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

Francisco Sant'Anna, Departamento de Informática, PUC–Rio
Adriano Branco, 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

CÉU is a reactive language based on Esterel that targets constrained embedded platforms. Employing the synchronous programming model, it allows for a simple reasoning about shared-memory concurrency. Furthermore, its restricted semantics enables 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 internal 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.

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

1. INTRODUCTION

- Relevancia de Esterel
- Ceu
  - Foco inicial em "constrained WSNs"
  - Estendendo para outros dominios? (jogos, artigo Mod'15)
  - Usos: SenSys, Terra, sala de aula, GSoC
- Limitacoes do modelo sincrono
- Semantica e Implementacao (nesse artigo, somente subset estatico)

2. OVERVIEW OF CÉU

CÉU is a synchronous reactive language based on Esterel [Boussinot and de Simone 1991] 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.

In the sections that follow, we review the main differences between CÉU and Esterel [Sant'Anna et al. 2013]: 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 (lines 4–9, in both implementations), as Esterel's '||' and CÉU's `par/and` constructs are equivalent. For the second phrase, the reset behavior, the Esterel version uses a `abort-when` (lines 3–10), which serves the same purpose of CÉU's `par/or` (lines 3–12): the occurrence of event `R` aborts the awaiting statements in parallel and restarts the `loop`.

CÉU, following the Esterel mindset, 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

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

```
// CEU
loop do
   par/or do
      par/and do
         await A;
      with
         await B;
      end
      emit O;
   with
      await R;
   end
end
```

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

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

```
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.

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 native 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].

2.1. 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 the entire reaction [Berry 1993]. In CÉU, however, internal events follow a stack-based execution policy, similar to subroutine calls in typical programming languages [Sant'Anna et al. 2013], contrasting with queue-based event handling adopted for external events. Figure 2 illustrates the use of internal signals (events) in Esterel and CÉU. In the version in Esterel, on the occurrence of `A`, `B` is emitted and they both 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 (line 5), emits `b`, and pauses.
stack: [1st-trail]
(2) 2nd trail awakes (line 9), calls `_f()`, and terminates.
stack: [1st-trail]
(3) 1st trail (on top of the stack) resumes, calls `_g()`, and terminates.
stack: []
(4) Both trails have terminated, so the `par/and` rejoins, and the program also terminates;

```
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 (lines 3–6), in parallel with the caller (lines 8–11).

Internal events bring support for a limited form of subroutines, as depicted in Figure 3. 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 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].

2.2. Shared-Memory Concurrency and Determinism

Embedded applications make extensive use of shared memory, such as for sharing resources through memory-mapped registers. Hence, an important goal of CÉU is to ensure a reliable execution for concurrent programs that share 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: *"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 chain. A trail segment is a sequence of statements followed by an `await` (or termination). In the program in the left of Figure 4, the two assignments to `x` can never run concurrently, because each trail segment reacts to a different input event and, according to the synchronous model, each reaction is atomic. However, the program in the right is non-deterministic, because the two assignments to `y` occur in the same reaction to input `A`.

CÉU performs a temporal analysis at compile time and detects concurrent accesses to shared variables, as follows [Sant'Anna et al. 2013]: *if a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable, nor*

```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```

```
input void A;
var int y = 1;
par/and do
    await A;
    y = y + 1;
with
    await A;
    y = y * 2;
end
```

Fig. 4. 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
[
    call f1();
||
    call f2();
];
```

```
// CEU
par/and do
    _f1();
with
    _f2();
end
```

Fig. 5. In Esterel, the execution order between f1 and f2 is unspecified, whereas in CÉU, _f1 executes before _f2.

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). The algorithm for the analysis holds the set of all events in preceding await statements for each variable access. Then, the sets for all accesses in parallel trails are compared to assert that no events are shared among them. Otherwise the compiler warns about the suspicious accesses.

Regardless of the static analysis, when multiple trails are active at a time in CÉU, they are scheduled in the order they appear in the program source code. Therefore, even though the program in the right of Figure 4 is suspicious, the assignments to y both atomic and deterministic, i.e., after A occurs, 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 5 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.

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]. However, handling abortion when dealing with external resources can be challenging, given that external entities are not subject to the same synchronous execution discipline.

