Semantics and Implementation of a Synchronous Reactive Language

Francisco Sant'Anna Adriano Branco
Roberto Ierusalimschy Noemi Rodriguez
Departamento de Informática, PUC–Rio
{fsantanna,abranco,roberto,noemi}@inf.puc-rio.br

Silvana Rossetto
Departamento de Ciência da Computação, UFRJ
silvana@dcc.ufrj.br

Abstract

CÉU is a Esterel-based reactive language that targets constrained embedded platforms. On the one hand, its synchronous programming model allows for a simple reasoning about shared-memory concurrency. On the other hand, its restricted semantics results in deterministic and memory-safe programs.

In this work, we propose a formal semantics for CÉU focusing on its particular control mechanisms, such as parallel compositions, finalization, and stack-based events. We also present an implementation with two backends: one aiming for resource efficiency and interoperability with C, and another for code dissemination in sensor networks.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages

Keywords Concurrency, Determinism, Embedded Systems, Esterel, Synchronous, Reactivity

1. Introduction

As a trade-off, our design imposes limitations on the language expressiveness, such as doing computationally-intensive operations and meeting hard real-time responsiveness.

2. Overview 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 time are either reacting to the current event or are awaiting another event; in other words, trails are never reacting to different events.

Figure 1 shows the implementations in Esterel and CÉU side-by-side for the following control specification [?]: *"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

```
// ESTEREL
loop
   abort
      [
         await A
      ||
         await B
      ];
      emit O
   when R
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
```

Figure 1. The same specification in Esterel and CÉU.

is translated almost identically in the two languages (lines 4-9): await and terminate only when both events occur (the '||' and `par/and` constructs are equivalent). For the second phrase, the reset behavior, the Esterel version uses a `abort-when`, which serves the same purpose of CÉU's `par/or`: the occurrence of event `R` aborts the awaiting statements in parallel and restarts the `loop`.

CÉU (like Esterel) has a strong imperative flavor, with explicit control flow through sequences, loops, and also assignments. Being designed for control-intensive applications, it provides support for concurrent lines of execution and broadcast communication through events. Programs advance in sequence of discrete reactions to external events. Internal computations within a reaction (e.g. expressions, assignments, and native calls) are considered to take no time in accordance with the synchronous hypothesis [?]. 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 [? ?].

In the sections that follow, we show the three basic differences between CÉU and Esterel: deterministic execution for operations with side-effects (Section 2.1), hierarchical abortion for lines of execution (Section 2.2), and stack-based execution for internal events (Section 2.3). By providing a precise control for concurrent lines of execution, these differences are fundamental to enable advanced mechanisms in CÉU (presented in Section ??).

2.1 External reactions and determinism

In Esterel, a reaction to the environment is composed of simultaneous signals, while in CÉU, a single event starts a reaction. The notion of time in Esterel is similar to that of digital circuits, in which multiple wires (signals) can be queried for their status (*present* or *absent*) on each clock tick. CÉU more closely reflects event-driven

```
// ESTEREL
abort
    await A;
    call f();
when A;



// code a
```

```
// CEU (or)
par/or do
    await A;
with
    await A;
    _f();
end

// code b
```

```
// CEU (hor)
par/hor do
    await A;
    _g();
with
    await A;
    _f();
end
// code c
```

Figure 2. Thread abortion in Esterel and CÉU.

programming, in which occurring events are handled sequentially and uninterruptedly by the program. Note that even with the single-event rule of CÉU, there is still concurrency given that multiple lines of execution may await and react to the same event.

Another difference between Esterel and CÉU regards their definitions for determinism: Esterel is deterministic with respect to reactive control: "the same sequence of inputs always produces the same sequence of outputs" [**?**]. 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" [**?**]. In CÉU, when multiple trails are active at a time, as in "`par/and do _f1() with _f2() end`", they are scheduled in the order they appear in the program source code (i.e., `_f1` executes first). This way, CÉU is deterministic also with respect to the order of execution of side effects within a reaction.

