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

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

General Terms
Design, Languages

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

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

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

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

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

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

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

2. OVERVIEW OF CÉU
Céu is a synchronous reactive language based on Esterel [6] with support for multiple concurrent lines of execution known as trails. By reactive, we mean that programs are stimulated by the environment through input events that are broadcast to all awaiting trails. By synchronous, we mean that any trail at any given time is either reacting to the current event or is awaiting another event; in other words, trails are always
of Céu occurring events are handled sequentially and uninterruptedly more closely reflects event-driven programming, in which occurrences are fundamental to enable advanced mechanisms in the synchronous execution of internal events (Section 2.3). By providing a precise control for concurrent lines of execution, these differences are essential for allowing adequate mechanisms in Céu (presented in Section 3).

2.1 External reactions and determinism
In Esterel, a reaction to the environment is constituted 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/or 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 seem 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 cannot 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 wreaking 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 r 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 Esterel, given the determinist 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 r never executes.

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

2.3 Internal events
Esterel makes no semantic distinctions between internal and external signals, both having only the notion of whether 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)
emit limitation that it cannot express recursive calls: an

On the one hand, this form of subroutines has a significant

time stacks. In Section 3.2, we take advantage of the lack

second, memory for locals is also bounded, not requiring run-

routines: first, they are guaranteed to react in bounded time;

not be awaiting itself. On the other hand, this very same

It may execute in any order after

may fail. The code is

read exceptions, we use an additional trail in Fig-

To handle read exceptions, we use an additional trail in Fig-

Figure 6 defines the read subroutine which performs the ac-

Céù, such as parallel compositions and the await statement.

In particular, they await keeping context information such as

3. Advanced Control Mechanisms

In this section, we explore the presented control primitives

3.1 Exception handling

Céù can naturally express different forms of exception mechani-

CONCLUSION

This mechanism can also support resumption if the excep-

This form of subroutines can use the other primitives of

Figure 2: Thread abortion in Esterel and Céù.

in Esterel and Céù. For the version in Esterel, given that

there is no control dependency between the calls to f, they

may execute in any order after A and B (internally emitted).

For the version in Céù, the occurrence of A makes the pro-

gram behave as follows (with the stack contents in italics):

1. 1st trail awakes (line 5), emits b, and pauses.
   
   stack: [lst]

2. 2nd trail awakes (line 9), calls _f(1), and terminates.
   
   stack: [lst]

3. 1st trail (on top of the stack) resumes, _f(2), and termin-
   
   stack: [[]]

4. Both trails have terminated, so the par/and rejoins, and
   
   the program also terminates;

Internal events bring support for a limited form of subrou-

tines, as depicted in Figure 4. The subroutine inc is defined

as a loop (lines 3-6) that continuously awaits its identifying

event (line 4), incrementing the value passed as reference

(line 5). A trail in parallel (lines 8-11) invokes the subro-

tine 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 (yield-

ing v=2), loops, and awaits the next “call” (line 4, again).

Only after this sequence that the calling trail resumes and

passes the assertion test.

On the one hand, this form of subroutines has a significant

limitation that it cannot express recursive calls: an emit to

itself will always be ignored, given that a running body can-

not be awaiting itself. On the other hand, this very same

limitation brings some important safety properties to sub-

routines: first, they are guaranteed to react in bounded time;

second, memory for locals is also bounded, not requiring run-

time stacks. In Section 3.2, we take advantage of the lack

of recursion to properly describe mutual dependency among

trails in parallel.

This form of subroutines can use the other primitives of

Figure 4: Subroutine inc is defined in a loop (lines 3-6), in parallel with the caller (lines 8-11).