To illustrate threats related to abortion, consider the unsafe example in the left of Figure 6, describing the state machine of a data collection protocol for sensor net-

```
1  input void STOP, RETRANSMIT, SENDACK;
2  par/or do
3      await STOP;
4  with
5      loop do
6          par/or do
7              await RETRANSMIT;
8          with
9              par/and do
10                 await 1min;
11             with
12                 var _pkt_t buffer;
13                 <fill-buffer-info>
14                 _send_enqueue(&buffer);
15                 await SENDACK;
16             end
17         end
18     end
19 end
```

```
11 <...>
12     var _pkt_t buffer;
13     <fill-buffer-info>
14     finalize
15         _send_enqueue(&buffer);
16     with
17         _send_cancel(&buffer);
18     end
19     await SENDACK;
20 <...>
```

Fig. 6. The unsafe network protocol in the left is extended with a finalization clause in the right to handle abortion properly.

works [Gnawali et al. 2009; Sant'Anna et al. 2013]. The input events `STOP`, `RETRANSMIT`, and `SENDACK` (line 1) represent the external interface of the protocol with a client application. The protocol has to transmit a packet every minute, unless it receives a `RETRANSMIT` request to immediately re-transmit, or a `STOP` request to terminate. The protocol is composed of two main trails: one simply monitors the stopping event (line 3); the other periodically transmits a packet (lines 5–18). The periodic transmission is a loop that starts two other trails (lines 6–17): one handles the immediate retransmission request (line 7); the other transmits the packet[1] and waits for a confirmation (lines 9–16). The actual transmission (lines 12–15) is enclosed with a `par/and` that takes at least one minute before looping, to avoid flooding the network with packets. Note that the call to `_send_enqueue` is *asynchronous*, handing to the radio driver a pointer to the lexically-scoped packet. The driver will make the transmission in the background, holding the packet until it signals the application with the `SENDACK` to acknowledge completion. At any time, the client may request a retransmission (line 7), which terminates the `par/or` (line 6), aborts the ongoing transmission (line 14, if not idle), and restarts the loop (line 5). The client may also request to stop the whole protocol at any time (line 3). In the meantime, if the sending trail is be aborted by the `STOP` or `RETRANSMIT` requests, the packet buffer goes out of scope, leaving behind a *dangling pointer* in the radio driver, which will possibly transmit wrong data.

The CÉU compiler tracks the interaction of `par/or` compositions with local variables and stateful *C* functions (e.g., device drivers) in order to preserve safe abortion of trails [Sant'Anna et al. 2013; 2015]. For instance, CÉU refuses to compile the program in the left of Figure 6, enforcing the programmer to write a *finalization* clause to accompany the stateful *C* call. The code in the right of Figure 6 properly cancels the packet transmission when the block of `buffer` goes out of scope, i.e., the finalization clause (after the `with`) executes automatically on external abortion.[2]

[1] The underline prefix marks (e.g., `_send_enqueue`) make interactions with external *C* libraries explicit and are required in CÉU.

[2] Note that the compiler only enforces the programmer to write the finalization clause, but cannot check if it actually handles the resource properly.

```
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. 7. First-class timers in CÉU.

2.4. First-Class Timers

Activities that involve reactions to *wall-clock time*^{3} 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 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 [Sant'Anna et al. 2013], resulting in more robust applications. For the example in the left of Figure 7, 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 7, 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 main differences to Esterel, 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 any conventional imperative language).

```
                                   // 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. 8. Reduced syntax of CÉU.

3.1. Abstract Syntax

Figure 8 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 map to their counterparts in the concrete 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). Note that mem and *await*/*emit* expressions do not share identifiers, i.e., an identifier is either a variable or an event.

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:

$$\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle \qquad \textbf{(relation-inner)}$$

where

$$\begin{aligned} S, S' &\in id^* && (sequence\ of\ event\ identifiers : [id_{top}, ..., id_{bottom}]) \\ p, p' &\in P && (program\ as\ described\ in\ Figure\ 8) \\ n &\in \mathbb{N} && (unique\ identifier\ for\ the\ reaction\ chain) \end{aligned}$$

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). The ever-increasing sequence number n will prevent that awaiting expressions awake in the same reaction they are reached (ensuring that the program can progress).

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

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

follows:

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

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

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

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

An *await* is simply transformed into an *awaiting* (rule **awaiting**) as an artifact to remember the external sequence number n 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 its sequence number is smaller or equal 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 (rule **emitting**). 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:

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

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

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

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

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

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 here 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 **awaiting**, 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:

$$\langle S, (\mathit{loop}\ p)\rangle \xrightarrow[n]{} \langle S, (p\ @\ \mathit{loop}\ p)\rangle \qquad \textbf{(loop-expd)}$$

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

$$\langle S, (\mathit{mem}(\mathit{id})\ @\ \mathit{loop}\ p)\rangle \xrightarrow[n]{} \langle S, \mathit{loop}\ p\rangle \quad \textbf{(loop-nop)}$$

$$\langle S, (\mathit{break}\ @\ \mathit{loop}\ p)\rangle \xrightarrow[n]{} \langle S, \mathit{nop}\rangle \qquad \textbf{(loop-brk)}$$

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 $\mathit{mem}(\mathit{id})$. 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.