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 possible. In contrast, Esterel does not support shared memory: *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [**?**]. For embedded development, we believe that deterministic shared-memory concurrency is beneficial, given the extensive use of memory mapped ports for I/O and lack of support for real parallelism. Other embedded languages made a similar design choice [**? ?**].

2.2 Thread abortion

The introductory example of Figure 1 illustrates how synchronous languages can abort awaiting lines of execution (i.e., awaiting `A` and `B`) without tweaking them with synchronization primitives. In contrast, traditional (asynchronous) multi-threaded languages cannot express thread termination safely [**? ?**].

The code fragments of Figure 2 show corner cases for thread abortion: when the event `A` occurs, the program behavior seems ambiguous. For instance, it is not clear in *code a* in Esterel if the call to `f` should execute or not after `A`, given that the body and abortion events are the same. For this reason, Esterel provides *weak* and *strong* variations for the `abort` statement. With *strong* abortion (the default), the body is aborted immediately and the call does not execute. In CÉU, given the deterministic scheduling rules, strong and weak abortions can be chosen by reordering trails inside a `par/or`, e.g., in *code b*, the second trail is strongly aborted by the first trail and the call to `_f` never executes.

CÉU also supports `par/hor` (*hierarchical-or*) compositions which schedules both sides before terminating. Therefore, in *code c*, both `_g` and `_f` (in this order) execute in reaction to `A`. Hierarchical traversal is fundamental for dataflow programming, ensuring that all running dependencies execute before they abort each other (to be discussed in Section **??**).

```
// ESTEREL
input A;    // external
signal B;   // internal
[[
    await A;
    emit B;
    call f("2");
||
    await B;
    call f("1");
]]
```

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

Figure 3. Internal signals (events) in Esterel and CÉU.

```
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
```

Figure 4. Subroutine `inc` is defined in a loop (lines 3-6), in parallel with the caller (lines 8-11).

2.3 Internal events

Esterel makes no semantic distinctions between internal and external signals, both having only the notion of either presence or absence during the entire reaction [**?**]. In CÉU, however, internal events follow a stack-based execution policy, similar to subroutine calls in typical programming languages. Figure 3 illustrates the use of internal signals (events) in Esterel and CÉU. For the version in Esterel, given that there is no control dependency between the calls to `f`, they may execute in any order after `A` and `B` (internally emitted). For the version in CÉU, the occurrence of `A` makes the program behave as follows (with the stack contents in italics):

1. 1st trail awakes (line 5), emits `b`, and pauses.
   stack: [1st]
2. 2nd trail awakes (line 9), calls `_f(1)`, and terminates.
   stack: [1st]
3. 1st trail (on top of the stack) resumes, calls `_f(2)`, 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 4. The subroutine `inc` is defined as a loop (lines 3-6) that continuously awaits its identifying event (line 4), incrementing the value passed as reference (line 5). A trail in parallel (lines 8-11) invokes the subroutine in reaction to event `A` through an `emit` (line 10). Given the stacked execution for internal events, the calling trail pauses, the subroutine awakes (line 4), runs its body (yielding `v=2`), loops, and awaits the next "call" (line 4, again). Only after this sequence that the calling trail resumes and passes the assertion test.

On the one hand, this form of subroutines has a significant limitation that it cannot express recursive calls: an `emit` to itself will always be 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 runtime 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, just like coroutines [?]. In Section **??**, we take advantage of the lack of recursion to properly describe mutual dependency among trails in parallel.

3. Formal Semantics

The disciplined synchronous execution of CÉU, together with broadcast communication and stacked execution for internal events, may raise doubts about the precise execution of programs. In this chapter, we introduce a reduced syntax of CÉU and propose an operational semantics in order to formally describe the language, eliminating imprecisions with regard to how a program reacts to an external event. 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 any conventional imperative language).

3.1 Abstract syntax

```
                                   // 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)
```

Figure 5. Reduced syntax of CÉU.

