A Deterministic and Terminating Semantics for the Synchronous Programming Language Céu

Anonymous Author(s)

Abstract

Céu is a synchronous programming language for embedded soft real-time systems. It focus on control-flow safety features, such as safe shared-memory concurrency and safe abortion of lines of execution, while enforcing internal determinism and time-bounded reactions to the environment. In this work, we present a small-step structural operational semantics for Céu and a proof that reactions are deterministic and always terminate: For a given timeline of input events, multiple executions of the same program always react in bounded time and arrive at the same final memory state.

CCS Concepts • **Theory of computation → Operational semantics**; • **Software and its engineering → Concurrent programming languages**; • **Computer systems organization** → *Embedded software*;

Keywords Operational semantics, Céu, Synchronous languages

ACM Reference Format:
Anonymous Author(s). 1997. A Deterministic and Terminating Semantics for the Synchronous Programming Language Céu. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.475/123_4

1 Introduction

- inspired by Esterel - detailed comparison cite tec

Céu [18] is a programming language for embedded soft real-time systems that aims to offer a concurrent and expressive alternative to C and C++ with the characteristics that follow:

Reactive: code only executes in reactions to events.

Structured: programs use structured control mechanisms, such as **`await`** (to suspend a line of execution), and **`par`** (to combine multiple lines of execution).

Synchronous: reactions run atomically and to completion on each line of execution, i.e., there's no implicit preemption or real parallelism.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WOODSTOCK'97, July 1997, El Paso, Texas USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Structured reactive programming eliminates the *callback hell* [?], letting programmers write code in direct/sequential style [?]. Previous work evaluates the expressiveness of Céu in comparison to event-driven code in C and attests a reduction in source code size (around 25%) with a small increase in memory usage (around 5–10% for *text* and *data*) [18]. Céu has been used in the context of wireless sensor networks [8, 18], multimedia systems [?], and games [1]. It has also been used as an alternative language in an undergradutate-level course on embedded systems for the past 5 years.

Francisco: Embedded systems requirements, determinism, termination

- Céu
  - variaveis compartilhadas
  - concorrencia segura
  - terminacao e determinismo
- Modelo Sincrono
  - conhecido como ammenable to verification
  - valido para Céu?

2 Céu

Céu is a synchronous reactive language inspired by Esterel [7] in which programs advance in a sequence of discrete reactions to external events. Céu is designed for control-intensive applications, supporting concurrent lines of execution, known as *trails*, and broadcast communication through events. Computations within a reaction (such as expressions, assignments, and C calls) execute in no time in accordance to the synchronous hypothesis [10]. The **`await`** is the only Céu statement that actually "consumes" time. An **`await`** statement blocks the current trail allowing the program to advance its other trails; when all trails are blocked, the reaction terminates and control returns to the environment.

In Céu, every execution path within a loop must contain at least one **`await`** statement [4, 18]. This restriction, which is checked statically by the Céu compiler, ensures that every reaction runs in bounded time, eventually terminating with all program trails blocked in **`await`** statements. Céu has a further restriction which it shares with Esterel and synchronous languages in general [2]: computations that actually take a non-negligible time to run (e.g., cryptography or image processing algorithms) violate the zero-delay hypothesis, and thus cannot be directly implemented.

[1]

Francisco: TODO

Listing 1 below shows a compact reference of Céu.

```
// Declarations:
input  ⟨type⟩ ⟨ids⟩;          // declare external input events
output ⟨type⟩ ⟨ids⟩;          // declare external output events
event  ⟨type⟩ ⟨ids⟩;          // declare internal events
var    ⟨type⟩ ⟨id⟩ = ⟨exp⟩;// declare and initialize variable

// Event handling:
⟨id⟩ = await ⟨id⟩;            // await event and assign the received value
⟨id⟩ = await ⟨time⟩;          // await time and assign the delayed delta
emit ⟨id⟩(⟨exp⟩);            // emit event passing a value

// Control-flow:
⟨stmt⟩ ; ⟨stmt⟩                          // sequence
if ⟨exp⟩ then ⟨stmts⟩ else ⟨stmts⟩ end// conditional
loop do ⟨stmts⟩ end                      // repetition
finalize ⟨stmts⟩ with ⟨stmts⟩ end  // finalization

par/or  do ⟨stmts⟩ with ⟨stmts⟩ end// aborts when any side terminates
par/and do ⟨stmts⟩ with ⟨stmts⟩ end// terminates when all sides terminate
par     do ⟨stmts⟩ with ⟨stmts⟩ end// never terminates

// Assignment & integration with C:
⟨id⟩ = ⟨exp⟩;                            // assign value to variable
_⟨id⟩(⟨exps⟩)                            // call C function (id starts with '_')
```

