Reconciling Control and Dataflow Reactivity in Embedded Systems

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 prioritizes safety aspects for the development of reliable applications targeting highly constrained platforms. Featuring a deterministic semantics, Céu provides safe shared memory concurrency even when multiple lines of execution are active at the same time. 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. The conjunction of internal events with shared-memory concurrency allows programs to express dependency among variables, which reconciles the control and dataflow reactive programming 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, Reliability

Keywords

Concurrency, Determinism, Embedded Systems, Safety, Static Analysis, Synchronous

1. INTRODUCTION

Embedded systems are usually designed with safety and real-time requirements under constrained hardware platforms. They are essentially reactive and interact permanently with the surrounding environment through input and output devices (e.g. buttons, timers, transceivers, etc.).

Software for embedded systems is usually developed in *C*, even though concurrent programming systems offering cooperative or preemptive multi-threading lack effective safety guarantees, being subject to unbounded execution [8], race conditions and deadlocks [11].

An established alternative to *C* in the field of safety-critical embedded systems is the family of reactive synchronous languages [2]. Two major styles of synchronous languages have evolved: in the *control–imperative* style (e.g. Esterel [5]), programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style (e.g. Lustre [10]), programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming.

We believe that embedded systems programming can benefit from a new language that reconciles both reactive synchronous styles, while preserving typical *C* features that programmers are familiarized, such as shared memory concurrency. Céu [14] is a Esterel-based reactive programming language that provides a reliable yet powerful programming environment for embedded systems with some fundamental distinctions. In this work we focus on the differences from Céu to Esterel which introduce new programming functionalities:

- The execution order for memory operations rely on a deterministic semantics, allowing programs to safely share memory.
- Internal events follow a stack-based execution policy, providing new advanced control mechanisms, such as exception handling, and dataflow programming.

We present a formal semantics for the control primitives of Céu and discuss how they avoid *glitches* and *cyclic dependencies* in dataflow programming [1].

Céu shares the same limitations with Esterel and synchronous languages in general: computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay hypothesis [13], and cannot be elegantly implemented.

Nonetheless, previous work focusing on Wireless Sensor Networks [14] shows that the expressiveness of Céu is sufficient for implementing a wide range of applications (e.g. network protocols and a radio driver), with a considerable reduction in code complexity and a small increase in resource usage in comparison to the state-of-the-art [9]. The memory footprint of Céu is around 2 Kbytes of ROM and 50 bytes of RAM. A program with sixteen lines of execution (with minimum bodies) incur extra 270 bytes of ROM and 60 bytes of RAM.

