Advanced Control Reactivity for 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 con-
strained embedded platforms. Relying on a deterministic se-
manitics, it provides safe shared-memory concurrency among
lines of execution. Céu introduces a stack-based execution
policy for internal events which enables advanced control
mechanisms considering the context of embedded systems,
such as exception handling and a limited form of coroutines.
The conjunction of shared-memory concurrency with inter-
nal events allows programs to express dependency among
variables reliably, reconciling the control and dataflow reac-
tive styles in a single language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and
Theory; D.3.3 [Programming Languages]: Language Con-
structs and Features

General Terms
Design, Languages

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

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

We believe that embedded-system programming can bene-
fit from a new language that reconciles both reactive syn-
chronous styles, while preserving typical C features that pro-
grammers are familiarized with, such as shared memory con-
currency. Céu [15] is a language targeting embedded sys-
tems based on Esterel with some differences that enable new
control functionalities, which are the focus of this work:

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

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

Céu shares limitations with Esterel and synchronous lan-
guages in general: computations that run in unbounded
time (e.g., cryptography, image processing) do not fit the
zero-delay hypothesis [14], and cannot be elegantly imple-
mented. Nonetheless, previous work focusing on Wireless
Sensor Networks [15] shows that the expressiveness of Céu
is sufficient for embedded applications, with a reduction in
source code size around 30% in comparison to event-driven
code in C. Céu has a small memory footprint, using less
than 5 Kbytes of ROM and 100 bytes of RAM for a pro-
gram with sixteen (simple) flows of execution.

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

2. OVERVIEW OF CÉU
Céu is a synchronous reactive language based on Esterel [6]
with support for multiple concurrent lines of execution known
as trails. By reactive, we mean that programs are stimulated
by the environment through input events that are broadcast
to all awaiting trails. By synchronous, we mean that all
trails at any given time are either reacting to the current
event or are awaiting another event; in other words, trails
are never reacting to different events.

Figure 1 shows the implementations in Esterel and Céu side-by-side for the following control specification [5]: "Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs". The first phrase of the specification is translated almost identically in the two languages (lines 4-9): await and terminate only when both events occur (the ‘||’ and par/and constructs are equivalent). For the second phrase, the reset behavior, the Esterel version uses a await-when, which serves the same purpose of Céu’s par/or: the occurrence of event R aborts the awaiting statements in parallel and restarts the loop.

Céu (like Esterel) has a strong imperative flavor, with explicit control flow through sequences, loops, and also assignments. Being designed for control-intensive applications, it provides support for concurrent lines of execution and broadcast communication through events. Programs advance in sequence of discrete reactions to external events. Internal computations within a reaction (e.g., expressions, assignments, and native calls) are considered to take no time in accordance with the synchronous hypothesis [14]. The await statements are the only ones that halt a running reaction and allow a program to advance in this notion of time. To ensure that reactions run in bounded time and programs always progress, loops are statically required to contain at least one await statement in all possible paths [15, 5].

In the sections that follow, we show the three basic differences between Céu and Esterel: deterministic execution for operations with side-effects (Section 2.1), hierarchical abortion for lines of execution (Section 2.2), and stack-based execution for internal events (Section 2.3). By providing a precise control for concurrent lines of execution, these differences are fundamental to enable advanced mechanisms in Céu (presented in Section 3).

2.1 External reactions and determinism

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

Another difference between Esterel and Céu regards their definitions for determinism: Esterel is deterministic with respect to reactive control: “the same sequence of inputs always produces the same sequence of outputs” [5]. However, the execution order for operations with side-effects within a reaction is non-deterministic: “if there is no control dependency, as in “call f1() || call f2()”, the order is unspecified and it would be an error to rely on it” [5]. In Céu, when multiple trails are active at a time, as in “par/and do _f1() with _f2() end”, they are scheduled in the order they appear in the program source code (i.e., _f1 executes first). This way, Céu is deterministic also with respect to the order of execution of side effects within a reaction.

On the one hand, enforcing an execution order for concurrent operations may seen arbitrary and also precludes true parallelism. On the other hand, it provides a priority scheme for trails, and makes shared-memory concurrency possible. In contrast, Esterel does not support shared memory: “If a variable is written by some thread, then it can neither be read nor be written by concurrent threads” [5]. For embedded development, we believe that deterministic shared-memory concurrency is beneficial, given the extensive use of memory mapped ports for I/O and lack of support for real parallelism. Other embedded languages made a similar design choice [9, 1].

