Advanced Control Reactivity for Embedded Systems

Accepted paper in REM'13 (preprint version)

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

ABSTRACT
Céu is a Esterel-based reactive language that targets constrained embedded platforms. Relying on a deterministic semantics, it provides safe shared-memory concurrency among lines of execution. Céu introduces a stack-based execution policy for internal events which enables advanced control mechanisms considering the context of embedded systems, such as exception handling and a limited form of coroutines. The conjunction of shared-memory concurrency with internal events allows programs to express dependency among variables reliably, reconciling the control and dataflow reactive styles in a single language.

Categories and Subject Descriptors
D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms
Design, Languages

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

1. INTRODUCTION
An established alternative to C in the field of embedded systems is the family of reactive synchronous languages [3]. Two major styles of synchronous languages have evolved: in the *control–imperative* style, programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style, programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming. Among the control-based languages, Esterel [6] is probably the most famous and has influenced a number of other embedded languages [9, 10, 1], offering a reliable and high-level set of control primitives.

We believe that embedded-system programming can benefit from a new language that reconciles both reactive synchronous styles, while preserving typical C features that programmers are familiarized with, such as shared memory concurrency. Céu [?] is a language targeting embedded systems based on Esterel with some differences that enable new control functionalities, which are the focus of this work:

- A deterministic execution semantics for memory operations allows programs to safely share memory.
- A hierarchical abortion for lines of execution enables dataflow programming.
- A stack-based execution policy for internal events provides advanced control mechanisms, such as exception handling and a limited form of coroutines.

We discuss how Céu achieves a precise control over reactions to the environment and present a formal semantics of the language to highlight its fundamental differences to Esterel.

Céu shares limitations with Esterel and synchronous languages in general: computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay hypothesis [14], and cannot be elegantly implemented. Nonetheless, previous work focusing on Wireless Sensor Networks [?] shows that the expressiveness of Céu is sufficient for embedded applications, with a considerable reduction in source code size in comparison to event-driven code in C. Céu has a small memory footprint, using less than 5 Kbytes of ROM and 100 bytes of RAM for a program with sixteen (simple) flows of execution.

The rest of the paper is organized as follows: Section 2 gives an overview of Céu, exposing its fundamental differences to Esterel. Section 3 shows how to build some advanced control mechanisms using internal events. Section 4 presents a formal semantics for the control primitives of Céu. Section 5 discusses other synchronous languages targeting embedded systems and concludes the paper.

2. OVERVIEW OF CÉU
Céu is a synchronous reactive language based on Esterel [6] 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 any trail at any given time is either reacting to the current event or is awaiting another event; in other words, trails are always

```
// 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.

synchronized at the current (and single) event.

Figure 1 shows the implementations in Esterel and CÉU side-by-side for the following control specification [5]: *"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 is translated almost identically in the two languages (lines 4-9): await and terminate only when both events occur (with a small syntactic mismatch between the '||' and `par/and` constructs). 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 [14]. 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 [**?**, 5].

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 3).

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 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" [5]. 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" [5]. 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"* [5]. 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 [9, 1].

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 [4, 13].

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` compositions (standing for *hierarchical-or*) 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 (discussed in Section 3.2).

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 [4]. 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)

```
// 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.

```
// 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.

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, `_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. In Section 3.2, we take advantage of the lack of recursion to properly describe mutual dependency among trails in parallel.

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 [12].

```
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).

3. ADVANCED CONTROL MECHANISMS

In this section, we explore the presented control primitives of Céu, showing how they enable support for *exceptions* and *dataflow programming* without requiring specific primitives.

3.1 Exception handling

Céu can naturally express different forms of exception mechanisms on top of internal events. In the example of Figure 5, an external entity periodically writes to a file and notifies the program the number of available characters through event `ENTRY` (defined in line 2). The application reacts to every `ENTRY` (lines 9-13), invoking the `read` subroutine (line 11), and then using the filled buffer (line 12). Because this code does not handle failures, it is straight to the point and easy to follow.