Figure 5 shows the BNF-like syntax for a subset of CÉU that is sufficient to describe all semantic peculiarities of the language. The $mem(id)$ primitive represents all accesses, assignments, and C function calls 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 ϵ is an unused identifier). Except for the fin and semantic-derived expressions, which are discussed further, the other expressions map to their counterparts in the concrete language in Figure **??**. Note that mem expressions cannot share identifiers with $await$/$emit$ expressions.

3.2 Operational semantics

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

$$\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle$$

where

$$S, S' \in id^* \quad (sequence\ of\ event\ identifiers: [id_{top}, ..., id_1])$$
$$p, p' \in P \quad (as\ described\ in\ Figure\ 5)$$
$$n \in \mathbb{N} \quad (univocally\ identifies\ a\ reaction\ chain)$$

At the beginning of a reaction chain, 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).

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:

$$\langle S,\ await(id)\rangle \xrightarrow[n]{} \langle S,\ awaiting(id, n)\rangle \qquad \textbf{(await)}$$

$$\langle id : S,\ awaiting(id, m)\rangle \xrightarrow[n]{} \langle id : S,\ nop\rangle,\ \ m < n \qquad \textbf{(awaiting)}$$

$$\langle S,\ emit(id)\rangle \xrightarrow[n]{} \langle id : S,\ emitting(|S|)\rangle \qquad \textbf{(emit)}$$

$$\langle S,\ emitting(|S|)\rangle \xrightarrow[n]{} \langle S,\ nop\rangle \qquad \textbf{(emitting)}$$

An $await$ is simply transformed into an $awaiting$ that remembers the current external sequence number n (rule **await**). An $awaiting$ can only transit to a nop (rule **awaiting**) if its referred event id matches the top of the stack and its sequence number is smaller than the current one ($m < n$). 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 (in rule **emit**). With the new stack level $|S| + 1$, the $emitting(|S|)$ itself cannot transit, as rule **emitting** 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:

$$\frac{val(id, n) \neq 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \xrightarrow[n]{} \langle S, p\rangle} \qquad \textbf{(if-true)}$$

$$\frac{val(id, n) = 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \xrightarrow[n]{} \langle S, q\rangle} \qquad \textbf{(if-false)}$$

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p\ ;\ q)\rangle \xrightarrow[n]{} \langle S', (p'\ ;\ q)\rangle} \qquad \textbf{(seq-adv)}$$

$$\langle S, (mem(id)\ ;\ q)\rangle \xrightarrow[n]{} \langle S, q\rangle \qquad \textbf{(seq-nop)}$$

$$\langle S, (break\ ;\ q)\rangle \xrightarrow[n]{} \langle S, break\rangle \qquad \textbf{(seq-brk)}$$

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 the memory identifier and current reaction sequence number, returning the current memory value. Although the value is arbitrary, it is unique in a reaction chain, 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:

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

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

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

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

When a program first 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 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.

The rules for parallel and compositions force transitions on the left branch p to occur before transitions on the right branch q (rules **and-adv1** and **and-adv2**). Then, if one of the sides terminates, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**):

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

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

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

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

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

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

The deterministic behavior of the semantics relies on the *isBlocked* predicate, defined in Figure 6 and used in rule **and-adv2**, requiring the left branch p to be blocked in order to allow the right transition from q to q'. An expression becomes blocked when all of its trails in parallel hang in $awaiting$ and $emitting$ expressions.

The last two rules **and-brk1** and **and-brk2** deal with a $break$ in each of the sides in parallel. A $break$ should terminate the whole composition in order to escape the innermost loop (*aborting* the other side). The $clear$ function in the rules, defined in Figure 7, concatenates all active fin bodies of the side being aborted (to execute before the and rejoins). Note that there are no transition rules for fin expressions. This is because once reached, an fin expression only executes when it is aborted by a trail in parallel. In Section 3.3.3, we show how an fin is mapped to a finalization block in the concrete language. Note that there is a syntactic restriction that an fin body cannot $emit$ or $await$—they are guaranteed to completely execute within a reaction chain.