2.2 Thread abortion

The introductory example of Figure 1 illustrates how synchronous languages can abort awaiting lines of execution (i.e., awaiting A and B) without tweaking them with synchronization primitives. In contrast, traditional (asynchronous) multi-threaded languages cannot express thread termination safely [4, 13].

The code fragments of Figure 2 show corner cases for thread abortion: when the event A occurs, the program behavior seems ambiguous. For instance, it is not clear in code a in Esterel if the call to f should execute or not after A, given that the body and abortion events are the same. For this reason, Esterel provides weak and strong variations for the abort statement. With strong abortion (the default), the body is aborted immediately and the call does not execute. In Céu, given the deterministic scheduling rules, strong and weak abortions can be chosen by reordering trails inside a par/or, e.g., in code b, the second trail is strongly aborted by the first trail and the call to f never executes.

Céu also supports par/or (hierarchical-or) compositions which schedules both sides before terminating. Therefore, in code c, both _g and _f (in this order) execute in reaction to A. Hierarchical traversal is fundamental for dataflow programming, ensuring that all running dependencies execute before they abort each other (to be discussed in Section 3.2).

2.3 Internal events

Esterel makes no semantic distinctions between internal and external signals, both having only the notion of either presence or absence during the entire reaction [4]. In Céu, however, internal events follow a stack-based execution policy, similar to subroutine calls in typical programming languages. Figure 3 illustrates the use of internal signals (events) in Esterel and Céu. For the version in Esterel, given that there is no control dependency between the calls to f, they
On the one hand, this form of subroutines has a significant limitation that it cannot express recursive calls: an `emit` to itself will always be ignored, given that a running body cannot be awaiting itself. On the other hand, this very same limitation brings some important safety properties to subroutines: first, they are guaranteed to react in bounded time; second, memory for locals is also bounded, not requiring runtime stacks. Also, this form of subroutines can use the other primitives of Céu, such as parallel compositions and the `await` statement. In particular, they await keeping context information such as locals and the program counter, just like coroutines [12]. In Section 3.2, we take advantage of the lack of recursion to properly describe mutual dependency among trails in parallel.

### 3. ADVANCED CONTROL MECHANISMS

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

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

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

1. Normal flow invokes the read operation (line 11 of Figure 5) and pauses.
   
   ```
   stack: [norm]
   ```

2. Read operation awakes (line 6 of Figure 6), throws an exception (line 8), and pauses.
   
   ```
   stack: [norm, read]
   ```

3. Exception handler awakes (line 3 of Figure 7) and terminates the par/or, aborting the read call, the normal behavior, and terminating the program.
   
   ```
   stack: []
   ```

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

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...
Figure 5: Normal flow to read file entries.

```plaintext
// DECLARATIONS
input int ENTRY;
var _FILE f = <...>; // file handler
var char[10] buf; // current entry
event int read;
event void except;

// NORMAL FLOW
loop do
  var int n = await ENTRY;
  _read(f,buf,n) != n
  _FILE {read => n; // calls 'read n chars'
   _read(f,buf,n) != n then
     emit except // throws exception
   }
end
```

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

default string (line 5). The program now behaves as follows (steps 1-2 are the same):

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

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

```plaintext
// DECLARATIONS
par/or do
loop do
  await except; // catches exceptions
  buf = <...>; // assigns a default
end
```

Figure 8: Exception handling with resumption.

Céü. 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/or (lines 7-11) ensures that b is only updated (in line 12) after e1 and E (in lines 8 and 10). The program behavior for a reaction to E=>1 (which should awake lines 8 and 17) is the following:

1. Line 8 awakes and assigns v1=1. (The par/or 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/or 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/or rejoins and correctly emits b=1 (i.e., v1=1 < v2=2).

Figure 10 shows a mutual conversion for temperatures in Celsius and Fahrenheit, so that whenever the value in one unit is set, the other is automatically recalculated (a problem proposed in [2]). Mutual dependency is another known issue in datalow languages, usually requiring the placement of a specific delay operator to avoid runtime cycles [7, 16]. However, an explicit delay is somewhat ad hoc because it splits an internal dependency problem across two reactions to the environment. Céü relies on the stack-based execution for internal events to avoid runtime cycles. The code in the right of the Figure 10 implements the conversion formula in Céü. We first define the tc and tf events to signal temperature changes (line 1). Then, we create the 1st and 2nd trails to await for changes and mutually update the temperatures (lines 3-6 and 8-11). The third trail (lines 13-14) signals a temperature change and the program behaves as follows:

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

3.2 Dataflow programming

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

A glitch is a situation in which a dependency graph is updated in an inconsistent order. It is usually avoided by traversing the graph in topological order [7, 2]. In a glitch-free implementation, when E changes, e1 should be updated before b (because b also depends on e1) to avoid yielding false. The code in the right of the graph implements it in
The semantics specifies a deterministic order for memory accessing on the particular control aspects of the language. In this section, we present a formal semantics of Céu.

