          ceu_sys_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_sys_wclock(10000);
16     }
17 #endif
18
19 #ifdef EVT_PHOTO_READDONE
20     event void Photo.readDone (int val) {
21         ceu_sys_go(EVT_PHOTO_READDONE, &val);
22     }
23 #endif
24
25 #ifdef EVT_RADIO_SENDDONE
26     event void RadioSend.sendDone (message_t* msg) {
27         ceu_sys_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_sys_go(EVT_RADIO_RECEIVE, &msg);
34         return msg;
35     }
36 #endif
37
38     <...>   // other events
39 }
```

Fig. 19. 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 *Final generation* phase of Figure 15.

different priorities for events that occur concurrently (i.e. while a reaction is running). However, a binding must never interleave or run multiple API calls in parallel. This would break the CÉU sequential/discrete semantics of time.

As an example, Figure 19 shows our binding for *TinyOS* [Hill et al. 2000], which maps callbacks 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 and advances 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 the TinyOS is already synchronous by default and always execute `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.

### 4.6. 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 20 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.

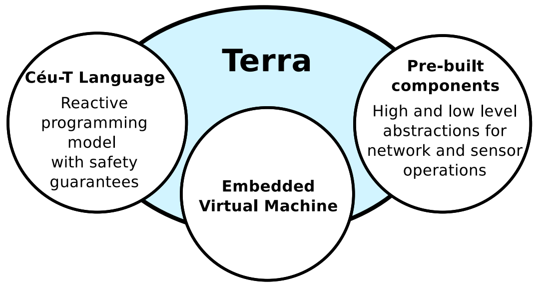


Fig. 20. Terra programming system basic elements.

The main difference between the standard *C* back end and the Terra *VM* is the *Code Generation* phase (as illustrated in Figure 15), 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 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. Considering 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*). The left of Figure 21 shows a CÉU interface with the available functionality for a customized *VM* (with temperature and radio components). The right of the figure 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 19) allows applications to choose all possible combinations of functionalities from the underlying platform at compile time.

## 5. RELATED WORK

CÉU has a strong influence from Esterel and embraces the disciplined synchronous-reactive model with support for lexical composition of lines of execution. However, there are fundamental semantic differences that prevents the design of CÉU as pure extensions to Esterel. In particular, Esterel has a notion of time similar to digital circuits in which multiple signals can be active at a clock tick. In fact, Esterel is also used in hardware design. In CÉU, instead of clock ticks, the occurrence of a single external

```
// 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 }
```

Fig. 21. CÉU interface (in the left) with customized *VM* (in the right). 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).

event that defines a time unit. CÉU also distinguishes external events from stack-based internal events, which provide a limited form of coroutines supporting reactive statements (e.g., `await` and `par/or`).

The event-driven approach of CÉU is well known [Ousterhout 1996] and popular in many software communities, such as client and server-side web frameworks (e.g., *jQuery* [Chaffer 2009] and *Node.js* [Tilkov and Vinoski 2010]), GUI toolkits (e.g., *Tcl/Tk* [Ousterhout 1991] and *Java Swing* [Eckstein et al. 1998]), and Games [Nystrom 2014]. Like CÉU, event-driven programming is essentially synchronous, i.e., events go through a queue and are dispatched sequentially and atomically to prevent race conditions. We believe that for software design, this approach is more familiar to programmers and simplifies the reasoning about concurrency. For instance, the uniqueness of external events in CÉU is a prerequisite for the temporal analysis that enables safe shared-memory concurrency.

Many synchronous languages have been designed to interoperate with C, such as *Reactive C* [Boussinot 1991], Protothreads [Dunkels et al. 2006], *PRET-C* [Andalam et al. 2010] and *SC* [Von Hanxleden 2009]. They offer Esterel-like parallel compositions with communication via shared variables, relying on deterministic scheduling to preserve determinism. However, it is the responsibility of the programmer to specify the execution order for threads, based on either explicit priorities, or source code lexical order. These languages have a tick-based notion of time similar to Esterel, which prevents the event-based temporal analysis of CÉU.