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

```
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: "ABRO" example in Esterel and Céu.

The rest of the paper is organized as follows: Section 2 describes the main differences between Céu and Esterel. Section 3 shows how to implement some advanced control-flow mechanisms on top of Céu internal events. Section 4 shows how to implement some advanced control-flow mechanisms on top of Céu internal events. Section 5 compares Céu to existing synchronous and asynchronous languages for embedded systems. Section 6 concludes the paper and makes final remarks.

2. OVERVIEW OF CÉU AND ESTEREL

Céu is a synchronous reactive language based on Esterel [6] with support for multiple 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 synchronized at the current (and single) event.

Figure 1 shows side-by-side the implementations in Esterel and Céu for the "ABRO" example with the following specification [4]: *"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 line of the specification is almost identical in the two implementations, with a small syntactic mismatch between the '||' operator and the `par/and` construct (lines 2-7 in Esterel, and 3-8 in Céu). The reset behavior of Esterel uses a `loop-each` syntactic sugar, which expands to the more primitive `abort-when` and serves as the same purpose of Céu's `par/or` (to be discussed in Section 2.2). In both cases, the occurrence of the event `R` aborts the awaiting statements and restarts the `loop`.

The languages have a strong imperative flavor, with explicit control flow through sequences, loops, and also assignments. Being designed for control-intensive applications, they provide additional support for concurrent lines of execution and broadcast communication through events. Esterel and Céu also rely on a *multiform notion of time*, in which programs advance in discrete and subsequent reactions to external *signals* (*events* in Céu). Internal computations within a reaction (e.g. expressions, assignments, and native calls) are considered to take no time in accordance with the synchronous (or *zero-delay*) hypothesis [13]. The `await` statements are the only that halt a running reaction and allow a program to advance. To ensure that reactions run in bounded time, loops must contain at least one `await` statement in all possible paths [14, 4].

2.1 External reactions and determinism

In Esterel, an external reaction can carry simultaneous signals, while in Céu, a single event defines a reaction. The notion of time in Esterel is similar to that of digital circuits, in which multiple wires (signals) can be queried for its status (*present* or *absent*) on each clock tick. Céu more closely reflects event-driven programming, in which occurring events are sequentially and uninterruptedly handled by the program [**?**, **?**]. Note that even with single-event rule there is still concurrency in Céu given that multiple lines of execution may react to the same event.

Another difference between Esterel and Céu is on their definition for determinism: Esterel is deterministic with respect to reactive control, i.e., "the same sequence of inputs always produces the same sequence of outputs" [4]. However, the order of execution for side-effect operations within a reaction is non-deterministic: "if there is no control dependency and no signal dependency, as in "`call f1() || call f2()`", the order is unspecified and it would be an error to rely on it" [4]. 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 text and run to completion (i.e. `_P1` executes first)[1]. This way, Céu is deterministic also with respect to the order of execution of side effects within a reaction, which is formally described in Section 4.

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 (discussed in Section 2.2), and also ensures a reproducible execution for shared-memory programs. For software development, we believe that determinism for side-effects is usually beneficial and is a design decision that makes sense for Céu.

2.2 Thread abortion

The introductory example of Figure 1 illustrates how awaiting lines of execution can be aborted without being tweaked with synchronization primitives. It is known that traditional (asynchronous) multi-threaded languages cannot express thread termination safely [3, 12].

The code fragments of Figure 2 illustrate thread abortion in Esterel and Céu. It is not clear in the example in Esterel if the call to `f` should execute or not, given that the body and abortion event are the same. For this reason, Esterel provides the *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 the trails within a `par/or`, e.g., in the example the first trail strongly aborts the second and the call to `_f` never executes.

The `par/hor` (standing for *hierarchical-or*) provides an alternative that schedules both sides before aborting the compo-

[1] Céu can call native functions by prefixing names with an underscore.

sition. In the rightmost example of Figure 2, both `_g` and `_f` (in this order) will execute in reaction to `S`. Hierarchical traversal is fundamental for dataflow programming, as we discuss in Section 3.2. Note that in contrast with a `par/and`, a `par/hor` rejoins on the same reaction one of its trails terminates, but it gives the opportunity for the other side to also react.

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

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

```
// CEU (hor)
par/hor do
   await S;
   _g();
with
   await S;
   _f();
end
```

Figure 2: Thread abortion in Esterel and Céu.

2.3 Internal events

Esterel makes no semantic distinctions between internal and external signals [3], both having only the notion of presence or absence. In Céu, however, internal and external events behave differently:

- External events can be emitted only by the environment, while internal events only by the program.
- A single external event can be active at a time, while multiple internal events can coexist within a reaction.
- External events are handled in a queue, while internal events follow a stacked execution policy (like subroutine calls in typical programming languages).

Figure 3 illustrates the use of internal signals (events) in Esterel and Céu. For the Esterel version, there is no specific execution order for the `printf` calls. For the Céu version, on the occurrence of the event `START` the program behaves as follows (with the stack in emphasis):

1. 1st trail awakes, emits `a`, and pauses.
   stack: [1st]
2. 2nd trail awakes, emits `b`, and pauses.
   stack: [1st,2nd]
3. 3rd trail awakes, prints 1, and terminates.
   stack: [1st,2nd]
4. 2nd trail (on top of the stack) resumes, prints 2, and terminates.
   stack: [1st]
5. 1st trail resumes, prints 3, and terminates.
   stack: []
6. All trails have terminated, so the `par/and` rejoins, and the program also terminates;

Internal events bring support for a limited form of subroutines, as illustrated int Figure 4. The subroutine `inc` is defined as a loop (lines 3-6) that continuously awaits its identifying event (line 4) and increments the value passed as reference (lines 5). A trail in parallel (lines 8-11) invokes the subroutine in reaction to event `A` through an `emit`. Given the stacked execution for internal events, the calling trail pauses (line 10) and the subroutine awakes (line 4). Only after the subroutine "returns" (after the next `await` in line 4), that the calling trail resumes and passes the assertion test.

This form of subroutines has some significant limitations:

```
// ESTEREL
input START;
signal A, B;
[[
   await START;
   emit A;
   call printf("3");
||
   await A;
   emit B;
   call printf("2");
||
   await B;
   call printf("1");
]]
```

```
// CEU
input void START;
event void a, b;
par/and do
   await START;
   emit a;
   _printf("3");
with
   await a;
   emit b;
   _printf("2");
with
   await b;
   _printf("1");
end
```

Figure 3: Internal signals (events) in Esterel and Céu.

```
1  event int* inc; // function '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: The subroutine `inc` is defined in a loop, in parallel with the application.

***Single instance*:** Calls to a running subroutine have no effect. For instance, if the subroutine in the example waits for another event before the loop, it cannot serve new requests.

***Single calling*:** Further calls to a subroutine in a reaction also have no effect. To avoid unbounded execution, an `await` statement must be awaiting before a reaction starts.

***No recursion*:** Recursive calls to a subroutine also have no effect. For the same reason of the *single instance* property, a trail cannot be awaiting itself while running and the recursive call is ignored.

***No concurrency*:** If two trails in parallel try to call the same subroutine, only the first trail takes effect (based on deterministic scheduling). The second call fails on the *single calling* property.

Céu provides no support for standard functions for a number of reasons:

- The interaction with other Céu control primitives is not obvious (e.g., executing an *await* or a *par/or* inside a function).
- They would still be restricted in some ways given the embedded context (e.g. no recursion or closures).
- Programs can always recur to *C* functions for low-level operations.

```
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; // throw exception
9        end
10    end
11 end
```

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

In Section 3.2, we show that we can even take advantage of non-recursive subroutines to properly describe mutual dependency among trails in parallel.

3. ADVANCED CONTROL MECHANISMS

In this section, we explore the stacked execution for internal events in Céu, demonstrating how it enables to derive support for *exceptions* and *dataflow programming* without requiring specific primitives.

3.1 Exception handling

Céu can naturally express different forms of exception handling on top of internal events without a specific construct. As the illustrative example in Figure 5, suppose an external entity periodically writes to a file and notifies the program through the event `ENTRY` carrying the number of available characters (line 2). The normal flow is to wait `ENTRY` requests in a loop (lines 8-13) and request a `read` operation (line 11). The low-level file operation `read` is defined as an internal event working as a subroutine that fills the variable `buf` (shown later). Relying on the stack-based execution for the `emit`, the code that manipulates the `buf` (line 12) is guaranteed to execute after it is filled. Because this code does not handle failures, it is straight to the point and easy to follow.

Figure 6 defines the subroutine that performs the actual low-level `_read` system call, which is placed in parallel with the normal flow. The subroutine awaits requests in a loop (lines 5-10) and may emit exceptions through event `excpt` on any error (lines 10-12).

Finally, to handle read exceptions, we use an additional `par/or` in Figure 7 that *strongly* aborts the normal flow on

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

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

```
...
  par/or do
     loop do
        await excpt; // catch exceptions
        buf = <...>; // assigns a default
     end
  with
     <...> // NORMAL FLOW
  end
...
```

Figure 8: Exception handling with resumption.

any exception. Now, suppose the normal flow tries to read a string and fails. The program behaves as follows (with the stack in emphasis):

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 4 of Figure 7) and terminates the `par/or`, aborting the normal behavior and terminating the program.
   stack: []

This mechanism can also support resumption if the exception handler does not terminate its surrounding `par/or`. For instance, the new handler of Figure 8 catch exceptions in a loop and provides a default string to the normal flow. The program 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 6 of Figure 6), and awaits the next call.
   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 remained unchanged, with all machinery to handle exceptions placed around it. With some syntactic sugar these exception mechanisms could be exposed in a higher level to developers.