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:

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

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

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

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

The deterministic behavior of the semantics relies on the *isBlocked* predicate, defined in Figure 9 and used in rules **and-adv2** and **or-adv2**, requiring 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 $\mathit{awaiting}$ and $\mathit{emitting}$ expressions.

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**). The last two rules **and-brk1**

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

Fig. 9. The recursive predicate *isBlocked* is true only if all branches in parallel are hanged in *awaiting* or *emitting* expressions that cannot transit.

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

Fig. 10. The function *clear* extracts *fin* expressions in parallel and put their bodies in sequence.

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):

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

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

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

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

The *clear* function, defined in Figure 10, 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, 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 a *fin* is mapped to a finalization block in the concrete language. 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 chain.

For a parallel *or*, if one of the sides terminates, the whole composition terminates and applies the function *clear* to the aborted side in order to properly finalize it (rules

or-nop1 and **or-nop2**). Finally, a $break$ (rules **or-brk1** and **or-brk2**) behaves in the same way as in a parallel and:

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

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

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

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

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 **pop**). Therefore, we define another relation that behaves as **relation-inner** (presented above) if the program is not blocked, 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 chain* 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 need multiple "invocations" of reaction chains, incrementing the sequence number:

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

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, the sequence number is incremented.

3.3. Concrete Language Mapping

Most statements from CÉU map directly to those presented in the reduced syntax of Figure 8, namely, the `if`, `';'`, `loop`, `par/and`, and `par/or`. For instance, the `if` in the concrete language behaves exactly like the if in the described semantics. However, there are some significant mismatches between CÉU and the formal semantics, which we provide a mapping in this section.

Again, we are not considering side-effects, which are all mapped to the mem semantic construct.

```
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. 11. Two-step translation from concrete to abstract `emit` and `await` expressions.

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

```
var int tot = 10000; // 10ms
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
)
```

Fig. 12. Two-step translation from concrete to abstract timer.

3.3.1. await and emit. The concrete `await` and `emit` primitives of CÉU support communication of values between them. In the two-step translation of Figure 11, 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 formal semantics, 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 chain.

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

$$\begin{aligned}
\langle [TICK], p\rangle &\overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle \\
\langle [e1], p'\rangle &\overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle \\
\langle [TICK], p''\rangle &\overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle \\
\langle [e2], p'''\rangle &\overset{*}{\underset{4}{\Longrightarrow}} \langle [], p''''\rangle \\
&\ldots
\end{aligned}$$

The `TICK` event has an associated variable `TICK_` carrying the wall-clock time elapsed between the two occurrences, as depicted by the two-step translation of Figure 12.

TODO: continuar explicacao

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. In the three-step translation of Figure 13, 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 to

```
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. 13. Three-step translation from concrete to abstract finalization.

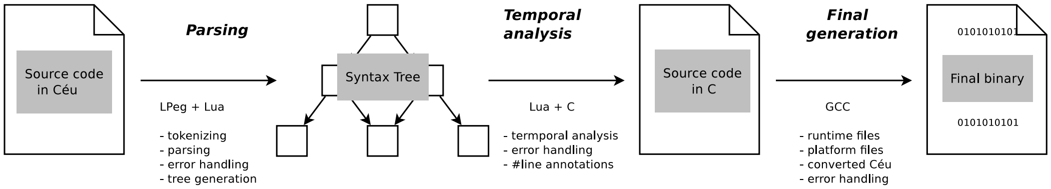

Fig. 14. Compilation process: from the source code in CÉU to the final binary.

the semantics, we first need to catch the `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 to run the finalization code. This way, the fin body executes whenever 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 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 14:

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.

Analysis. This phase detects inconsistencies in CÉU programs, such as unbounded loops, and also makes “classical” semantic analysis, such as building a symbol table for checking variable declarations. This phase outputs code in C, given the way CÉU is tied to C by design. 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.

Code Generation. The final phase packs the generated C file with the CÉU runtime and platform-dependent functionality, compiling them with *gcc* and generating the

```
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 ;
```

Fig. 15. A program with blocks in sequence and in parallel, with corresponding memory layout.

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.

