Title of the paper

Subtitle of the paper

Francisco Sant'Anna   Noemi Rodriguez   Roberto Ierusalimschy

Departamento de Informática — PUC-Rio, Brasil

{fsantanna,noemi,roberto}@inf.puc-rio.br

Abstract

We present CÉU, a reactive language for embedded systems that prioritizes safety aspects for the development of reliable applications targeting highly constrained platforms.

CÉU supports concurrent lines of execution that run in time steps and are allowed to share variables. We formally describe the language and show how its synchronous and static nature enables a compile-time analysis to ensure that reactions to the environment are deterministic and execute with bounded memory and CPU time.

Nevertheless, CÉU does not renounce to practical aspects, providing seamless integration with C for low-level manipulation and a novel stacked execution policy for internal events that enables atypical mechanisms in the context of embedded systems, such as *finally blocks* and exception handling.

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

General Terms Design, Languages, Reliability

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

1. Introduction

Embedded systems are usually designed with safety and real-time requirements under constrained hardware platforms. At the same time, developers demand effective programming abstractions, ideally with unrestricted access to low-level functionality.

These particularities impose a challenge to embedded-language designers, who must provide a comprehensive set of features requiring correct and predicable behavior under platforms with limited memory and CPU. As a consequence, embedded languages either lack functionality or fail to offer a small and reliable programming environment.

This dilemma is notably evident in multithreading support for embedded systems, which implies a considerable overhead for synchronization primitives and per-thread stacks. Furthermore, preemptive multithreading is a potential source of safety hazards [18]. Alternative designs enforce cooperative scheduling to eliminate race conditions, but potentialize unbounded execution, breaking real-time responsiveness in programs [11]. Therefore, language designers have basically three options: not providing threads at all [14], affecting the productivity of programmers; providing restricted alternatives, such as disallowing locals in threads [12]; or preserving full support, but offering coarsed-grained concurrency only [8].

In this work, we present the design of CÉU[1], a reactive programming language that provides a reliable yet powerful programming environment for embedded systems. CÉU is based on Esterel [9] and follows a synchronous execution model [6], which enforces a disciplined step-by-step execution that enables lock-free concurrency. Both languages preclude the dynamic creation of lines of execution, as they employ static analysis in order to provide safety warranties. CÉU distinguishes from Esterel in basically two characteristics:

- Programs can only react to a *single* external event at a time.
- Internal events follow a *stacked* execution policy (like function calls in typical programming languages).

These design decisions are fundamental to introduce new functionalities into CÉU:

- From the uniqueness of external events, CÉU provides a static analysis that enables safe shared-memory concurrency.
- From the stacked execution of internal events, CÉU can derive many advanced control mechanisms, such as *finally blocks*, exception handling, and dataflow programming.

In our discussion, shared memory concerns not only variables, but also low-level accesses that ultimately use shared resources in the underlying platform (e.g., memory-mapped ports for I/O).

The stacked execution for internal events introduces support for a restricted non-recursive form of subroutines, resulting in memory-bounded programs that preclude stack overflows.

The proposed new functionalities are compliant with the safety and constrained requirements of embedded systems and, arguably, do not dramatically reduce the expressiveness of the language. As a limitation of the synchronous model, computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay hypothesis [20], and cannot be elegantly implemented in CÉU.

The implementation of CÉU offers fine-grained concurrency for highly constrained platforms. For instance, the current memory footprint under Arduino [3] is around 2 Kbytes of ROM and 50 bytes of RAM. A program with sixteen lines of execution (with least possible bodies) that synchronize on termination incur extra 270 bytes of ROM and 60 bytes of RAM.

The rest of the paper is organized as follows: Section 2 briefly introduces CÉU and describes it formally through an operational semantics. Section 3 demonstrates how the language can ensure deterministic reactions to the environment using bounded mem-

[1] Céu is the Portuguese word for *sky*.

```
 1:  input void BT1, BT2;   // external input events
 2:  event int clicked;     // an internal event
 3:  par/or do
 4:     loop do             // 1st trail
 5:        await BT1;
 6:        emit clicked(1);
 7:     end
 8:  with
 9:     loop do             // 2nd trail
10:        await BT2;
11:        emit clicked(-1);
12:     end
13:  with
14:     int diff = 0;      // 3rd trail
15:     loop do
16:        int v = await clicked;
17:        diff = diff + v;
18:        _printf("BT1 - BT2 = %d\n", diff);
19:        if diff < 0 then
20:            break;
21:        end
22:     end
23:  end
```

Figure 1. A concurrent program in Céu.

ory and CPU time. Section 4 shows how to implement some advanced control-flow mechanisms on top of the simpler semantics of internal events. Section 5 compares Céu to existing synchronous and asynchronous languages for embedded systems. Section 6 concludes the paper and makes final remarks.

