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 targets constrained embedded platforms. Relying on a deterministic semantics, Céu provides safe shared memory among concurrent 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 dataflow programming. 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

Embedded systems are essentially reactive and interact permanently with the surrounding environment through I/O devices (e.g. buttons, timers, transceivers). 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 programming for embedded systems 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 [17] is a Esterel-based programming language targeting embedded systems aiming to address this motivation. In this work, we focus on the differences between Céu and Esterel that enable new programming functionalities:

- A deterministic execution semantics for memory operations allows programs to safely share memory.
- A stack-based execution policy for internal events provides new advanced control mechanisms, such as exception handling, and dataflow programming.

We discuss in detail the differences between Céu and Esterel, in particular, how they avoid *glitches* and *cyclic dependencies* in dataflow programming [1]. We also present a formal semantics for the control primitives of Céu.

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 [16], and cannot be elegantly implemented.

Nonetheless, previous work focusing on Wireless Sensor Networks [17] 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 in comparison to the state-of-the-art [9]. Céu has a small memory footprint, using less than 5 Kbytes of ROM and 100 bytes of RAM for a program with sixteen (simple) lines of execution.

The rest of the paper is organized as follows: Section 2 gives an overview of Céu and Esterel, exposing the fundamental similarities and differences between the two. Section 3 shows how to implement some advanced control-flow mechanisms on top of Céu's internal events. Section 4 presents a formal semantics for the control primitives of Céu. Section 5 compares Céu to existing synchronous languages targeting embedded systems. Section 6 concludes the paper and makes final remarks.

```
// 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
```

Figure 1: The same specification in Esterel and Céu.

2. OVERVIEW OF CÉU AND ESTEREL

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 synchronized at the current (and single) event.

Figure 1 shows the implementations in Esterel and Céu side-by-side for 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 phrase of the specification is almost identical in the two implementations, with a small syntactic mismatch between the '||' and `par/and` constructs. The reset behavior of Esterel uses a `abort-when`, which serves the same purpose of Céu's `par/or` (to be discussed in Section 2.2). In both cases, the occurrence of event `R` aborts the awaiting statements and restarts the `loop`.

Esterel and Céu 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. Programs advance in discrete and subsequent reactions to external *signals* (or *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 hypothesis [16]. The `await` statements are the only 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 required to contain at least one `await` statement in all possible paths [17, 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 their 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 the single-event rule of Céu, there is still concurrency given that multiple lines of execution may react to the same event.

Another difference between Esterel and Céu is on their definitions 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 (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 (discussed in Section 2.2), and ensures a reproducible execution for shared-memory programs. For software development, we believe that deterministic shared-memory concurrency is beneficial, specially considering embedded systems which make extensive use of memory mapped ports for I/O.

2.2 Thread abortion

The introductory example of Figure 1 illustrates how synchronous languages can abort awaiting lines of execution without tweaking them with synchronization primitives. In contrast, it is known that traditional (asynchronous) multi-threaded languages cannot express thread termination safely [3, 15].

The code fragments of Figure 2 show a corner case for thread abortion in Esterel and Céu. Note that it is not clear in the leftmost example in Esterel if the call to `f` should execute or not, 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 the example in the middle, the second trail is strongly aborted and the call to `_f` never executes.

Céu also supports `par/hor` compositions (standing for *hierarchical-or*) which schedules both sides before terminating. This way, 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 the sides terminates, but also allows the other side to run before.

```
// 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.