4.1. 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 [Kasten and Römer 2005; Bernauer and Römer 2013]: memory for blocks in sequence are packed in a `struct`, while blocks in parallel, in a `union`. As an example, Figure 15 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.

4.2. Trail Allocation

Each line of execution in CÉU carries 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. 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 of the events defined in the program:

```
enum {
    INACTIVE = 0,
    STACKED,
    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 minimum sequence number (*seqno*) in which it can awake. 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` is 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 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 has at most eight simultaneous lines of execution with an overhead of 2% in comparison to the original version in *nesC* (a dialect of *C* for event-driven programming) [Sant'Anna et al. 2013].

4.3. Code Generation and Scheduling

In the final generated code in *C*, each trail label representing an entry point becomes a *switch case* with the associated code to execute. Figure 16 illustrates the generation process. For the program in the left of the figure, the compiler extracts the entry points and associated trails. The entry points translate to an `enum` in the generated code (lines 1–10, in the middle of the figure). The trails translate to a vector of `trail_t` elements with the maximum number of simultaneous trails (lines 12–16). Initially, `TRAIL-0` is set to execute the `Main` entry point, while all others are set to `INACTIVE`.

The scheduler executes in two passes: In the *broadcast* pass, the scheduler sets to `STACKED` all trails that are waiting for the current event to execute in the current stack level. In the *dispatch* pass, the scheduler executes each trail that is `STACKED` to run in the current level, setting it immediately to `INACTIVE`.

During the dispatch pass, if a trail executes and emits and 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 chain, the scheduler starts from the *dispatch* pass,

```
input void A;
event void e;
// TRAIL 0 — lbl Main
par/and do
  // TRAIL 0 — lbl Main
  await e;
  // TRAIL 0 — lbl Awake_e
  // TRAIL 0 — lbl And_chk
with
  // TRAIL 1 — lbl And_sub_2
  await A;
  // TRAIL 1 — lbl Awake_A_1
  emit e;
  // TRAIL 1 — lbl Emit_cont
  // TRAIL 1 — lbl And_chk
end
// TRAIL 0 — lbl And_out
await A;
// TRAIL 0 — lbl Awake_A_2






.
```

```
 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. 16. Static allocation of trails, entry-point labels, and dispatch loop.

given that the `Main` label is the only one that can be active at the stack level 0 (line 13, in the middle of Figure 16).

The code the right of the Figure 16 dispatches a trail according to its 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` (lines 4–8) and proceeds to the code in `TRAIL-0` (lines 9–14), 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 `dispatch` takes `TRAIL-1` and executes its registered label `And_sub_2` (lines 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 `FIN` event, but limited to the range of trails covered by the terminating `par/or`. Trails that do not match the `FIN` 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 trigger the same abortion process.

4.4. 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.

```
1  implementation
2  {
3      #include "ceu.h"
4      #include "ceu.c"
5
6      event void Boot.booted () {
7          ceu_go_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_go_wclock(10000);
16     }
17 #endif
18
19 #ifdef _EVT_PHOTO_READDONE
20     event void Photo.readDone (uint16_t val) {
21         ceu_go_event(EVT_PHOTO_READDONE, (void*)val);
22     }
23 #endif
24
25 #ifdef _EVT_RADIO_SENDDONE
26     event void RadioSend.sendDone (message_t* msg) {
27         ceu_go_event(EVT_RADIO_SENDDONE, msg);
28     }
29 #endif
30
31 #ifdef _EVT_RADIO_RECEIVE
32     event message_t* RadioReceive.receive (message_t* msg) {
33         ceu_go_event(EVT_RADIO_RECEIVE, msg);
34         return msg;
35     }
36 #endif
37
38     <...>   // other events
39 }
```

Fig. 17. The *TinyOS* binding for CÉU.

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.

TODO: refs/cite WSNs

As an example, Figure 17 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 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)[4]. The remaining lines

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

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).

5. RELATED WORK

- Kiel Esterel compiler

- We are not concerned with WCET: Predictable system design is concerned with the challenge o f building sys- tems in such a way that requirements can be guaranteed from th e design. This means that an off-line analysis should demonstrate satisfa ction of timing require- ments, subject to assumptions made on operating conditions foreseen for the sys- tem [Stankovic and Ramamritham 1990]

PRET-C http://embedded.cs.uni-saarland.de/publications/BuildingTimingPredictableEmbeddedSystems.pdf

6. FINAL REMARKS

REFERENCES

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

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.

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

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

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.

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

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.

Francisco Sant'Anna and others. 2015. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*. to appear.