Most rules for parallel or compositions are similar to and compositions:

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

Figure 6. The recursive predicate $isBlocked$ is true only if all branches in parallel are hanged in $awaiting$ or $emitting$ expressions that cannot transit.

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

Figure 7. The function $clear$ extracts fin expressions in parallel and put their bodies in sequence.

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p \mathit{ or } q)\rangle \xrightarrow[n]{} \langle S', (p' \mathit{ or } q)\rangle} \quad \textbf{(or-adv1)}$$

$$\frac{isBlocked(n, S, p)\ ,\ \ \langle S, q\rangle \xrightarrow[n]{} \langle S', q'\rangle}{\langle S(p \mathit{ or } q)\rangle \xrightarrow[n]{} \langle S', (p \mathit{ or } q')\rangle} \quad \textbf{(or-adv2)}$$

$$\langle S, (mem(id) \mathit{ or } q)\rangle \xrightarrow[n]{} \langle S, clear(q)\rangle \quad \textbf{(or-nop1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p \mathit{ or } mem(id))\rangle \xrightarrow[n]{} \langle S, clear(p)\rangle} \quad \textbf{(or-nop2)}$$

$$\langle S, (break \mathit{ or } q) \xrightarrow[n]{} \langle S, (clear(q)\ ;\ break)\rangle \quad \textbf{(or-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p \mathit{ or } break)\rangle \xrightarrow[n]{} \langle S, (clear(p)\ ;\ break)\rangle} \quad \textbf{(or-brk2)}$$

For a parallel or, the rules **or-nop1** and **or-nop2** must terminate the composition, and also apply the function $clear$ to the aborted side, in order to properly finalize it.

A reaction chain eventually blocks in $awaiting$ and $emitting$ expressions in parallel trails. If all trails hangs only in $awaiting$ expressions, it means that the program cannot advance in the current reaction chain. However, $emitting$ expressions should resume their continuations of previous $emit$ in the ongoing reaction, they are just hanged in lower stack indexes (see rule **emit**). Therefore, we define another relation that behaves as the previous if the program is not blocked, and, otherwise, pops the stack:

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, p\rangle \xRightarrow[n]{} \langle S', p'\rangle} \qquad \frac{isBlocked(n,\ s : S,\ p)}{\langle s : S, p\rangle \xRightarrow[n]{} \langle S, p\rangle}$$

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

$$\langle S, p\rangle \xRightarrow[n]{*} \langle S', p'\rangle$$

Finally, to describe the complete execution of a program, we need multiple "invocations" of reaction chains, incrementing the sequence number:

$$\langle [e1], p\rangle \xRightarrow[1]{*} \langle [], p'\rangle$$
$$\langle [e2], p'\rangle \xRightarrow[2]{*} \langle [], p''\rangle$$
$$\ldots$$

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, the sequence number is incremented.

3.3 Concrete language mapping

Although the reduced syntax presented in Figure 5 is similar to the concrete language in Figure **??**, there are some significant mismatches between CÉU and the formal semantics that require some clarification. In this section, we describe an informal mapping between the two.

Most statements from CÉU map directly to the formal semantics, e.g., `if` $\mapsto if$, `';'` $\mapsto$ $';'$, `loop` $\mapsto loop$, `par/and` $\mapsto$ and, `par/or` $\mapsto or$. (Again, we are not considering side-effects, which are all mapped to the mem semantic construct.)

3.3.1 await and emit

The `await` and `emit` primitives of CÉU are slightly more complex in comparison to the formal semantics, as they support communication of values between emits and awaits. In the two-step translation below, we start with the 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 formal semantics, translating side-effect statements into mem expressions:

```
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
```

Note that a similar translation is required for external events, i.e., each external event has a corresponding variable that is explicitly set by the environment before each reaction chain.