Listing 1. The concrete syntax of Céu.

To make matters concrete, consider the program of Listing 2. This program continuously turns a LED on for 2 seconds and off for 1 second, and terminates after 1 minute of activity with the LED off. The implementation uses a **par/or** to run two activities in parallel: an endless loop that blinks the LED on and off (lines 2–7), and a single statement that waits for 1 minute before terminating (line 9). The **par/or** block stands for a *parallel-or* composition; when executed it creates n parallel trails (in this case, $n = 2$) and rejoins them when any of these n trails terminates, automatically aborting the other trails.

```
1  par/or do
2      loop do
3          _led(1);
4          await 2s;
5          _led(0);
6          await 1s;
7      end
8  with
9      await 1min;
10 end
11 _led(0);
```

Listing 2. Céu program that blinks a LED during 1 minute.

In Céu, any identifier prefixed with an underscore (e.g., `_led`) is passed unchanged to the underlying C compiler. Therefore, access to C is straightforward and syntactically traceable. To ensure that programs operate synchronously, the compiler environment should only provide access to C operations that are assumed run in zero time, such as non-blocking I/O and access to **struct**'s.

2.1 External and Internal Events

Céu defines time as a discrete sequence of reactions to unique external input events. These external input events are received from the environment; each of them delimits a new logical unit of time and triggers a corresponding reaction. The life-cycle of a program in Céu can be summarized as follows [18].

1. The program initiates the "boot reaction" in a single trail (parallel constructs may create new trails).
2. Active trails execute until they await or terminate, one after another. This step is called a *reaction chain*, and always runs in bounded time.
3. When all its trails are blocked, the program goes idle and the environment takes control.
4. On the occurrence of a new external input event, the environment awakes *all* trails awaiting that event, and the program goes back to step (ii).

A program must react to an event completely before handling the next one. By the synchronous hypothesis, the time the program spends in step (ii) above is conceptually zero (in practice, negligible). Thus, from the point of view of the environment, the program is always idle—on step (iii). In practice, if a new external input event occurs while a reaction is being computed, the event is saved on a queue, which effectively schedules it to be processed by a subsequent reaction.

External events and discrete time

The processing of external input events induces a discrete notion of time in Céu. Figure 1 illustrates this notion. The continuous timeline shows an absolute reference clock with "physical timestamps" for the event occurrences (e.g., event `C` occurs at $17ms521us$). The discrete timeline shows how the same occurring events fit in the logical notion of time of Céu. The boot reaction `boot-0` happens before any input on program startup. Event `A` "physically" occurs during `boot-0` but, because time is discrete, its corresponding reaction can only execute afterwards, at logical instant `A-1`. Similarly, event `B` occurs during `A-1` and its reaction is postponed to execute at `B-2`. Event `C` also occurs during `A-1` but its reaction must also wait for `B-2` to execute and so it is postponed to execute at `C-3`. Event `D` occurs during an idle period and can start

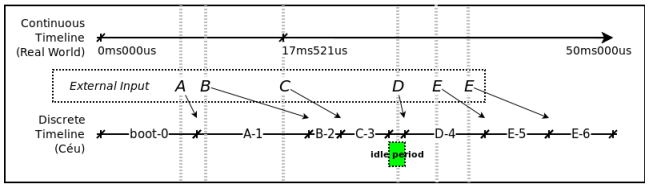

Figure 1. The discrete notion of time in Céu.

immediately at D-4. Finally, two instances of event E occur during D-4; they are handled in the subsequent reactions E-5 and E-6.