4. THE SEMANTICS OF CÉU

4.1 Model of Execution

The exposed dataflow techniques are rather verbose and low on the complexity of the two examples in Figures 9 and 10. The exposed dataflow techniques are rather verbose and low on the complexity of the two examples in Figures 9 and 10.}

4.2 Formal Semantics

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

The exposed dataflow techniques are rather verbose and low level. Again, some syntactic sugar could reduce considerably the complexity of the two examples in Figures 9 and 10.

4.3 Formal Semantics

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

The exposed dataflow techniques are rather verbose and low level. Again, some syntactic sugar could reduce considerably the complexity of the two examples in Figures 9 and 10.

4.4 Formal Semantics

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

The exposed dataflow techniques are rather verbose and low level. Again, some syntactic sugar could reduce considerably the complexity of the two examples in Figures 9 and 10.
its referred event $id$ matches the top of the stack and its sequence number is smaller than the current one ($m < n$). An $emit$ transits to an $emitting$ holding the current stack level ($S$ stands for the stack size), and pushes the referred event on the stack ($rule \ emit$). With the new stack level $[s : S]$ after an emit, the resulting $emitting([S])$ cannot transit yet, as it requires $emitting$ expects its parameter $[S]$ to match the current stack level. This trick provides the desired stack-based semantics for internal events.

Proceeding to compound expressions, the rules for conditionals and sequences are straightforward. Given that our semantics focuses on control, rules $if\-true$ and $if\-false$ are the only to query $mem$ expressions. The "magical" function $val$ receives the memory identifier and current reaction sequence number, returning the current memory value. Although the value is arbitrary, it is unique, because a given expression can execute only once within a reaction (remember that loops must contain $awaits$ which, from rules $await$ and $awaiting$, cannot awake in the same reaction they are reached).

The rules for loops are analogous to sequences, but use "$\ast$" as separators to properly bind breaks to their enclosing loops. When a program first encounters a $loop$, it first expands its body in sequence with itself (rule $loop\-expd$). Rules $loop\-adv$ and $loop\-nop$ are similar to rules $seq\-adv$ and $seq\-nop$, advancing the loop until they reach a $mem(id)$. However, what follows the loop is the loop itself (rule $loop\-nop$). Rule $loop\-brk$ escapes the enclosing loop, transforming everything into a $nop$.

The rules with the $par$ prefix are valid for all and/or/hor compositions (substituting the $par$ in the rules for each of them). The difference between the three parallel compositions consists only in how to deal with one of the sides terminating. The rules $par\-adv1$ and $par\-adv2$ force the transitions on the left branch $p$ to occur before transitions on the right branch $q$. These are the only rules that could lead to simultaneous transition options in the semantics. Therefore, the deterministic behavior relies on the $isBlocked$ predicate, defined in Figure 13 and used in rule $par\-adv2$, requiring the left branch $p$ to be blocked in order to allow the right transition from $q$ to $q'$. An expression becomes blocked when all of its trails in parallel hang in $awaiting$ and $emitting$ expressions. The rules $par\-brk1$ and $par\-brk2$ deal with a $break$ in each of the parallel sides. A $break$ terminates the whole composition to escape the innermost loop (strongly aborting the other side).