Figure 6 defines the `read` subroutine which performs the actual low-level `_read` system call and may fail. The code is placed in parallel so that it can be invoked by the normal application flow. The subroutine awaits requests in a loop (lines 5-10) and may emit exceptions through event `excpt` (lines 7-9).

To handle read exceptions, we use an additional trail in Figure 7 that *strongly* aborts the normal flow on exceptions (line 3). For instance, if the application tries to read an entry and fails, it will behave as follows:

1. Normal flow invokes the read operation (line 11 of Figure 5) and pauses.
   stack: [norm]
2. Read operation awakes (line 6 of Figure 6), throws an exception (line 8), and pauses.
   stack: [norm, read]
3. Exception handler awakes (line 3 of Figure 7) and terminates the `par/or`, aborting the normal behavior and terminating the program.
   stack: []

The exception handler (line 3 of Figure 7) can effectively abort the stacked continuation (line 12 of Figure 5), avoiding the invalid access to `buf`.

This mechanism can also support resumption if the exception handler does not terminate its surrounding `par/or` (line 3 of Figure 7). For instance, the new handler of Figure 8

```
1  // DECLARATIONS
2  input int ENTRY;
3  var _FILE* f = <...>;   // file handler
4  var char[10] buf;       // current entry
5  event int read;
6  event void excpt;
7
8  // NORMAL FLOW
9  loop do
10     var int n = await ENTRY;
11     emit read => n;     // calls ‘read’
12     _printf("line: %s\n", buf);
13 end
```

Figure 5: Normal flow to read file entries.

```
1  <...>           // DECLARATIONS (previous code)
2  par/or do
3     <...>        // NORMAL FLOW (previous code)
4  with
5     loop do      // READ subroutine
6        var int n = await read;
7        if _read(f,buf,n) != n then
8           emit excpt; // throws exception
9        end
10    end
11 end
```

Figure 6: Low-level `read` operation is placed in parallel with the normal flow.

```
<...>       // DECLARATIONS
par/or do
   await excpt; // catches exceptions
with
   <...>    // NORMAL FLOW
with
   <...>    // READ subroutine (throw exceptions)
end
```

Figure 7: Exceptions are caught with a `par/or` that strongly aborts the normal flow.

```
1  <...>       // DECLARATIONS
2  par/or do
3     loop do
4        await excpt; // catch exceptions
5        buf = <...>; // assigns a default
6     end
7  with
8     <...>    // NORMAL FLOW
9  with
10    <...>    // READ subroutine (throw exceptions)
11 end
```

Figure 8: Exception handling with resumption.

catches exceptions in a loop (lines 3-6) and fallbacks to a default string (line 5). The program now behaves as follows (steps 1-2 are the same):

3. Exception handler awakes (line 4 of Figure 8), assigns a default string to `buf` (line 5), and awaits the next exception (line 4).
   stack: [norm, read]
4. Read subroutine resumes (line 8 of Figure 6), and awaits the next call (line 6).
   stack: [norm]
5. Read call resumes (line 11 of Figure 5), and uses `buf` normally (line 12), as if no exceptions had occurred.
   stack: []

Note that throughout the example, the normal flow of Figure 5 (lines 9-13) remains unchanged, with all machinery to handle exceptions around it. With some syntactic sugar these exception mechanisms could be exposed in a higher level to developers.

3.2 Dataflow programming

Reactive dataflow programming provides a declarative style to express dependency relationships among data. Figure 9 shows the dependency graph for the reactive expression `E < E+1`, which should always yield *true*. CÉU can express data dependency relying on `par/hor` compositions and internal events to address two common subtleties in this context: *glitches* and *cyclic dependencies* [2].