Unique input events imply mutually exclusive reactions, which execute atomically and never overlap. Automatic mutual exclusion is a prerequisite for deterministic reactions as we discuss in Section 3.

In practice, the synchronous hypothesis for Céu holds if reaction rate is faster than the rate of incoming input events. Otherwise, the program continuously accumulates a delay between the real occurrence time and actual reaction time of events. In the soft real-time systems targeted by Céu (e.g., sensor networks, multimedia systems, interactive games, etc.) such delay and postponed reactions might be tolerated by users as long as they are infrequent and the application does not take too long to catch up with real time.

Internal events as subroutines

In Céu, the queue-based processing of events described previously applies only to external input events, i.e., those events submitted to the program by the environment. Internal events, which are those events generated internally by the program via **emit** statements, are processed in a stack-based manner. These internal events provide a fine-grained execution control and, because of their stack-based processing, can be used to implement a limited form of subroutine, as illustrated in Listing 3 below.

```
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      ...
10     emit inc(&v);     // call 'inc'
11     _assert(v==2);    // after return
12 end
```

Listing 3. A Céu program with a "subroutine".

In Listing 3, the "subroutine" inc is defined as a loop (lines 3–6) that continuously awaits its identifying event (line 4), and increments the value passed to it by reference (lines 5). A trail in parallel (lines 8–11) invokes the subroutine through the **emit** statement at line 10. Given the stack-based execution mode of internal events, after the emit statement at line 10 is executed, 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 does the calling trail resumes and moves on to execute the assertion on line 11.

Céu also supports nested **emit** invocations for internal events. For instance, the body of the subroutine inc in Listing 3 could **emit** another internal event after awaking (line 4), creating a new level in the stack. We can think of the stack as a record of the nested, fine-grained internal reactions that happen inside the same bigger reaction to some external event.

This form of subroutine has a significant limitation though: it cannot express recursion (an **emit** to itself is always ignored as a running trail cannot be waiting on itself). That said, it is this very limitation that brings important safety properties to subroutines. First, such subroutines are guaranteed to react in bounded time. Second, memory for locals is also bounded, not requiring data stacks. Third, Céu subroutines can be safely used by the other primitives of the language, such as parallel compositions and the **await** statement, without breaking the programming model. In particular, after calling a subroutine these primitives wait while keeping context information, such as locals and the program counter, which makes the calls behave similarly to those of coroutines [9]. Finally, in previous work, we built other advanced control mechanisms on top of internal events, such as resumable exceptions and reactive variables [17].

Guilherme: Revisei do início da Seção 2 até aqui.

2.2 Shared-Memory Concurrency

- referenciar warnings

Embedded applications make extensive use of global memory and shared resources, such as through memory-mapped registers and system calls to device drivers. Hence, an important goal of Céu is to ensure a reliable behavior for programs with concurrent lines of execution sharing memory and interacting with the environment.

```
1 input void A, B;
2 var int x = 1;
3 par/and do
4     await A;
5     x = x + 1;
6 with
7     await B;
8     x = x * 2;
9 end
```

[a] Accesses to x are never concurrent.

```
1 input void A;
2 var int y = 1;
3 par/and do
4     await A;
5     y = y + 1;
6 with
7     await A;
8     y = y * 2;
9 end
```

[b] Accesses to y are concurrent but deterministic.

Figure 2. Shared-memory concurrency in Céu: example [a] is safe because the trails access x atomically in different reactions; example [b] is unsafe because both trails access y in the same reaction.

In Céu, when multiple trails are active during the same reaction, they are scheduled in lexical order, i.e., in the order they appear in the program source code. For instance, consider the two examples in Figure 2, both defining a shared variable (ln. 2), and assigning to it in parallel trails (ln. 5, 8).

In the example [a], the two assignments to x can only execute in reactions to different events A and B, which cannot occur simultaneously by definition (Section 2.1). Hence, for the sequence of events A->B, x becomes 4 ((1+1)*2), while for B->A, x becomes 3 ((1*2)+1).