3.2 Dataflow programming

```
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    var int v = await tf; // 3rd trail
14    <...>
15 with
16    <...> // 4th trail
17    emit tc => 0;
18 end
```

Figure 9: A dataflow program with mutual dependency.

Reactive dataflow programming provides a declarative style to express dependency relationships among data [1]. Two issues

Mutual dependency is a known issue in dataflow languages, requiring the explicit placement of a specific delay operator to avoid runtime cycles [7, 15]. This solution is somewhat *ad hoc* and splits an internal dependency problem across two reactions to the environment. Céu can safely express cyclic dependencies, given that reactions are guaranteed to terminate.

As an example, the program in Figure 9 applies the temperature conversion formula between Celsius and Fahrenheit, so that whenever the value in one unit is set, the other is automatically recalculated (a problem proposed in [1]).

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 and fourth trails respectively react to (lines 13-14) and signal temperature changes (lines 16-17). The program behaves as follows (with the stack in emphasis):

1. 4th trail signals `tc=>0` and pauses.
   stack: [4th]
2. 1st trail awakes, signals `tf=>32`, and pauses.
   stack: [4th,1st]
3. 2nd trail awakes, signals `tc=>0`, and pauses.
   stack: [3rd,1st,2nd]
4. no trails are awaiting `tc` (1st trail is paused), so 2nd trail (on top of the stack) resumes, loops, and awaits `tf` again.
   stack: [3rd,1st]
5. 3rd trail also awakes from `tf=>32`, uses the updated value, and eventually halts.
   stack: [3rd,1st]
6. 1st trail resumes, loops, and awaits `tc` again.
   stack: [3rd]
7. 3rd trail resumes *with all dependencies resolved* and terminates the program.
   stack: []

The stack-based execution for internal events can unambiguously express mutual dependencies, as seen in step 4, when the `emit tc=>0` of line 10 is ignored by the 1st trail which is already reacting to the first `emit tc=>0` of line 17. The stack also ensures that an `emit` that triggers nested dependencies only resumes after the relationships are completely updated.

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

4. THE SEMANTICS OF CÉU

We present a formal semantics of Céu focusing on the control aspects of the language. The syntax for a subset of Céu that is sufficient to describe all semantic peculiarities of the language is as follows:

```
// primary statements     // description
nop(v)                    (constant value)
mem                       (any memory access)
await(e)                  (await event ‘e’)
emit(e,v)                 (emit event ‘e’ passing ‘v’)
break                     (loop escape)

// compound statements
mem ? p : q               (conditional)
p ; q                     (sequence)
loop p                    (repetition)
p and q                   (par/and)
p or q                    (par/or)
p hor q                   (par/hor)

// derived by semantic rules
awaiting(e,m)             (awaiting ‘e’ since seqno ‘m’)
emitting(t)               (emitting on stack level ‘t’)
p @ loop p                (unwinded loop)
```

A *nop* represents a terminated computation associated with a constant value. The *mem* primitive represents all memory accesses, assignments, and *C* function calls. The semantic rules generate three statements that the programmer cannot write: the *awaiting* avoids that an *await* awakes during the same reaction it is reached; the *emitting* represents the continuation of an *emit* and enables the desired stacked behavior for internal events; a *loop* is expanded with the special `‘@’` separator (instead of `‘;’`) to properly bind *break* statements inside p to the enclosing loop.

We use a *small-step* structural operational semantics to formally describe a *reaction chain* in Céu, i.e., how a program behaves in reaction to a single external event. The semantics require an explicit stack to properly nest the emission of internal events. A complete reaction chain in Céu is formalized as follows:

$$\langle S, s, p\rangle \xrightarrow[n]{*} pop(p,\ \langle S', s', p'\rangle)$$

At a given external sequence n, a program p with a stack of events S with top s continuously progress until no transitions are possible. After every transition, the top of the stack is popped if the program becomes blocked.