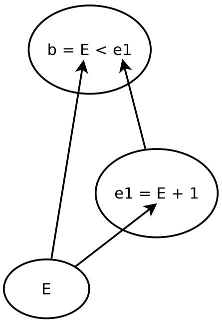
A glitch is a situation in which a dependency graph is updated in an inconsistent order. It is usually avoided by traversing the graph in topological order [7, 2]. In a glitch-free implementation, when `E` changes, `e1` should be updated before `b` to avoid yielding `false` (because `b` also depends on `e1`). The code in the right of the graph implements it in CÉU. The first trail (lines 4-13) updates and signals `b` whenever either `E` or `e1` changes. The second trail (lines 15-19) updates and signals `e1` whenever `E` changes. The `par/hor` (lines 6-11) ensures that `b` is only updated (in line 12) after `e1` and `E` (in lines 8 and 10). The program behavior for a reaction to `E=>1` (which should awake lines 8 and 17) is the following:

1. Line 8 awakes and assigns `v1=1`. (The `par/hor` cannot rejoin yet, allowing other trails to react.)
2. Line 17 awakes, emits `e1=>2`, and pauses.
3. Line 10 awakes and assigns `v2=2`. (The `par/hor` still hangs until the program blocks.)
4. Line 18 resumes, loops, and awaits the next occurrence of `E`.
5. Now that the program cannot advance, the `par/hor` rejoins and correctly emits `b=>1` (i.e., `v1=1 < v2=2`).

Note that the described behavior does not depend on the order the trails are defined in the source code. The `par/hor` is fundamental to avoid the abortion of the composition before both sides have the chance to awake, ensuring the update to `e1` in line 10.

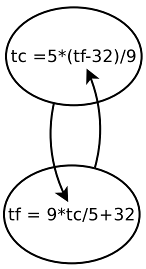
Figure 10 shows a mutual conversion for temperatures in Celsius and Fahrenheit, so that whenever the value in one unit is set, the other is automatically recalculated (a problem proposed in [2]). Mutual dependency is another known issue in dataflow languages, usually requiring the placement of a specific delay operator to avoid runtime cycles [7, 15]. However, an explicit delay is somewhat *ad hoc* because it splits an internal dependency problem across two reactions to the environment. CÉU relies on the stack-based execution for internal events to avoid runtime cycles. The code in the right of the Figure 10 implements the conversion formula in CÉU. We first define the `tc` and `tf` events to signal temperature changes (line 1). Then, we create the 1st and 2nd trails to await for changes and mutually update the temperatures (lines 3-6 and 8-11). The third trail (lines 13-14) signals a temperature change and the program behaves as follows:

1. 3rd trail signals `tc=>0` (line 14) and pauses.
   stack: [3rd]
2. 1st trail awakes (line 4), signals `tf=>32` (line 5), and pauses.
   stack: [3rd,1st]




```
input int E;
event int b, e1;
par/or do
  // b = E < E + 1
  loop do
    var int v1=0,v2=1;
    par/hor do
      v1 = await E;
    with
      v2 = await e1;
    end
    emit b => v1 < v2;
  end
with
  // s1 = s + 1
  loop do
    var int v = await E;
    emit e1 => v + 1;
  end
end
```

Figure 9: Glitch avoidance in Céu with a `par/hor`.




```
1  event int tc, tf;
2  par/or do
3      loop do          // 1st trail
4          var int v = await tc;
5          emit tf => (9 * v / 5 + 32);
6      end
7  with
8      loop do          // 2nd trail
9          var int v = await tf;
10         emit tc => (5 * (v−32) / 9);
11     end
12 with
13     <...>            // 3rd trail
14     emit tc => 0;
15 end
```

Figure 10: A dataflow program with mutual dependency.

3. 2nd trail awakes (line 9), signals `tc=>0` (line 10), and pauses.
   stack: [3rd,1st,2nd]
4. no trails are awaiting `tc` (1st trail is paused at line 5, breaking the cycle), so 2nd trail (on top of the stack) resumes, loops, and awaits `tf` again.
   stack: [3rd,1st]
5. 1st trail resumes, loops, and awaits `tc` again (line 4).
   stack: [3rd]
6. 3rd trail resumes with all dependencies resolved and terminates the program.
   stack: []

As seen in step 4, the second `emit tc=>0` (line 10) is ignored by the 1st trail which is stacked in the reaction to the first `emit tc=>0` (line 14). This way, the stack-based execution for internal events can unambiguously express mutual dependencies. An actual application would run the dependency code in 1st and 2nd trails in parallel and invoke `await` and `emit` on the events `tc` and `tf` (as exemplified in lines 13-14).