In the example [b], the two assignments to y are simultaneous because they execute in reaction to the same event A. Since Céu employs lexical order for intra-reaction statements, the execution is still deterministic, and y always becomes 4 ((1+1)*2). However, that an (apparently innocuous) change in the order of trails modifies the behavior of the program. To mitigate this threat, Céu performs concurrency checks at compile time to detect conflicting accesses to shared variables: if a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable [18]. Nonetheless, the static checks are optional and do not affect the semantics of the language.

2.3 Abortion and Finalization

The **par/or** of Céu is an *orthogonal abortion mechanism* [2] because the two sides in the composition need not be tweaked with synchronization primitives or state variables in order to affect each other. In addition, abortion is *immediate* in the sense that it executes atomically in the current micro reaction. Immediate orthogonal abortion is a distinctive feature of synchronous languages and cannot be expressed effectively in traditional (asynchronous) multi-threaded languages [2, 16].

However, aborting lines of execution that deal with external resources may lead to inconsistencies. For this reason, Céu provides a **finalize** construct to unconditionally execute a series of statements even if the enclosing block is aborted.

Céu also enforces the use of **finalize** for system calls that deal with pointers representing resources, as illustrated in the two examples of Figure 3:

- If Céu **passes** a pointer to a system call (ln. [a]:5), the pointer represents a **local** resource (ln. [a]:2) that requires finalization (ln. [a]:7).
- If Céu **receives** a pointer from a system call return (ln. [b]:4), the pointer represents an **external** resource (ln. [b]:2) that requires finalization (ln. [b]:6).

Céu tracks the interaction of system calls with pointers and requires finalization clauses to accompany them. In the example in Figure 3.a, the local variable msg (ln. 2) is an internal resource passed as a pointer to _send_request (ln. 5), which is an asynchronous call that transmits the buffer in the background. If the block aborts (ln. 11) before receiving an acknowledge from the environment (ln. 9), the local msg goes out of scope and the external transmission now holds a *dangling pointer*. The finalization ensures that the transmission also aborts (ln. 7). In the example in Figure 3.b, the call to _fopen (ln. 4) returns an external file resource as a pointer. If the block aborts (ln. 12) during the **await** A (ln. 9), the file remains open as a *memory leak*. The finalization ensures that the file closes properly (ln. 6). In both cases, the code does not compile without the **finalize** construct.[2]

```
1  par/or do
2     var _buffer_t msg;
3     <...> // prepare msg
4     finalize
5        _send_request(&msg);
6     with
7        _send_cancel(&msg);
8     end
9     await SEND_ACK;
10 with
11    <...>
12 end
13 .
```

[a] Local resource finalization

```
1  par/or do
2     var _FILE* f;
3     finalize
4        f = _fopen(...);
5     with
6        _fclose(f);
7     end
8     _fwrite(..., f);
9     await A;
10    _fwrite(..., f);
11 with
12    <...>
13 end
```

[b] External resource finalization

Figure 3. Céu enforces the use of finalization to prevent *dangling pointers* for local resources and *memory leaks* for external resources.

The finalization mechanism of Céu is fundamental to preserve the orthogonality of the **par/or** construct since the clean up code is encapsulated in the aborted trail itself.

3 Formal Semantics

In this section, we introduce a reduced syntax of Céu and propose an operational semantics to formally describe the language. We describe a small synchronous kernel highlighting the peculiarities of Céu, in particular the stack-based execution for internal events. For the sake of simplicity, we focus on the control aspects of the language, leaving out side effects and system calls (which behave like in conventional imperative languages).

3.1 Abstract Syntax

Figure 4 shows the syntax for a subset of Céu that is sufficient to describe all semantic peculiarities of the language. Except for fin and the expressions used internally by the semantics (i.e., *awating*, *emitting*, p @ *loop*; p, and *nop*), all other expressions are equivalent to their counterparts in the concrete language.