At the beginning of each reaction, the external sequence number n is incremented and the stack is initialized with the new external event, i.e., $s = 1$ and $S = \{1 \mapsto ext\}$. The top of the stack represents the single event being handled at a time: it is initialized to the external event, but *emit*

statements can push new events on top of it. When no transitions are possible in the current level, the stack is popped and the previous event is reactivated. The reaction chain terminates when the level 0 is popped from the stack, signaling that the original external event has been completely handled and no trails are pending.

To describe the full execution of a program with reaction chains in sequence, we need multiple "invocations" of the operational semantics, i.e.:

$$\langle \{1 \mapsto e1\}, 1, p \rangle \Longrightarrow_{1} \langle \{\}, -1, p' \rangle$$
$$\langle \{1 \mapsto e2\}, 1, p' \rangle \Longrightarrow_{2} \langle \{\}, -1, p'' \rangle$$
$$\ldots$$

Each invocation starts with the 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.

Figure 10 shows the transitions rules for the complete semantics of Céu. At any time, at most one transition is possible. This way, there is a single order of execution for side effects (represented by `mem` operations), and reaction chains are always deterministic.

A *mem* reduces to a *nop* in a single step (rule **mem**), representing the (bounded) execution of a side-effect operation. Although we opted not to formalize side effects, the actual values yielded by *mem* operations are required in conditionals (e.g., the value of a variable or the result of a *C* function call). (In the rules that an yielded *nop* cannot appear in a conditional, we omit its generated value, i.e., we use *nop* instead of $nop(v)$.)

An *await* is simply transformed into an *awaiting* that remembers the current external sequence number n (rule å). An *awaiting* can only transit to a *nop* (rule **awaiting**) if its referred event matches the top of the stack ($S(s)$) and its sequence number is smaller than the current one ($m < n$). An *emit* transits to an *emitting* and stacks its referred event (rule **emit**). With the new stack level $s+1$, the $emitting(s)$ cannot transit, as rule **emitting** expects its parameter to match the current stack level. This trick enforces the desired stack-based semantics for internal events.

The function *pop* (called after every transition) removes the top of the stack when the rules cannot make a transition, allowing that previously stacked *emitting* operations progress:

$$pop(p,\ \langle S, s, p' \rangle) = \begin{cases} \langle S, s-1, p' \rangle, & p = p' \\ \langle S, s, p' \rangle, & p \neq p' \end{cases}$$

The rules for conditionals and sequences are straightforward (**if-**$*$ and **seq-**$*$). Given that the semantics focus on control, note that rules **if-true** and **if-false** are the only to query $nop(v)$ values.

The rules for loops are analogous to sequences, but use **`'@'`** as separators to properly bind a break to its enclosing loop. When a program first encounters a *loop*, it first expands its body in sequence with itself (rule **loop-expd**). Rules **loop-**

$$\langle S, s, mem \rangle \underset{n}{\longrightarrow} \langle S, s, nop(v) \rangle, v \in \mathbb{N} \quad \textbf{(mem)}$$

$$\langle S, s, await(e) \rangle \underset{n}{\longrightarrow} \langle S, s, awaiting(e, n) \rangle \quad \textbf{(await)}$$

$$\langle S, s, awaiting(S(s), m) \rangle \underset{n}{\longrightarrow} \langle S, s, nop \rangle, m < n \quad \textbf{(awaiting)}$$

$$\langle S, s, emit(e, v) \rangle \underset{n}{\longrightarrow} \langle S \uplus \{(s+1) \mapsto e\}, s+1, emitting(s) \rangle \quad \textbf{(emit)}$$

$$\langle S, s, emitting(s) \rangle \underset{n}{\longrightarrow} \langle S, s, nop \rangle \quad \textbf{(emitting)}$$

$$\frac{m \underset{n}{\longrightarrow} m'}{\langle S, s, (m\ ?\ p\ :\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, (m'\ ?\ p\ :\ q) \rangle} \quad \textbf{(if-adv)}$$

$$\langle S, s, (nop(v)\ ?\ p\ :\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, p \rangle\ ,\quad (v \neq 0) \quad \textbf{(if-true)}$$

$$\langle S, s, (nop(0)\ ?\ p\ :\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, q \rangle \quad \textbf{(if-false)}$$