2. The programming language Céu

Céu is a synchronous reactive language with support for multiple lines of execution known as *trails*. By reactive, we mean that programs are stimulated by the environment through input events that are broadcast to all awaiting trails. By synchronous, we mean that any trail at any given time is either reacting to the current event or is awaiting another event; in other words, trails are always synchronized at the current (and single) event.

In this work, we focus on a formal description of Céu which allows us to discuss the safety properties of programs, such as deterministic behavior. For an extensive and informal presentation with demos and examples of typical patterns found in embedded systems, refer to the technical report of Céu [22].

As an introductory example, the program in Figure 1 counts the difference between clicks in buttons `BT1` and `BT2` (represented as external input events), terminating when the number of occurrences of `BT2` is higher. The program is structured with three trails in parallel to illustrate the concurrent and reactive nature of Céu. The first and second trails react, respectively, to buttons `BT1` and `BT2` in a loop, while the third trail reacts to internal event `clicked`.

Lines 1-2 declare the events used in the program. An event declaration includes the type of value the occurring event carries. For instance, the two buttons are notify-only external input events (carrying no values), while `clicked` is an internal event that holds an integer value.

The `par/or` construct at line 3 spawns three trails in parallel (lines 4-7, 9-12, and 14-22). The loops in the first and second trails continuously wait for the referred buttons and notify their occurrences through the `clicked` event. The third trail holds the difference of clicks in local variable `diff` (line 14) and awaits for new occurrences of clicks in a loop. Whenever event `clicked` is emitted, the third trail awakes (line 16), updates the difference (line 17), prints it on screen[2] (line 18), and breaks the loop when it is negative (lines 19-21).

```
// primary primitives         // description
nop(v)                        (constant value)
mem                           (any memory access)
await(e)                      (event await)
emit(e)                       (event emit)
break                         (loop escape)

// compound statements
if mem then p else q          (conditional)
p ; q                         (sequence)
loop p                        (repetition)
p and q                       (par/and)
p or q                        (par/or)

// derived by semantic rules
defer(e,i)                    (deferred emit)
cont(i)                       (emit continuation)
p @ loop p                    (unwinded loop)
```

Figure 2. Simplified syntax of Céu.

Given the uniqueness of external events in Céu, the first and second trails never execute concurrently, and consequently, emits (and reactions) to event `clicked` are race free.

A `par/or` composition rejoins when any of its trails terminates; hence, only the termination of the third trail causes the termination of the program, as the other trails never terminate. (Céu also supports `par/and` compositions, which rejoin when *all* spawned trails terminate.)

The conjunction of parallelism with typical imperative primitives provides structured reactive programming and help in developing applications more concisely. In particular, the use of trails in parallel allows programs to wait for multiple events while keeping context information [1], such as local variables and the program counter.

One of the particularities of Céu is how internal and external events behave differently:

- External events can be emitted only by the environment, internal events only by the program.
- A single external event can be active at a time, while multiple internal events can coexist.
- External events are handled in queue, while internal events follow a stacked execution policy.

As an example of the stacked behavior for internal events, whenever the `emit` in line 11 of Figure 1 executes, its continuation (lines 12,9,10) is delayed until the awaken trail in line 16 completely reacts, either breaking the loop (line 20) or awaiting again (line 16).

Note that both internal and external events are unbuffered, i.e., at the moment an event occurs, only trails already awaiting can react to that instance.

2.1 Abstract syntax

Figure 2 shows the syntax for a subset of Céu that is sufficient to describe all semantic peculiarities of the language.

A nop represents a terminated computation associated with a constant value. The mem primitive represents any memory accesses, assignments, and C function calls. 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 occur-

[2] Céu can call C functions (such as `printf`) by prefixing names with an underscore.

rences in programs. We refer back to side effects when discussing determinism in Section 3.2.