The *mem*(*id*) primitive represents all accesses, assignments, system calls, and output events that affect a memory location identified by *id*. According to the synchronous hypothesis of Céu, *mem* expressions are considered to be atomic and instantaneous. 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

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

```
                                  // primary expressions
  p ::= mem(id)                   (any memory access to `id')
      | await(id)                 (await event `id')
      | emit(id)                  (emit event `id')
      | break                     (loop escape)
                                  // compound expressions
      | if mem(id) then p else p  (conditional)
      | p ; p                     (sequence)
      | loop p                    (repetition)
      | p and p                   (par/and)
      | p or p                    (par/or)
      | fin p                     (finalization)
                                  // derived by semantic rules
      | awaiting(id,n)            (awaiting `id' since sequence number `n')
      | emitting(n)               (emitting on stack level `n')
      | p @ loop p                (unwinded loop)
      | nop                       (terminated expression)
```

Figure 4. Reduced syntax of Céu.

in programs. Note that *mem* and *await*/*emit* expressions do not share identifiers, i.e., an identifier is either a variable or an event.

3.2 Operational Semantics

The core of our semantics describes how a program reacts to a single external input event, i.e., starting from the input event, how the program behaves and becomes idle again to proceed to the subsequent reaction. We use a set of small-step operational rules, which are built in such a way that at most one transition is possible at any time, resulting in deterministic reactions. Each reaction is identified by a ever-increasing n that remains constant during the entire reaction. The transition rules map a program p and a stack of events S in a single step to a modified program and stack:

$$\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle \qquad \textbf{(rule-inner)}$$

where

$S, S' \in id^*$ (*stack of event identifiers* : $[id_{top}, ..., id_{bottom}]$)
$p, p' \in P$ (*program as described in Figure* 4)
$n \in \mathbb{N}$ (*unique identifier for the entire reaction*)

At the beginning of a reaction, the stack is initialized with the occurring external event *ext* ($S = [ext]$), but *emit* expressions can push new events on top of it (we discuss how they are popped further). The sequence number n, which is incremented each reaction, prevents that awaiting expressions awake in the same reaction they are reached (the *delayed awaits* as explained in Section ??).

The transition rules for the primary expressions are as follows:

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

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

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

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

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

A *mem* operation executes immediately and becomes a *nop* to indicate termination (rule **mem**). An *await* is transformed into an *awaiting* (rule **await**) as an artifice to remember the external sequence number $n + 1$ it can awake: an *awaiting* can only transit to a *nop* (rule **awake**) if its referred event *id* matches the top of the stack and it was reached in a previous reaction (i.e., sequence number $m \leq n$). An *emit* transits to an *emitting* holding the current stack level ($|S|$ stands for the stack length), and pushing the referred event on the stack (rule **emit**). With the new stack level $|S|+1$, the *emitting*($|S|$) itself cannot transit, as rule **pop** expects its parameter k to match the current stack level. This trick provides the desired stack-based semantics for internal events.

Proceeding to compound expressions, the rules for conditionals and sequences are straightforward:

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

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

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

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

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

Given that our semantics focuses on control, rules **if-true** and **if-false** are the only to query *mem* expressions. The "magical" function *val* receives a memory identifier and the current reaction sequence number, returning the current memory value. Although the value here is arbitrary, it is

unique in a reaction, because a given expression can execute only once within it (remember that *loops* must contain *awaits* which, from rule **await**, cannot awake in the same reaction they are reached).

The rules for loops are analogous to sequences, but use `@' as separators to properly bind breaks to their enclosing loops:

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

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

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

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

When a program encounters a *loop*, it first expands its body in sequence with itself (rule **loop-expd**). Rules **loop-adv** and **loop-nop** are similar to rules **seq-adv** and **seq-nop**, advancing the loop until they reach a *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*.

Proceeding to parallel compositions, the semantic rules for *and* and *or* always force transitions on their left branches p to occur before their right branches q:

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

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

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

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

The deterministic behavior of the semantics relies on the *isBlocked* predicate, which is defined in Figure 5 and used in rules **and-adv2** and **or-adv2**. These rules require the left branch p to be blocked in order to allow the right transition from q to q'. Basically, the *isBlocked* predicate determines that an expression becomes blocked when all of its trails

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

Figure 5. The recursive predicate *isBlocked* is true only if all branches in parallel are hanged in *awaiting* or *emitting* expressions that cannot transit.

in parallel hang in *awaiting* and *emitting* expressions that cannot advance.