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

This mechanism can also support resumption if the excep-

tion handler does not terminate its surrounding par/or (line

3 of Figure 7). For instance, the new handler of Figure 8
3.2 Dataflow programming

Reactive dataflow programming provides a declarative style to express dependency relationships among data. Figure 9 shows the dependency graph for the reactive expression $E < E+1$, which should always yield $true$. CÉu can express data dependency relying on par/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 $b$ to avoid yielding false (because $b$ also depends on $e1$). The code in the right of the graph implements it in CÉu. The first trail (lines 4-13) updates and signals $b$ when one of the two changes, $e1$ or $e$. The second trail (lines 15-19) updates and signals $e$ whenever $e$ changes. The par/or (lines 6-11) ensures that $b$ is only updated (in line 12) after $e1$ and $E$ (in lines 8 and 10). The program behavior for a reaction to $E=>1$ (which should awake lines 8 and 17) is the following:

1. Line 8 awakes and assigns $v1=1$. (The par/or cannot rejoin yet, allowing other trails to react.)
2. Line 17 awakes, emits $v1=2$, 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., $v1=1 < v2=2$).

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

Figure 10 shows a mutual conversion for temperatures in Celsius and Fahrenheit, so that whenever the value in one unit is set, the other is automatically recalculated (a problem proposed in [2]). Mutual dependency is another known issue in dataflow languages, usually requiring the placement of a specific delay operator to avoid runtime cycles [7, 15]. However, an explicit delay is somewhat ad hoc because it splits an internal dependency problem across two reactions to the environment. CÉu relies on the stack-based execution for internal events to avoid runtime cycles. The code in the right of the Figure 10 implements the conversion formula in CÉu. We first define the $tf$ 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.

Figure 7: Exceptions are caught with a par/or that strongly aborts the normal flow.
par/hor before aborting a nal events. It also ensures that a reaction becomes blocked operations and relies on an explicit stack to dispatch inter-

The semantics specifies a deterministic order for memory cudcising on the particular control aspects of the language.

code in 1st and 2nd trails in parallel and invoke await
dependencies. An actual application would run the dependency
internal events can unambiguously express mutual depen-
dency. As seen in step 4, the second emit tc=>0 (line 10), and pauses.
At the beginning of a reaction chain, the stack is initialized with the special η event and the occurring external event ext 
(S = [η, ext]), but emit expressions may push new events on top of it (we discuss how they are popped further). The event η is used as a special marker to check for and resume pending hor expressions before terminating the reaction.

We describe the relation with a set of small-step structural semantics rules, which are built in such a way that at most one transition is possible at any time, resulting in deterministic reaction chains. Figure 12 shows the transitions rules for the complete semantics of Céu.

An await is simply transformed into an awaiting that remembers the current external sequence number n (rule await).

An awaiting can only transit to a nop (rule awaiting) if its referred event id matches the top of the stack and its sequence number is smaller than the current one (n < n). An emit transits to an emitting holding the current stack level (|S| stands for the stack size), and pushes the referred event on the stack (in rule emit). With the new stack level |s : S| after an emit, the resulting emitting(|S|) cannot transit yet, as rule emitting expects its parameter |S| to match the cur-

Figure 11: Reduced syntax of Céu.

primitive represents all accesses, assignments, and C function calls that affect a memory location identified by id. As the challenging parts of Céu reside on its control structures, we are not concerned here with a precise semantics for side effects, but only with their occurrences in programs. The special notation nop is used to represent innocuous mem expressions (it can be thought as a synonym for mem(ε), where ε is an unused identifier). All other expressions map to their counterparts in the concrete language.

The core of our semantics is a relation that, given a sequence number n identifying the current reaction chain, maps a program p and a stack of events S in a single step to a modified program and stack:

\[
(S, p) \rightarrow_n (S', p')
\]

where

S, S' ∈ id* (sequence of event identifiers : [id_0, ..., id_i])
p, p' ∈ P (as described in the syntax above)
n ∈ N (univocally identifies a reaction chain)

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

4. THE SEMANTICS OF CÉU

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

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

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

Figure 10: A dataflow program with mutual dependency.
rent stack level. This trick provides the desired stack-based semantics for internal events.

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

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

The rules with the par prefix are valid for all and/or/hor compositions (substituting the par in the rules for each of them). The difference between the three parallel compositions consists only in how to deal with one of the sides terminating. The rules par-adv1 and par-adv2 force the transitions on the left branch p to occur before transitions on the right branch q. The deterministic behavior of the semantics relies on the isBlocked predicate, defined in Figure 13 and used in rule and-adv2, requiring the left branch p to be blocked in order to allow the right transition from q to q’. An expression becomes blocked when all of its trails in parallel hang in awaiting and emitting expressions.