4. THE SEMANTICS OF CÉU

In this section, we present a formal semantics of CÉU focusing on the particular control aspects of the language. The semantics specifies a deterministic order for memory operations and relies on a explicit stack to dispatch internal events. It also ensures that a reaction becomes blocked before aborting a `par/hor` composition.

Figure 11 shows a reduced syntax of CÉU. The $mem(id)$

```
                            // 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
    | mem(id) ? p : p       (conditional)
    | p ; p                 (sequence)
    | loop p                (repetition)
    | p and p               (par/and)
    | p or p                (par/or)
    | p hor p               (par/hor)
                            // derived by semantic rules
    | awaiting(id,n)        (awaiting 'id' since seqno 'n')
    | emitting(n)           (emitting on stack level 'n')
    | p @ loop p            (unwinded loop)
```

Figure 11: Reduced syntax of Céu.

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 innocuous mem expressions (it can be thought as a synonym for $mem(\epsilon)$, where ϵ is an unused identifier). All other expressions map to their counterparts in the concrete language.

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

$$\begin{aligned} S, S' &\in id^* && (sequence\ of\ event\ identifiers: [id_{top}, ..., id_1]) \\ p, p' &\in P && (as\ described\ in\ the\ syntax\ above) \\ n &\in \mathbb{N} && (univocally\ identifies\ a\ reaction\ chain) \end{aligned}$$

At the beginning of a reaction chain, the stack is initialized with the special η event and the occurring external event ext ($S = [\eta, ext]$), but $emit$ expressions may push new events on top of it (we discuss how they are popped further). The event η is used as a special marker to check for and resume pending hor expressions before terminating the reaction.

We describe the 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. Figure 12 shows the transitions rules for the complete semantics of CÉU.

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 size), and pushes the referred event on the stack (in rule **emit**). With the new stack level $|s : S|$ after an *emit*, the resulting $emitting(|S|)$ cannot transit yet, as rule **emitting** expects its parameter $|S|$ 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. 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, because a given expression can execute only once within a reaction (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. 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**). Rule **loop-brk** escapes the enclosing loop, transforming everything into a *nop*.

The rules with the **par** prefix are valid for all *and*/*or*/*hor* compositions (substituting the *par* in the rules for each of them). The difference between the three parallel compositions consists only in how to deal with one of the sides terminating. The rules **par-adv1** and **par-adv2** force the transitions on the left branch p to occur before transitions on the right branch q. The deterministic behavior of the semantics relies on the *isBlocked* predicate, defined in Figure 13 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 rules **par-brk1** and **par-brk2** deal with a *break* in each of the parallel sides. A *break* terminates the whole composition to escape the innermost loop (*strongly aborting* the other side).

For an *and* composition, if one of the sides terminates, the composition is simply substituted by the other side, as both sides are required to terminate (rules **and-nop1** and **and-nop2**). For a parallel *or*, reaching a *nop* in either of the sides should immediately terminate the composition (rules **or-nop1** and **or-nop2**). However, for a parallel *hor* it is not enough that one of the sides terminates, as the other should still be allowed to react. The rules **hor-nop1** and **hor-nop2** ensure, first, that a composition rejoins only after no transitions are possible in either sides, and second, that rejoins happen from inside out, i.e., that nested compositions rejoin before outer compositions. The first condition is achieved by only allowing transitions with η at the top of the stack, when the program is guaranteed to be blocked. For the second condition, we check if there is a pending nested *hor*, forcing it to transit before (via rules **par-adv1** or **par-adv2**).