For a parallel *and*, if one of the sides terminates, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**, as follows). For a parallel *or*, if one of the sides terminates, the whole composition terminates, also applying the *clear* function to properly finalize the aborted side (rules **or-nop1** and **or-nop2**):

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

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

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

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

The *clear* function, defined in Figure 6, concatenates all active fin bodies of the side being aborted, so that they execute before the composition rejoins. Note that there are no transition rules for fin expressions. This is because once reached, a fin expression halts and will only execute when it is aborted by a trail in parallel and is expanded by the *clear* function. In Section 3.3.3, we show how to map a finalization block in the concrete language to a fin in the formal semantics. Note also that there is a syntactic restriction that a fin body can only contain *mem* expressions in sequence, i.e., they are guaranteed to execute entirely within a reaction.

Finally, a *break* in one of the sides in parallel escapes the closest enclosing *loop*, properly aborting the other side by

applying the *clear* function:

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

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

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

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

A reaction 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. However, *emitting* expressions are pending in lower stack indexes and should eventually resume in the ongoing reaction (see rule **pop**). Therefore, we define another rule that behaves as **rule-inner** (presented above) if the program can advance, and, otherwise, pops the stack to resume the lower level:

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

To describe a *reaction* in Céu, i.e., how a program behaves in reaction to a single external event, we use the reflexive transitive closure of **rule-outer**:

$$\langle S, p\rangle \overset{*}{\underset{n}{\Longrightarrow}} \langle S', p'\rangle$$

Finally, to describe the complete execution of a program, we trigger multiple "invocations" of reactions in sequence:

$$\langle [e1], p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle$$
$$\langle [e2], p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle$$
$$\langle [e3], p''\rangle \overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle$$
$$\ldots$$

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

Figure 6. The function *clear* extracts *fin* expressions in parallel and put their bodies in sequence.

```
par/or do
  <...>
  emit e => 1;
with
  v = await e;
  _printf("%d\n",v);
end
.
```

(a)

```
par/or do
  <...>
  e_ = 1;
  emit e;
with
  await e;
  v = e_;
  _printf("%d\n",v);
end
```

(b)

```
<...> ; mem ; emit(e)
        or
await(e) ; mem ; mem
.
```

(c)

Figure 7. Two-step translation from concrete to abstract **emit** and **await** expressions. The concrete code in (a) communicates the value 1 from the **emit** to the **await**. The abstract code in (c) uses a shared variable to hold the value.

Each invocation starts with the occurring external event at the top of the stack and finishes with a modified program and an empty stack. After each invocation, we increment the sequence number.

3.3 Concrete Language Mapping

Most statements from Céu ("concrete Céu") map directly to those presented in the reduced syntax in Figure 4 ("abstract Céu"). For instance, the **if** in the concrete language behaves exactly like the *if* in the formal semantics. However, there are some significant mismatches between the concrete and abstract Céu, and we (informally) present appropriate mappings in this section. Again, we are not considering side-effects, which are all mapped to the *mem* semantic construct.

3.3.1 await and emit

The concrete **await** and **emit** primitives support communication of values between them. In the two-step translation in Figure 7, we start with the concrete program in Céu (a), which communicates the value 1 between the **emit** and **await** in parallel. In the intermediate translation (b), we include the shared variable `e\_` to hold the value being communicated between the two trails in order to simplify the **emit**. Finally, we convert the program into the equivalent in the abstract syntax (c), translating side-effect statements into *mem* expressions. External events have a similar translation, i.e., each external event requires a corresponding variable that is explicitly set by the environment before each reaction.