The rules par-brk1 and par-brk2 deal with a break in each of the parallel sides. A break terminates the whole composition to escape the innermost loop (strongly aborting the other side).

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

A reaction chain eventually blocks in awaiting and emitting expressions in parallel trails. If all trails hang only in awaiting expressions, it means that the program cannot advance in the current reaction chain. However, emitting ex-

\[
\begin{align*}
(S, \text{await}(id)) & \rightarrow_n (S, \text{awaiting}(id, n)) & \text{(await)} \\
(id : S, \text{awaiting}(id, m)) & \rightarrow_n (id : S, \text{nop}), \ m < n & \text{(awaiting)} \\
(S, \text{emit}(id)) & \rightarrow_n (id : S, \text{emitting}([S])) & \text{(emit)} \\
(S, \text{emitting}([S])) & \rightarrow_n (S, \text{nop}) & \text{(emitting)}
\end{align*}
\]

\[
\begin{align*}
(S, \text{mem}(id) ? p : q) & \rightarrow_n (S, p) & \text{(if-true)} \\
(S, \text{mem}(id) ? p : q) & \rightarrow_n (S, q) & \text{(if-false)} \\
(S, p) & \rightarrow_n (S', p') & \text{(seq-adv)} \\
(S, p) & \rightarrow_n (S', q') & \text{(seq-nop)} \\
(S, \text{break} ; q) & \rightarrow_n (S, \text{break}) & \text{(seq-brk)} \\
(S, \text{loop} p) & \rightarrow_n (S, (p @ \text{loop} p)) & \text{(loop-expd)} \\
(S, p) & \rightarrow_n (S', p') & \text{(loop-adv)} \\
(S, \text{mem}(id) @ \text{loop} p) & \rightarrow_n (S, \text{loop} p) & \text{(loop-nop)} \\
(S, \text{break} @ \text{loop} p) & \rightarrow_n (S, \text{nap}) & \text{(loop-brk)} \\
(S, p) & \rightarrow_n (S', p') & \text{(par-adv1)} \\
(S, p @ \text{par} q) & \rightarrow_n (S', (p @ \text{par} q)) & \text{(par-adv2)} \\
(S, p @ \text{par} q) & \rightarrow_n (S', (p \text{ par} q)) & \text{(par-brk1)} \\
(S, \text{break} \text{ par} q) & \rightarrow_n (S, \text{break}) & \text{(par-brk2)} \\
(S, p @ \text{break}) & \rightarrow_n (S, \text{break}) & \text{(and-nop1)} \\
(S, \text{mem}(id) \text{ and} q) & \rightarrow_n (S, q) & \text{(and-nop2)} \\
(S, p \text{ and} \text{mem}(id)) & \rightarrow_n (S, p) & \text{(or-nop1)} \\
(S, \text{mem}(id) \text{ or} q) & \rightarrow_n (S, \text{nop}) & \text{(or-nop2)} \\
(S, p @ \text{mem}(id)) & \rightarrow_n (S, \text{nop}) & \text{(hor-nop1)} \\
p \neq (a \text{ hor} b) \lor (a \neq \text{mem}(v) \land b \neq \text{mem}(v)) & \rightarrow_n ([\eta], \text{mem}(v)) & \text{(hor-nop2)} \\
\end{align*}
\]

Figure 12: The semantics of Céu.
isBlocked(n, a : S, awaiting(b, m)) = (a ≠ b ∨ m = n)

isBlocked(n, S, emitting(s)) = (∃S ≠ s)

isBlocked(n, S, (p ; q)) = isBlocked(n, S, p)

isBlocked(n, S, (p @ loop q)) = isBlocked(n, S, p)

isBlocked(n, S, (p or q)) = isBlocked(n, S, p) ∧

isBlocked(n, S, q)

isBlocked(n, S, *) = false (mem, await, emit, break, if, loop)

Figure 13: The recursive predicate isBlocked.

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

6. REFERENCES