3.3.2 First-class timers

To encompass first-class timers, we need a special `TICK` event that should be intercalated with each other event occurrence in an application (e.g. *e1, e2*):

$$\langle [TICK], p\rangle \xRightarrow[1]{*} \langle [], p'\rangle$$
$$\langle [e1], p'\rangle \xRightarrow[2]{*} \langle [], p''\rangle$$
$$\langle [TICK], p''\rangle \xRightarrow[3]{*} \langle [], p'''\rangle$$
$$\langle [e2], p'''\rangle \xRightarrow[4]{*} \langle [], p''''\rangle$$
$$\ldots$$

The `TICK` event has an associated variable `TICK_` (as illustrated in the previous section) with the time elapsed between the two occurrences of external events.

The translation in two steps from a timer await to the semantics is as follows:

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

```
var int tot = 10000;
loop do
    await TICK;
    tot = tot — TICK_;
    if tot <= 0 then
        dt = tot;
        break;
    end
end
```

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

3.3.3 Finalization blocks

The biggest mismatch between CÉU and the formal semantics is regarding finalization blocks, which require more complex modifications in the program for a proper mapping using the fin semantic construct. The code that follows uses a `finalize` to safely `_release` the reference to `ptr` kept after the call to `_hold`:

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

In the translation to the semantics, the first required modification is to catch the `do-end` termination to run the finalization code. For this, we translate the block into a `par/or` with the original body in parallel with a fin to run the finalization code:

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

In this intermediate code (mixing the syntaxes), the fin body will execute whenever the `par/or` terminates, either normally (after the `await B`) or aborted from an outer composition (rules **and-brk1**, **and-brk2**, **or-nop1**, **or-nop2**, **or-brk1**, and **or-brk2** in the semantics). However, the fin will also (incorrectly) execute even if the call to `_hold` is not 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:

```
1   f_ = 0;
2   par/or do
3       var int* ptr = <...>;
4       await A;
5       _hold(ptr);
6       f_ = 1;
7       await B;
8   with
9       { fin
10          if f_ then
11              _release(ptr);
12          end }
13  end
```

The flag is initially set to false (line 1), avoiding the finalization code to execute (lines 9-12). Only after the call to `_hold` (line 5) that we set the flag to true (line 6) and enable the fin body to execute. The complete translation from the original example in CÉU is as follows:

```
mem;     // f_ = 0
(
   mem;          // ptr = <...>
   await(A);
   mem;          // _hold(ptr)
   mem;          // f_ = 1
   await(B);
or
   fin
     if mem then    // if f_
        mem         // release _ptr
     else
        nop
)
```

4. Implementation

The compilation process of a program in CÉU is composed of three main phases, as illustrated in Figure 8:

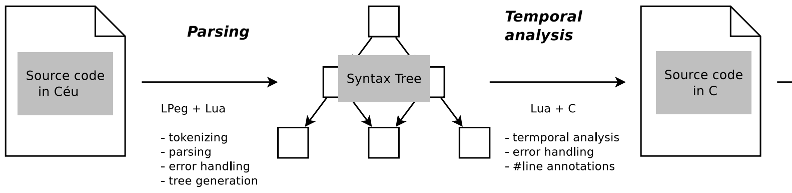

Figure 8. Compilation process: from the source code in CÉU to the final binary.

Parsing The parser of CÉU is written in *LPeg* [**?**], 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 and the forms of non-determinism. It also makes some "classical" semantic analysis, such as building a symbol table for checking variable declarations. However, most of type checking is delayed to the last phase to take advantage of GCC's error handling. Therefore, this phase needs to annotate the C output with `#line` pragmas that match the original file in CÉU. This phase must output code in C, given how tied CÉU is to C by design.

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 includes 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.

In the sections that follow, we discuss the most sensible parts of the compiler considering our design, such as the temporal analysis, runtime scheduler, and the external API.

5. Temporal analysis

As introduced, the *temporal analysis* phase detects inconsistencies in CÉU programs. Here, we focus on the algorithm that detects non-deterministic access to variables, as presented in Section **??**.

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 terminate, 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 the sequence parent, 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 (`par/and` / `par/or`) may terminate from any of its branches, hence $O = O_1 \cup ... \cup O_n$.