The $await$ and $emit$ primitives are responsible for the reactive nature of CÉU. An $await$ can refer either to an external or internal event, while an $emit$ can only refer to an internal event.

The semantic rules to be presented generate three statements that the programmer cannot write: $defer$ and $cont$ avoid the immediate execution of an $emit$ and are used as an artifice to provide the desired stacked behavior for internal events. A $loop$ is expanded with the special '@' separator (instead of ';') to properly bind $break$ statements inside p to the enclosing loop.

2.2 Operational semantics

In the remaining of this section, we present an operational semantics to formally describe a *reaction chain* in CÉU, i.e., how a program behaves in reaction to a single external event.[3]

The semantics is split in two sets of rules: *big-step* and *small-step* rules. First, we apply a single big step to awake all awaiting trails matching the broadcast external event. Then, we continuously apply small steps until all trails await or emit an internal event. The next big step awakes all awaiting trails matching the deferred emits at once. The two set of rules are interleaved to achieve a complete reaction chain, terminating when the program either terminates or awaits in all trails.

In order to provide the desired stacked execution for internal events, the semantic rules are associated with an index i that represents the current runtime stack depth level. An emit creates a deeper level $i+1$ and is deferred to be matched in the next big step. The emit continuation (i.e., the statement that follows the emit) can only execute after the program completely reacts to the event and remains at stack level i. Awaken trails may emit new events that will increase the stack depth ($i+2$ and so on). Only after the stack unrolls to depth i that the original emitting trail can continue.

In the semantic rules, a $defer(e,i)$ represents a deferred emit on e to be matched at stack level i, while a $cont(j)$ represents a matched emit to continue at stack level j.

A complete reaction chain to an external event is formalized as follows:

$$(\quad p \xRightarrow[(i,E)=bcast(p)]{} p' \xrightarrow[i]{*} p'' \quad)*$$

A big step is represented with the double arrow, where the tuple (i,E) is the set of deferred emits E to be matched at stack depth i.

For the initial big step, function $bcast$ returns the single external event emitted at starting depth 0. For further big steps, $bcast$ is defined in Figure 3 and returns the deepest stack level and all deferred internal emits (if any) from the previous small-step sequence. Note that $bcast$ is only defined for blocked primitives ($await$, $defer$, and $cont$), as small-step sequences that precede big steps always reach these primitives (as discussed further).

A small-step sequence is represented with the single arrow and is associated with depth i from the previous big step, representing the current (deepest) stack depth.

The sets of rules are interleaved until one of the two possible terminating conditions for a reaction chain apply:

- The program is awaiting in all trails, i.e., function $bcast$ returns $(0,\{\})$.
- The program terminates, i.e., the small-step rules transform the whole program into a nop.

In Section 3.1 we show that reaction chains always reach one of these conditions in a finite number of steps, meaning that reactions to the environment always execute in bounded time.

[3] We could extend the semantics to describe the full execution of a program by holding new incoming external events in a queue and processing them in consecutive reaction chains that never overlap.

$$\begin{aligned}
bcast(await(e)) &= (0,\{\})\\
bcast(defer(i,e)) &= (i,\{e\})\\
bcast(cont(i)) &= (i,\{\})\\
bcast(p\ ;\ q) &= bcast(p)\\
bcast(p\ @\ loop\ q) &= bcast(p)\\
bcast(p\ and/or\ q) &= (max(j,k), F\cup G)\\
&\qquad where\ (j,F) = bcast(p)\\
&\qquad\quad and\ (k,G) = bcast(q)
\end{aligned}$$

Figure 3. The recursive definition for $bcast$.

$$\begin{aligned}
isBlocked(await(e)) &= true\\
isBlocked(defer(e,i)) &= true\\
isBlocked(cont(i)) &= true\\
isBlocked(p\ ;\ q) &= isBlocked(p)\\
isBlocked(p\ @\ loop\ q) &= isBlocked(p)\\
isBlocked(p\ and\ q) &= isBlocked(p) \land isBlocked(q)\\
isBlocked(p\ or\ q) &= isBlocked(p) \land isBlocked(q)\\
isBlocked(*) &= false\\
&\qquad (i.e.,\ nop, mem, break, if, loop)
\end{aligned}$$

Figure 4. The recursive predicate $isBlocked$.