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

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

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

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

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

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

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

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

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

$$\langle S, s, (break\ par\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, break \rangle \quad \textbf{(par-brk1)}$$

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

$$\langle S, s, (nop\ and\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, q \rangle \quad \textbf{(and-nop1)}$$

$$\langle S, s, (p\ and\ nop) \rangle \underset{n}{\longrightarrow} \langle S, s, p \rangle \quad \textbf{(and-nop2)}$$

$$\langle S, s, (nop\ or\ q) \rangle \underset{n}{\longrightarrow} \langle S, s, nop \rangle \quad \textbf{(or-nop1)}$$

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

$$\frac{q \neq (a\ hor\ b)\ \vee\ (a \neq nop\ \wedge\ b \neq nop)}{\langle S, 0, (nop\ hor\ q) \rangle \underset{n}{\longrightarrow} \langle S, 1, nop \rangle} \quad \textbf{(hor-nop1)}$$

$$\frac{p \neq (a\ hor\ b)\ \vee\ (a \neq nop\ \wedge\ b \neq nop)}{\langle S, 0, (p\ hor\ nop) \rangle \underset{n}{\longrightarrow} \langle S, 1, nop \rangle} \quad \textbf{(hor-nop2)}$$

Figure 10: The semantics of Céu.

adv and **loop-nop** are similar to rules **seq-adv** and **seq-nop**, advancing the loop until it reaches a *nop*. 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 with the **par** prefix are valid for all *and*/*or*/*hor* compositions (substituting the *par* in the rules for each of them). The transitions on the left branch p are forced to occur before transitions on the right branch q (rules **par-adv1** and **par-adv2**). The deterministic behavior of the semantics relies on the *isBlocked* recursive predicate, which is only true if all branches in parallel are hanged in *awaiting* or *emitting* operations that cannot transit:

$$\begin{aligned}
isBlocked(n, S, s, awaiting(e, m)) &= (e \neq S(s) \ \vee \ m = n) \\
isBlocked(n, S, s, emitting(t)) &= (t \neq s) \\
isBlocked(n, S, s, (p \ ; \ q)) &= isBlocked(n, S, s, p) \\
isBlocked(n, S, s, (p \ @ \ loop \ q)) &= isBlocked(n, S, s, p) \\
isBlocked(n, S, s, (p \ par \ q)) &= isBlocked(n, S, s, p) \ \wedge \\
&\quad\ isBlocked(n, S, s, q) \\
isBlocked(n, S, s, *) &= false \ (all \ others)
\end{aligned}$$

For instance, rule **par-adv2** requires the left branch p to be blocked in order to allow the right transition from q to q'. The rules **par-brk1** and **par-brk2** deal with a *break* in each of the parallel sides, which terminates the whole composition to escape the innermost loop (strongly aborting the other side).

The difference among the three parallel compositions consists in how to deal with one of the sides terminating with a *nop*. For an *and* composition, if one of the sides terminate, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**), as both sides are required to terminate. For a parallel *or*, reaching a *nop* in either of the sides should immediatelly terminate the composition (rules **or-nop1** and **or-nop2**). However, for a parallel *hor* it is not enough that one of the sides reaches a *nop*, as the other should also 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 allowing transitions with $s = 0$ only, when the program is guaranteed to be blocked. For the second condition, we check if a nested *hor* is also pending, forcing that composition to transit first (via rules **par-adv1** or **par-adv2**).

5. RELATED WORK

6. CONCLUSION

7. REFERENCES

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

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

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

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

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[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 *15th European Symposium on Programming*, pages 294–308, 2006. LNCS 3924.

[8] C. Duffy et al. A comprehensive experimental comparison of event driven and multi-threaded sensor node operating systems. *JNW*, 3(3):57–70, 2008.

[9] D. Gay et al. The nesC language: A holistic approach to networked embedded systems. In *PLDI'03*, pages 1–11, 2003.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, September 1991.

[11] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

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

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

[14] F. Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of SenSys'13*. ACM, 2013. to appear.

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