With all sets calculated, any two nodes that perform side effects and are in parallel branches can have their I sets compared for intersections. If the intersection is not the empty set, they are marked as suspicious (see Section **??**).

Figure 9 reproduces the second code of Figure **??** and shows the corresponding AST with the sets I and O for each node. The event . (dot) represents the "boot" reaction. The assignments to `y` in parallel (lines 5,8 in the code) have an empty intersection of I (lines 6,9 in the AST), hence, they do not conflict. Note that although the accesses in lines 5, 11 in the code (lines 6,11 in the AST) do have an intersection, they are not in parallel and are also safe.

```
input void A, B;
var int y;
par/or do
  await A;
  y = 1;
with
  await B;
  y = 2;
end
await A;
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}
```

Figure 9. 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.

6. Memory layout

CÉU favors a fine-grained use of trails, being common the use of 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 flags needed during runtime. Memory for trails in parallel must coexist, while statements in sequence can reuse it. 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.

Translating this idea to C is straightforward [**? ?**]: memory for blocks in sequence are packed in a `struct`, while blocks in parallel, in a `union`. As an example, Figure 10 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 address. The example also illustrates the presence of runtime flags related to the parallel composition, which also reside in reusable slots in the static memory.

7. Trail allocation

The compiler extracts the maximum number of trails a program can have at the same time and creates a static vector to hold runtime information about them. Again, trails that cannot be active at the same time can share memory slots in the static vector.

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

```
enum {
    INACTIVE = 0,
    STACKED,
```

```
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
        u8 _and_3: 1;
        u8 _and_4: 1;
    };
} MEM ;
```

Figure 10. A program with blocks in sequence and in parallel, with corresponding memory layout.

```
    FIN,
    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 its associated stack level and is delayed until the scheduler runtime level reaches that value again. A `FIN` 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 sequence number (*seqno*) in which it started awaiting. 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 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 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` will eventually overflow during execution (every 256 reactions). However, given that the scheduler traverses all trails in each reaction, it can adjust them to properly handle overflows (actually 2 bits to hold the `seqno` would be already enough). The stack size depends on the maximum depth of nested emissions and is bounded to the maximum number of trails, e.g., 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 will be all hanged in a `STACKED` state. In WSNs applications, the size of `trail_t` is typically only 3 bytes (1 byte for each field).

7.1 Code generation

The example in Figure 11 illustrates how trails and labels are statically allocated in a program. The program has a maximum of 2 trails, because the `par/and` (line 4) can reuse *TRAIL 0*, and the join point (line 16) can reuse both *TRAIL 0* and *TRAIL 1*. Each label is

```
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 ParAnd_chk
9  with
10   // TRAIL 1 — lbl ParAnd_sub_2
11   await A;
12   // TRAIL 1 — lbl Awake_A_1
13   emit e;
14   // TRAIL 1 — lbl Emit_e_cont
15   // TRAIL 1 — lbl ParAnd_chk
16 end
17 // TRAIL 0 — lbl ParAnd_out
18 await A;
19 // TRAIL 0 — lbl Awake_A_2
```

```
enum {
  Main = 1,      // ln  3
  Awake_e,       // ln  7
  ParAnd_chk,    // ln
8, 15
  ParAnd_sub_2,  // ln 10
  Awake_A_1,     // ln 12
  Emit_e_cont,   // ln 14
  ParAnd_out,    // ln 17
  Awake_A_2      // ln 19
};
```

Figure 11. Static allocation of trails and entry-point labels.