To be compliant with the reactive nature of CÉU, we assume that all programs start awaiting the main event "$", which is emitted once by the environment on startup, i.e., $(i,E) = (0,\{\$\})$ for the very first big step.

2.2.1 Small-step rules

As briefly introduced, small-step rules continuously apply transformations to unblocked trails. A program becomes blocked when all parallel branches are hanged in $await$, $defer$, or $cont$ primitives, as defined in Figure 4.

All small-step rules are associated with the current (deepest) stack depth level i acquired from the previous big step.

We start with the small-step rules for primary primitives:

$$mem \xrightarrow[i]{} nop(?) \qquad \textbf{(mem)}$$

$$emit(e) \xrightarrow[i]{} defer\ (e, i+1) \qquad \textbf{(emit)}$$

A mem operation terminates with a nondeterministic value (e.g., the value of a variable).

An $emit$ is deferred with a deeper depth and blocks. As rule **emit** is the only one that increases the stack depth, function $bcast$ for the next big step will necessarily take all deferred emits to match against awaits.

All other primitives (nop, $break$, $await$, $defer$, and $cont$) represent terminated or blocked trails and, therefore, have no associated small-step rules.

The rules for conditionals and sequences are straightforward:

$$\frac{m \underset{i}{\longrightarrow} m'}{(if\ m\ then\ p\ else\ q) \underset{i}{\longrightarrow} (if\ m'\ then\ p\ else\ q)} \quad \textbf{(if-adv)}$$

$$(if\ nop(v)\ then\ p\ else\ q) \underset{i}{\longrightarrow} p\ , \quad (v \neq 0) \quad \textbf{(if-true)}$$

$$(if\ nop(0)\ then\ p\ else\ q) \underset{i}{\longrightarrow} q \quad \textbf{(if-false)}$$

$$\frac{p \underset{i}{\longrightarrow} p'}{(p\ ;\ q) \underset{i}{\longrightarrow} (p'\ ;\ q)} \quad \textbf{(seq-adv)}$$

$$(nop\ ;\ q) \underset{i}{\longrightarrow} q \quad \textbf{(seq-cst)}$$

$$(break\ ;\ q) \underset{i}{\longrightarrow} break \quad \textbf{(seq-brk)}$$

Given that the semantics focus on control, note that rules **if-true** and **if-false** are the only to query *nop* values. For all other rules, we omit these values (e.g., **seq-cst**).

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

$$(loop\ p) \underset{i}{\longrightarrow} (p\ @\ loop\ p) \quad \textbf{(loop-expd)}$$

$$\frac{p \underset{i}{\longrightarrow} p'}{(p\ @\ loop\ q) \underset{i}{\longrightarrow} (p'\ @\ loop\ q)} \quad \textbf{(loop-adv)}$$

$$(nop\ @\ loop\ p) \underset{i}{\longrightarrow} loop\ p \quad \textbf{(loop-cst)}$$

$$(break\ @\ loop\ p) \underset{i}{\longrightarrow} nop \quad \textbf{(loop-brk)}$$

When a program first encounters a *loop*, it first expands its body in sequence with itself (rule **loop-expd**). Rules **loop-adv** and **loop-cst** are similar to rules **seq-adv** and **seq-cst**, advancing the loop until it reaches a *nop*. However, what follows the loop is the loop itself (rule **loop-cst**). Rule **loop-brk** escapes the enclosing loop, transforming everything into a *nop*. Note that if we used ';' as a separator in loops, rules **loop-brk** and **seq-brk** would conflict.

The small-step rules for parallel compositions advance trails independently and require that both sides either block or terminate to advance to the next big step.

For an *and*, if one of the sides terminate, the composition is simply substituted by the other side. For an *or*, if one of the sides terminate, the other side must advance until it blocks. However, the whole composition terminates and the blocked side is *killed*.

A similar situation occurs when one of the sides of an *and/or* reaches a *break*. In this case, the enclosing *loop* must terminate and the other side is killed as soon as it blocks, transforming the whole composition into a *break*.