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* ex-

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\langle S,(break\ par\ q)\rangle \xrightarrow[n]{} \langle S,break\rangle \qquad \textbf{(par-brk1)}$$

$$\frac{isBlocked(n,S,p)}{\langle S,(p\ par\ break)\rangle \xrightarrow[n]{} \langle S,break\rangle} \qquad \textbf{(par-brk2)}$$

$$\langle S,(mem(id)\ and\ q)\rangle \xrightarrow[n]{} \langle S,q\rangle \qquad \textbf{(and-nop1)}$$

$$\langle S,(p\ and\ mem(id))\rangle \xrightarrow[n]{} \langle S,p\rangle \qquad \textbf{(and-nop2)}$$

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

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

$$\frac{q\neq(a\ hor\ b)\ \vee\ (a\neq mem(v)\ \wedge\ b\neq mem(v))}{\langle[\eta],(mem(v)\ hor\ q)\rangle \xrightarrow[n]{} \langle[\eta],nop\rangle} \qquad \textbf{(hor-nop1)}$$

$$\frac{p\neq(a\ hor\ b)\ \vee\ (a\neq mem(v)\ \wedge\ b\neq mem(v))}{\langle[\eta],(p\ hor\ mem(v))\rangle \xrightarrow[n]{} \langle[\eta],mem(v)\rangle} \qquad \textbf{(hor-nop2)}$$

Figure 12: The semantics of Céu.

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

Figure 13: The recursive predicate *isBlocked*.

pressions should resume in the ongoing reaction, once their lower stack indexes are restored (see rule **emit**). Therefore, we define another relation to pop the stack if the program becomes blocked (behaving as the previous relation, otherwise):

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

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

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

5. RELATED WORK AND CONCLUSION

With respect to control-based languages for embedded systems, a number of synchronous alternatives to low-level event-driven systems have appeared [8, 9, 10, 1]. Protothreads [8] offer predictable and lightweight multi-threading with shared-memory concurrency, but lack thread composition and abortion (as described in Section 2.2). OSM [10] provides parallel state machines with a formal and mature synchronous model that supports thread composition and abortion. However, although machines can share memory, the execution order for operations among them is non-deterministic. Also, describing sequential code across reactions is not straightforward with state machines [10] (in comparison to CÉU's `await` statement). Other related synchronous languages [9, 1] also rely on a deterministic scheduler for safe memory sharing, but do not differ from Esterel regarding event handling and thread composition.

Functional Reactive Programming (FRP) adapts modern functional languages to the reactive dataflow style [16]. In particular, Flask [11] shows that dataflow languages can also target constrained systems. Dataflow in CÉU is limited to static relationships only, and is less abstract in comparison to FRP.

As a descendant of Esterel, CÉU achieves a high degree of reliability for constrained embedded systems, while also embracing practical aspects, such as supporting shared-memory concurrency. CÉU introduces a stack-based execution policy for internal events, expanding its expressiveness for describing exceptions and dataflow programming. As far as we know, CÉU is the first language to reconcile the control and dataflow reactive styles.

6. REFERENCES

[1] S. Andalam et al. Predictable multithreading of embedded applications using PRET-C. In *Proceeding of MEMOCODE'10*, pages 159–168. IEEE, 2010.

[2] E. Bainomugisha et al. A survey on reactive programming. *ACM Computing Surveys*, 2012.

[3] A. Benveniste et al. The synchronous languages twelve years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, Jan 2003.

[4] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993.

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

[6] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.

[7] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of ESOP'06*, pages 294–308, 2006.

[8] Dunkels et al. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*, pages 29–42. ACM, 2006.

[9] M. Karpinski and V. Cahill. High-level application development is realistic for wireless sensor networks. In *Proceedings of SECON'07*, pages 610–619, 2007.

[10] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of IPSN '05*, pages 45–52, April 2005.

[11] Mainland et al. Flask: staged functional programming for sensor networks. In *Proceeding of ICFP'08*, pages 335–346, New York, NY, USA, 2008. ACM.

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

[13] ORACLE. Java thread primitive deprecation. `http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`, 2011.

[14] D. Potop-Butucaru et al. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. 2005.

[15] F. Sant'Anna and R. Ierusalimschy. LuaGravity, a reactive language based on implicit invocation. In *Proceedings of SBLP'09*, pages 89–102, 2009.

[16] Z. Wan and P. Hudak. Functional reactive programming from first principles. *SIGPLAN Notices*, 35(5):242–252, 2000.