```
// 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");
]]
```

```
1  // CEU
2  input void START;
3  event void a, b;
4  par/and do
5      await START;
6      emit a;
7      _printf("3");
8  with
9      await a;
10     emit b;
11     _printf("2");
12 with
13     await b;
14     _printf("1");
15 end
```

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

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 [3]. In CÉU, however, internal and external events behave differently:

- External events can only be emitted 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 stack-based execution policy (like subroutine calls in typical programming languages).

Figure 3 illustrates the use of internal signals (events) in Esterel and CÉU. For the version in Esterel, given that there is no control dependency between the calls to `printf`, they may execute in any order on `START`. For the version in CÉU, the occurrence of `START` makes the program behave as follows (with the stack in emphasis):

1. 1st trail awakes (line 5), emits `a`, and pauses.
   stack: [1st]
2. 2nd trail awakes (line 9), emits `b`, and pauses.
   stack: [1st,2nd]
3. 3rd trail awakes (line 13), 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 in 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 (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 only a *single instance* can be active at a time, e.g., if the subroutine in the example waits for another event before the loop, it cannot serve new requests in the meantime. In particular, recursive calls are always ignored as a subroutine cannot be awaiting itself while it is running. On the other hand, this form of subroutines is guaranteed to react in bounded time and can be combined with the other primitives of CÉU, such as parallel compositions. They can also `await`, keeping context information such as locals and the program counter (like coroutines). The fact that they do not require stacks in the underlying implementation is also fundamental considering the context of embedded systems. In Section 3.2, we take advantage of non-recursive subroutines to properly describe mutual dependency among trails in parallel. Note that programs in CÉU can still recur to *C* functions if necessary.

```
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: The subroutine `inc` is defined in a loop, in parallel with the caller.

```
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.

3. ADVANCED CONTROL MECHANISMS

In this section, we explore the stacked execution for internal events in CÉU, showing 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 mechanisms on top of internal events. In the example of Figure 5, an external entity periodically writes to a file and notifies the program through the event `ENTRY` carrying the number of available characters. The application reacts to every `ENTRY` in a loop (lines 8-13), invoking the `read` operation (line 11), and then using the filled buffer (line 12). Note that the file operation `read` is defined as an internal event working as a subroutine, which is expected to fill the variable `buf` with the entry contents. Because this code does not handle failures, it is straight to the point and easy to follow.

```
 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.

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

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

Figure 6 defines the `read` subroutine which performs the actual low-level `_read` system call (which 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 `par/or` in Figure 7 that *strongly* aborts the normal flow on any exception. For instance, if the application tries to read an entry and fails, it will behave 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: []

The exception handler (lines 3-7 of Figure 7) was placed around the normal behavior (line 6), and could effectively abort its execution to avoid the unsafe use of `buf` (line 12 of Figure 5). Note that the original sequential code (lines 8-13 of Figure 5) did not have a single line modified.

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 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):

```
 1  <...>
 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     end
10  <...>
```

Figure 8: Exception handling with resumption.

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 remains 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

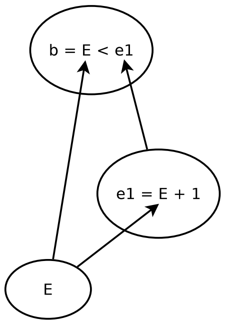
Reactive dataflow programming provides a declarative style to express dependency relationships among data. CÉU can express data dependency relying on `par/hor` compositions and internal events to address two common subtleties in this context: *glitches* and *cyclic dependencies* [1].

A glitch is a situation in which a dependency graph is updated in an inconsistent order and can be avoided by traversing the graph in topological order [7, 1]. Figure 9 shows a graph for the expression `E < E+1`, which should always yield *true*. In a glitch-free implementation, when `E` changes, `e1` should be updated before `b` to avoid an inconsistent state (because `b` also depends on `e1`). The code in the right of the graph implements this "reactive expression" 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 after `e1` and `E`. Follows the behavior of the program for a reaction to `E=>1` (i.e., lines 8 and 17 should awake):

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 fact, the `par/hor` is fundamental to avoid the abortion resulting in missing the update to `e1` in line 10.

Mutual dependency is another known issue in dataflow languages, requiring the explicit placement of a specific delay



```
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.

operator to avoid runtime cycles in some systems [7, 18]. 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. As an example, the program in Figure 10 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. Then, we create the 1st and 2nd trails to await for changes and mutually update the temperatures. The third trail signals a temperature change and the program behaves as follows (with the stack in emphasis):

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]
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 parallel and invoke `await` and `emit` on the events `tc` and `tf`.

4. THE SEMANTICS OF CÉU

We present a formal semantics of CÉU focusing on the control aspects of the language, with a reduced syntax as follows:

```
// primary statements     // description
nop(v)                    (constant value)
mem                       (any memory access)
await(e)                  (await event 'e')
emit(e)                   (emit event 'e')
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 at stack level 't')
p @ loop p                (unwinded loop)
```

A *nop* represents a terminated computation associated with a constant value. A *mem* 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 uses 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 each transition, the top of the stack is popped if the program cannot advance.

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 that level: 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.:

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

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 11 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).

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 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 pushes its referred event into the stack (rule **emit**). With the new stack level $s+1$, the $emitting(s)$ in rule **emitting** cannot transit, as it expects the parameter s to match the current stack level. This trick enforces the desired stack-based behavior for internal events.

The function *pop* (called after each 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-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

$$\langle S,s,mem\rangle \xrightarrow[n]{} \langle S,s,nop(v)\rangle, v\in\mathbb{N} \qquad \textbf{(mem)}$$

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

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

$$\langle S,s,emit(e)\rangle \xrightarrow[n]{} \langle S \uplus \{(s+1)\mapsto e\}, s+1, emitting(s)\rangle \qquad \textbf{(emit)}$$

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

$$\frac{m \xrightarrow[n]{} m'}{\langle S,s,(m\ ?\ p\ :\ q)\rangle \xrightarrow[n]{} \langle S,s,(m'\ ?\ p\ :\ q)\rangle} \qquad \textbf{(if-adv)}$$

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

$$\langle S,s,(nop(0)\ ?\ p\ :\ q)\rangle \xrightarrow[n]{} \langle S,s,q\rangle \qquad \textbf{(if-false)}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{q\neq(a\ hor\ b)\ \vee\ (a\neq nop\ \wedge\ b\neq nop)}{\langle S,0,(nop\ hor\ q)\rangle \xrightarrow[n]{} \langle S,1,nop\rangle} \qquad \textbf{(hor-nop1)}$$

$$\frac{p\neq(a\ hor\ b)\ \vee\ (a\neq nop\ \wedge\ b\neq nop)}{\langle S,0,(p\ hor\ nop)\rangle \xrightarrow[n]{} \langle S,1,nop\rangle} \qquad \textbf{(hor-nop2)}$$

Figure 11: The semantics of Céu.

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 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* 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, terminating the whole composition to escape the innermost loop (strongly aborting the other side).

The difference betwen 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 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 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 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 $s = 0$, when the program is guaranteed to be blocked. For the second condition, we check if a nested *hor* is also pending, forcing it to transit before (via rules **par-adv1** or **par-adv2**).

5. RELATED WORK

With respect to control-based languages for embedded systems, the emerging area of Wireless Sensor Networks produced a number of synchronous alternatives to low-level event-driven systems [8, 11, 12, 14]. Some offer predictable and lightweight multi-threading [8, 14] with shared-memory concurrency, but lack thread composition and abortion (as described in Section 2.2). They also do not provide first-class events or a powerful broadcast mechanism, limiting their control mechanisms. OSM [12] provides parallel state machines with a formal and mature synchronous model, with (equivalent) support to 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 [12] (in comparison to Céu's `await` statement). Another Esterel-based language for constrained systems [11] made similar design decisions to ours, handling a single external event at a time, and relying on a deterministic scheduler for safe memory sharing.

Functional Reactive Programming (FRP) adapts modern functional languages to the reactive dataflow style [19]. In particular, Flask [13] shows that dataflow languages can also target constrained systems. Céu borrows some ideas from an FRP implementation [7], such as a push-driven evaluation and glitch prevention. However, dataflow in Céu is less abstract in comparison to FRP, and limited to static relationships only.

As far as we know, Céu is the first language to reconcile the control and dataflow reactive styles.

6. CONCLUSION

As a descendant of Esterel, Céu achieves a high degree of reliability for embedded systems, while also embracing practical aspects, such as shared-memory concurrency, and advanced control-flow mechanisms. Céu introduces a stack-based execution policy for internal events, expanding its expressiveness, such as for describing exceptions and dataflow programming without dedicated primitives. Nonetheless, the language is still simple and all control details (e.g. deterministic execution, stack-based internal events, trail abortion) can be described with a compact formal semantics.

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

[10] N. Halbwachs et al. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, September 1991.

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

[12] 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.

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

[14] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of SenSys'06*, pages 167–180, New York, NY, USA, 2006. ACM.

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

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

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

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

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