3.3.2 First-class Timers

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

```
dt = await 10ms;
.
```
(a)

```
var int tot = 10000;
loop do
    await DT;
    tot = tot - DT_;
    if tot <= 0 then
        dt = -tot;
        break;
    end
end
.
```
(b)

```
mem; // 10ms
loop(
    await(DT);
    mem;
    if mem then
        mem;
        break
    else
        nop
)
```
(c)

Figure 8. Two-step translation from concrete to abstract timer.

$$\langle [DT], p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle$$
$$\langle [e1], p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle$$
$$\langle [DT], p''\rangle \overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle$$
$$\langle [e2], p'''\rangle \overset{*}{\underset{4}{\Longrightarrow}} \langle [], p''''\rangle$$
$$\ldots$$

The event DT has an associated variable DT_ carrying the wall-clock time elapsed between two occurrences in sequence, as depicted by the two-step translation in Figure 8. In the concrete program (a), the variable dt holds the residual delta time (as described in Section ??) after awaking from the timer. In the first step of the translation (b), we expand the **await** 10ms to a **loop** that decrements the elapsed number of microseconds for each occurrence of DT. When the variable tot reaches zero, we escape the **loop** setting the variable dt to contain the appropriate delta. In the last step (c), we convert the program to the abstract syntax.

3.3.3 Finalization Blocks

The biggest mismatch between concrete and abstract Céu is regarding the **finalize** blocks, which require more complex modifications in the program for a proper mapping using *fin* expressions. In the three-step translation in Figure 9, we start with a concrete program (a) that uses a **finalize** to safely _release the reference to ptr kept after the call to _hold. In the translation, we first need to catch the outermost **do-end** termination to run the finalization code. For this, we translate the block into a **par/or** (b) with the original body in parallel with a *fin* expression to run the finalization code. Note that the *fin* has no transition rules in the semantics, keeping the **par/or** alive. This way, the *fin* body only executes when the **par/or** terminates either normally (after the **await** B), or aborted from an outer composition. However, the *fin* still (incorrectly) executes even if the call

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

```
par/or do
  var int* ptr =
  await A;
  _hold(ptr);
  with
  fin
    _release(ptr);
  end
.
```
(b)

```
f_ = 0;
par/or do
  var int* ptr =
  await A;
  _hold(ptr);
  f_ = 1;
  await B;
  { fin
    if f_ then
      _release(ptr);
    end }
end
.
```
(c)

```
mem;
(
  mem;.>;
  await(A);
  mem;
  mem;
  await(B);
  or
  fin
  if mem then
    mem;
  else
    nop
)
```
(d)

Figure 9. Three-step translation from concrete to abstract finalization.

to _hold is not reached in the body due to an abort before awaking from the **await** A. To deal with this issue, for each *fin* we need a corresponding flag to keep track of code that needs to be finalized (c). The flag is initially set to false, avoiding the finalization code to execute. Only after the call to _hold that we set the flag to true and enable the *fin* body to execute. The complete translation substitutes the side-effect operations with *mem* expressions (d).

4 Related Work

Céu was strongly influenced by Esterel but they differ in the most fundamental aspect of the notion of time [18]. Esterel defines time as a discrete sequence of logical unit instants or "ticks". At each tick, the program reacts to an arbitrary number of simultaneous input events from the environment. The presence of multiple inputs requires careful static analysis to detect and reject programs with *causality cycles* and *schizophrenia problems* [3, 6, 12, 19–22, 26]. In contrast, Céu defines time as a discrete sequence of reactions to unique input events. In the formal semantics, ...

Francisco: como isso aparece na semantica

... Céu also rejects some syntactically correct programs to avoid infinite execution, but with simple restrictions in the abstract syntax tree.

Another distinction is that, in Esterel, the behavior of internal and external events is equivalent, while in Céu internal events introduce stack-based micro reactions within external reactions, providing more fine-grained control for intra-reaction execution. In the formal semantics, ...

Francisco: como isso aparece na semantica

... Some variants of the Statecharts synchronous visual language also distinguish internal from external events [24]. In Statemate [14], *"reactions to external and internal events*

(...) can be sensed only after completion of the step", implying queue-based execution. In Stateflow [13], *"the receiving state (of the event) acts here as a function"*, which is similar to CÉU's stack-based execution. We are not aware of formalizations for these ideas for a deeper comparison with CÉU.