*URBI* [Baillie 2005] is a reactive scripting language with a rich set of control constructs for time management, event-driven communication, and concurrency. Concurrency is based on stackful coroutines, diverging from our goals regarding resource efficiency and static bounds for memory and execution time.

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] 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 respecting the constructive semantics of Esterel. Then, it flattens the graph into sequential code in *C* suitable for static scheduling.

A number of virtual machines have been proposed for embedded systems. The *Sun SPOT* platform with the *Squawk JVM* brings Java for the embedded domain [Simon et al. 2006], but requires a much powerful hardware.[6] *Darjeeling* [Brouwers et al. 2008] and *TakaTuka* [Aslam et al. 2010] are complete *JVMs* 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 with 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.

## 6. CONCLUSION

We presented the design, semantics, and implementation of CÉU, a synchronous reactive language based on Esterel targeting constrained embedded systems.

CÉU is a concurrency-safe language, employing a static analysis that encompass all control constructs and ensures 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 external 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 signaling mechanism. 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.[7]

For the past three years, we have been teaching CÉU for undergraduate and graduate students in research projects and two hands-on 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 of weeks.

## REFERENCES

Sidharta Andalam and others. 2010. Predictable multithreading of embedded applications using PRET-C. In *Proceeding of MEMOCODE'10*. IEEE, 159–168.

Faisal Aslam and others. 2010. Optimized java binary and virtual machine for tiny motes. In *Distributed Computing in Sensor Systems*. Springer, 15–30.

[6]The *Sun SPOT* uses a 32-bit CPU with 4 Mbytes of FLASH and 512 KBytes of SRAM.
[7]Website of CÉU: http://www.ceu-lang.org/

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

Albert Benveniste and others. 2003. The synchronous languages twelve years later. In *Proceedings of the IEEE*, Vol. 91. 64–83.

Alexander Bernauer and Kay Römer. 2013. A Comprehensive Compiler-Assisted Thread Abstraction for Resource-Constrained Systems. In *Proceedings of IPSN'13*. Philadelphia, USA.

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

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

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

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

Adriano Branco, Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, and Silvana Rossetto. 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

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

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

Etienne Closse and others. 2002. Saxo–RT: Interpreting esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science* 65, 5 (2002), 80–94.

Sajeewa Dayaratne and others. 2005. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of SLAP'05*.

Dunkels and others. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*. ACM, 29–42.

Robert Eckstein, Marc Loy, and Dave Wood. 1998. *Java swing*. O'Reilly & Associates, Inc.

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

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

David Gay and others. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI'03*. 1–11.

Omprakash Gnawali and others. 2009. Collection tree protocol. In *Proceedings of SenSys'09*. ACM, 1–14.

Hill and others. 2000. System architecture directions for networked sensors. *SIGPLAN Notices* 35 (November 2000), 93–104. Issue 11.

Roberto Ierusalimschy. 2009. A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.* 39 (March 2009), 221–258. Issue 3.

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

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

Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM TOPLAS* 31, 2 (Feb. 2009), 6:1–6:31.

Robert Nystrom. 2014. *Game Programming Patterns*. Genever Benning.

ORACLE. 2011. Java Thread Primitive Deprecation. http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (accessed in Aug-2014). (2011).

John Ousterhout. 1996. Why Threads Are A Bad Idea (for most purposes). (January 1996).

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

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

Dumitru Potop-Butucaru and others. 2005. The Synchronous Hypothesis and Synchronous Languages. In *Embedded Systems Handbook*, R. Zurawski (Ed.).

Francisco Sant'Anna and others. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. 2006. Java on the bare metal of wireless sensor devices: the Squawk Java Virtual Machine. In *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 78–88.

Olivier Tardieu and Robert De Simone. 2003. Instantaneous termination in pure Esterel. In *Static Analysis*. Springer, 91–108.

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

Reinhard Von Hanxleden. 2009. SyncCharts in C: a proposal for light-weight, deterministic concurrency. In *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 225–234.

Jeong-Han Yun, Chul-Joo Kim, Seonggun Kim, Kwang-Moo Choe, and Taisook Han. 2013. Detection of harmful schizophrenic statements in esterel. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 3 (2013), 80.