```
1  while (<...>) {              // scheduler main loop
2     trail_t* trail = <...>    // choose next trail
3     switch (trail->lbl) {
4        case Main:
5           // activate TRAIL 1 to run next
6           TRLS[1].evt = STACKED;
7           TRLS[1].lbl = ParAnd_sub_2;
// 2nd trail of par/and
8           TRLS[1].stk = current_stack;
9
10          // code in the 1st trail of par/and
11          // await e;
12          TRLS[0].evt = EVT_e;
13          TRLS[0].lbl = Awake_e;
14          TRLS[0].seq = current_seqno;
15          break;
16
17       case ParAnd_sub_2:
18          // await A;
19          TRLS[1].evt = EVT_A;
20          TRLS[1].lbl = Awake_A_1;
21          TRLS[1].seq = current_seqno;
22             break;
23
24       <...>   // other labels
25    }
26 }
```

Figure 12. Generated code for the program of Figure 11.

associated with a unique identifier in the `enum`. The static vector to hold the two trails in the example is defined as

```
trail_t TRLS[2];
```

In the final generated *C* code, each label becomes a *switch case* working as the entry point to execute its associated code. Figure 12 shows the corresponding code for the program of Figure 11. The program is initialized with all trails set to `INACTIVE`. Then, the scheduler executes the *Main* label in the first trail. When the *Main* label reaches the `par/and`, it "stacks" the 2nd trail of the `par/and` to run on *TRAIL 1* (line 5-8) and proceeds to the code in the 1st trail (lines 10-15), respecting the deterministic execution order. The code sets the running *TRAIL 0* to await `EVT_e` on label `Awake_e`, and then halts with a `break`. The next iteration of the scheduler takes *TRAIL 1* and executes its registered label `ParAnd_sub_2` (lines 17-22), which sets *TRAIL 1* to await `EVT_A` and also halts.

Regarding cancellation, trails in parallel are always allocated in subsequent slots in the static vector `TRLS`. Therefore, when a `par/or` terminates, the scheduler sequentially searches and executes `FIN` trails within the range of the `par/or`, and then clears all of them to `INACTIVE` at once. Given that finalization blocks cannot contain `await` statements, the whole process is guaranteed to terminate in bounded time. Escaping a `loop` that contains parallel compositions also trigger the same process.

8. 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. trails) and executes the "boot" reaction (i.e., the `Main` label).

ceu_go_event(id,param): executes the reaction for the received 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 different priorities for events that occur concurrently (i.e. while a reaction chain is running). However, a binding must never interleave or run multiple functions in parallel. This would break the CÉU sequential/discrete semantics of time.

As an example, Figure 13 shows our binding for *TinyOS* which maps *nesC* callbacks to input events in CÉU. The file `ceu.h` (included in line 3) contains all definitions for the compiled CÉU program, which are further queried through `#ifdef`'s. The file `ceu.c` (included in line 4) contains the main loop of CÉU pointing to the labels defined in the program. The callback `Boot.booted` (lines 6-11) is called by TinyOS on mote startup, so we initialize CÉU inside it (line 7). If the CÉU program uses timers, we also start a periodic timer (lines 8-10) that triggers callback `Timer.fired` (lines 13-17) every 10 milliseconds and advances the wall-clock time of CÉU (line 15)[1]. The remaining lines map pre-defined TinyOS events that can be used in CÉU programs, such as the light sensor (lines 19-23) and the radio transceiver (lines 25-36).

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

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

```
implementation
{
    #include "ceu.h"
    #include "ceu.c"

    event void Boot.booted () {
        ceu_go_init();
#ifdef CEU_WCLOCKS
        call Timer.startPeriodic(10);
#endif
    }

#ifdef CEU_WCLOCKS
    event void Timer.fired () {
        ceu_go_wclock(10000);
    }
#endif

#ifdef _EVT_PHOTO_READDONE
    event void Photo.readDone (uint16_t val) {
        ceu_go_event(EVT_PHOTO_READDONE, (void*)val);
    }
#endif

#ifdef _EVT_RADIO_SENDDONE
    event void RadioSend.sendDone (message_t* msg) {
        ceu_go_event(EVT_RADIO_SENDDONE, msg);
    }
#endif

#ifdef _EVT_RADIO_RECEIVE
    event message_t* RadioReceive.receive (message_t* msg) {
        ceu_go_event(EVT_RADIO_RECEIVE, msg);
        return msg;
    }
#endif

    <...>   // other events
}
```

Figure 13. The *TinyOS* binding for CÉU.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...