Like other synchronous languages (*Reactive C* [5], *Protothreads* [11], *SOL* [15], *SC* [25], and *PRET-C* [1]), CÉU relies on deterministic scheduling to preserve intra-reaction determinism. In addition, it also performs concurrency checks to detect trails that, when reordered, change the observable behavior of the program, i.e., trails that actually rely on deterministic scheduling [18]. Esterel is only deterministic with respect to external behavior: *"the same sequence of inputs always produces the same sequence of outputs"* [4]. However, the execution order for operations 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"* [4]. For this reason, Esterel, does not support shared-memory concurrency: *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [4].

Esterel describes a finalization mechanism in a standardization proposal [23] that is similar to CÉU's. However, we are not aware of an open implementation or a formal semantics for a deeper comparison.

Francisco: outras linguagens sincronas

Francisco: outras linguagens deterministicas

Francisco: outras linguagens com terminacao

5 Conclusion

Francisco: TODO

References

[1] Sidharta Andalam, Partha Roop, and Alain Girault. 2010. Predictable multithreading of embedded applications using PRET-C. In *Proceeding of MEMOCODE'10*. IEEE, 159–168.

[2] Gérard Berry. 1993. Preemption in Concurrent Systems.. In *FSTTCS (LNCS)*, Vol. 761. Springer, 72–93.

[3] Gérard Berry. 1999. *The Constructive Semantics of Pure Esterel (draft version 3)*. Ecole des Mines de Paris and INRIA.

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

[5] Frédéric Boussinot. 1991. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience* 21, 4 (1991), 401–428.

[6] Frédéric Boussinot. 1998. SugarCubes implementation of causality. (1998).

[7] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.

[8] Adriano Branco, Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, and Silvana Rossetto. 2015. Terra: Flexibility and Safety in Wireless Sensor Networks. *ACM Trans. Sen. Netw.* 11, 4, Article 59 (Sept. 2015), 27 pages. https://doi.org/10.1145/2811267

[9] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM TOPLAS* 31, 2 (Feb. 2009), 6:1–6:31.

[10] Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. 2005. The Synchronous Hypothesis and Synchronous Languages. In *Embedded Systems Handbook*, R. Zurawski (Ed.).

[11] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*. ACM, 29–42.

[12] Stephen A. Edwards. 2005. Using and Compiling Esterel. MEMOCODE'05 Tutorial. (July 2005).

[13] Grégoire Hamon and John Rushby. 2007. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 447–456.

[14] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5, 4 (1996), 293–333.

[15] Marcin Karpinski and Vinny Cahill. 2007. High-Level Application Development is Realistic for Wireless Sensor Networks. In *Proceedings of SECON'07*. 610–619.

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

[17] Francisco Sant'Anna, Noemi Rodriguez, and Roberto Ierusalimschy. 2013. Advanced Control Reactivity for Embedded Systems. Workshop on Reactivity, Events and Modularity (REM'13). (2013).

[18] Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

[19] Klaus Schneider and Michael Wenz. 2001. A new method for compiling schizophrenic synchronous programs. In *Proceedings of CASES'01*. ACM, 49–58.

[20] Ellen M Sentovich. 1997. Quick conservative causality analysis. In *System Synthesis, 1997. Proceedings., Tenth International Symposium on*. IEEE, 2–8.

[21] Thomas R Shiple, Gerard Berry, and Hemé Touati. 1996. Constructive analysis of cyclic circuits. In *European Design and Test Conference, 1996. ED&TC 96. Proceedings*. IEEE, 328–333.

[22] Olivier Tardieu and Robert De Simone. 2004. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of MEMOCODE'04*. IEEE, 39–48.

[23] Esterel Technologies. 2005. *The Esterel v7 Reference Manual*. Initial IEEE standardization proposal.

[24] Michael von der Beeck. 1994. A comparison of statecharts variants. In *Proceedings of FTRTFT'94*. Springer, 128–148.

[25] Reinhard von Hanxleden. 2009. SyncCharts in C: a proposal for light-weight, deterministic concurrency. In *Proceedings EMSOFT'09*. ACM, 225–234.

[26] Jeong-Han Yun, Chul-Joo Kim, Seonggun Kim, Kwang-Moo Choe, and Taisook Han. 2013. Detection of harmful schizophrenic statements in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 3 (2013), 80.