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

$$
\frac{}{S, \ await(id)} \longrightarrow [S, \ awaiting(id, n)] \quad (await)\\
\frac{id : S, \ awaiting(id, m)}{S, \ emit(id)} \longrightarrow [id : S, \ nop, \ m < n] \quad (awaiting)\\
\frac{S, \ emit(id)}{S, \ emitting([S])} \longrightarrow [S, \ nop] \quad (emit)\\
\frac{val(id, n) \neq 0}{S, (mem(id) ? p : q)} \longrightarrow [S, p] \quad (if\-true)\\
\frac{val(id, n) = 0}{S, (mem(id) ? p : q)} \longrightarrow [S, q] \quad (if\-false)\\
\frac{S, (p : q)}{S, (p : q)} \longrightarrow [S', (p' : q)] \quad (seq\-adv)\\
\frac{S, (mem(id) : q)}{S, (break : q)} \longrightarrow [S, break] \quad (seq\-nop)\\
\frac{S, (loop p)}{S, (loop p)} \longrightarrow [S, (p @ \ loop p)] \quad (loop\-expd)\\
\frac{S, (mem(id) @ \ idle)}{S, (mem(id) @ \ idle)} \longrightarrow [S, idle] \quad (loop\-nop)\\
\frac{S, (break @ \ idle)}{S, (break @ \ idle)} \longrightarrow [S, break] \quad (loop\-brk)\\
\frac{S, (p)}{S, (p)} \longrightarrow [S', (p') \ par] \quad (par\-adv1)\\
\frac{isBlocked(n, S, p), [S, q]}{S, (break \ par q)} \longrightarrow [S', (p \ par q)] \quad (par\-adv2)\\
\frac{S, (break \ par q)}{S, (break \ par q)} \longrightarrow [S, break] \quad (par\-brk1)\\
\frac{S, (p \ par \ break)}{S, (p \ par \ break)} \longrightarrow [S, break] \quad (par\-brk2)\\
\frac{S, (mem(id) \ and \ q)}{S, (p \ and \ mem(id))} \longrightarrow [S, q] \quad (and\-nop1)\\
\frac{S, (p \ and \ mem(id))}{S, (p \ and \ mem(id))} \longrightarrow [S, p] \quad (and\-nop2)\\
\frac{S, (mem(id) \ or \ q)}{S, (nop)} \longrightarrow [S, (nop)] \quad (or\-nop1)\\
\frac{isBlocked(n, S, p)}{S, (p \ or \ mem(id))} \longrightarrow [S, (nop)] \quad (or\-nop2)\\
\frac{\eta \neq (a \ hor \ b) \lor (a \ neq \ mem(v) \land b \ neq \ mem(v))}{[\eta], (mem(v) \ hor \ q)} \longrightarrow [\eta, \ nop] \quad (hor\-nop1)\\
\frac{\eta \neq (a \ hor \ b) \lor (a \ neq \ mem(v) \land b \ neq \ mem(v))}{[\eta], (p \ hor \ mem(v))} \longrightarrow [\eta, \ mem(v)] \quad (hor\-nop2)
$$

Figure 12: The semantics of Céu.
To describe a reaction chain, incrementing the sequence number: complete execution of a program is a series of “invocations” of reflexive transitive closure of this relation. Finally, the component behaves in reaction to a single external event, we use the par-adv1 condition, we check if there is a pending nested relationships only, and is less abstract in comparison to FRP.

A reaction chain eventually blocks in awaiting expressions should resume in the ongoing reaction, once their awaiting expressions in parallel trails. If all trails hangs only in becomes blocked: we define another relation to pop the stack if the program

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

6. REFERENCES


Figure 13: The recursive predicate isBlocked.

ond condition, we check if there is a pending nested hor, forcing it to transit before (via rules par-adv1 or par-adv2).

A reaction chain eventually blocks in awaiting and emitting expressions in parallel trails. If all trails hangs only in awaiting expressions, it means that the program cannot advance in the current reaction chain. However, emitting expressions should resume in the ongoing reaction, once their lower stack indexes are restored (see rule emit). Therefore, we define another relation to pop the stack if the program becomes blocked:

To describe a reaction chain in CÊU, i.e., how a program behaves in reaction to a single external event, we use the reflexive transitive closure of this relation. Finally, the complete execution of a program is a series of “invocations” of reaction chains, incrementing the sequence number:

5. RELATED WORK AND CONCLUSION

With respect to control-based languages for embedded systems, a number of synchronous alternatives to low-level event-driven systems have appeared [8, 9, 10, 1]. Protothreads [8] offer predictable and lightweight multi-threading with shared-memory concurrency, but lack thread composition and abortion (as described in Section 2.2). OSM [10] provides parallel synchronous state machines with support for composition and abortion. However, although machines can share memory, the execution order for side-effect operations among them is non-deterministic. Other related synchronous languages [9, 1] also rely on a deterministic scheduler for safe memory sharing, but do not differ from Esterel regarding event handling and thread composition.

Functional Reactive Programming (FRP) adapts functional languages to the reactive dataflow style [17]. In particular, Flask [11] shows that dataflow languages can also target constrained systems. Dataflow in CÊU is limited to static relationships only, and is less abstract in comparison to FRP.