Follow the rules for a parallel *and*:

$$\frac{p \underset{i}{\longrightarrow} p'}{(p\ and\ q) \underset{i}{\longrightarrow} (p'\ and\ q)} \quad \textbf{(and-adv1)}$$

$$\frac{q \underset{i}{\longrightarrow} q'}{(p\ and\ q) \underset{i}{\longrightarrow} (p\ and\ q')} \quad \textbf{(and-adv2)}$$

$$(nop\ and\ q) \underset{i}{\longrightarrow} q \quad \textbf{(and-cst1)}$$

$$(p\ and\ nop) \underset{i}{\longrightarrow} p \quad \textbf{(and-cst2)}$$

$$\frac{q = break\ \vee\ isBlocked(q)}{(break\ and\ q) \underset{i}{\longrightarrow} break} \quad \textbf{(and-brk1)}$$

$$\frac{p = break\ \vee\ isBlocked(p)}{(p\ and\ break) \underset{i}{\longrightarrow} break} \quad \textbf{(and-brk2)}$$

Rules **and-cst1** and **and-cst2** handle the termination of one of the sides, substituting the whole composition by the other side. Rules **and-brk1** and **and-brk2** handle the special case for reaching a *break*, in which the other blocked side is killed and the whole composition becomes a *break* to terminate the enclosing loop. For instance, the program `loop(break and emit(a))` never emits a, because the *emit* blocks (rule **emit**) and is then killed (rule **and-brk1**).

The rules for a parallel *or* are slightly different:

$$\frac{p \underset{i}{\longrightarrow} p'}{(p\ or\ q) \underset{i}{\longrightarrow} (p'\ or\ q)} \quad \textbf{(or-adv1)}$$

$$\frac{q \underset{i}{\longrightarrow} q'}{(p\ or\ q) \underset{i}{\longrightarrow} (p\ or\ q')} \quad \textbf{(or-adv2)}$$

$$\frac{q = nop\ \vee\ isBlocked(q)}{(nop\ or\ q) \underset{i}{\longrightarrow} nop} \quad \textbf{(or-cst1)}$$

$$\frac{p = nop\ \vee\ isBlocked(p)}{(p\ or\ nop) \underset{i}{\longrightarrow} nop} \quad \textbf{(or-cst2)}$$

$$\frac{q = nop\ \vee\ q = break\ \vee\ isBlocked(q)}{(break\ or\ q) \underset{i}{\longrightarrow} break} \quad \textbf{(or-brk1)}$$

$$\frac{p = nop\ \vee\ p = break\ \vee\ isBlocked(p)}{(p\ or\ break) \underset{i}{\longrightarrow} break} \quad \textbf{(or-brk2)}$$

Rules **or-cst1** and **or-cst2** terminate the *or* when at least one of the sides is a *nop*. For instance, the program `loop(nop and emit(a))` never emits a, because the *emit* blocks (rule **emit**) and is then killed (rule **or-cst1**). Rules **or-brk1** and **or-brk2** are similar to their *and* counterparts, with the additional remark that a *break* has preference over a *nop*, i.e., when they appear on each side, the whole composition becomes a *break* instead of a *nop*.

Note that rule **mem** and the pairs **and-adv1/and-adv2** and **or-adv1/or-adv2** bring nondeterminism to the semantics of CÉU. However, in Section 3.2 we discuss how to detect programs with deterministic behavior (even with nondeterminism in *mem* operations and scheduling), refusing all other programs at compile time.

2.2.2 Big-step rules

The big-step semantics matches deferred emits with corresponding awaiting trails, providing broadcast communication in the language. It is important to use a big-step operational semantics in order to apply transformations in parallel, all at once. Deferred emits with no matching awaits are simply discarded, characterizing the unbuffered communication typically adopted in synchronous languages.

Big-step rules are associated with a tuple (i, E) that represents the set of occurring events E triggered at stack depth i.

The rules for $await$ check if deferred emits match awaiting trails. An $await$ either awakes or remains awaiting, depending on the set of current deferred events:

$$await(e) \underset{(i,E)}{\Longrightarrow} nop\ , \quad (e \in E) \qquad \textbf{(Await-awk)}$$

$$await(e) \underset{(i,E)}{\Longrightarrow} await(e)\ , \quad (e \notin E) \qquad \textbf{(Await-rep)}$$

The rules for $defer$ and $cont$ provide the desired stacked semantics for internal events:

$$defer\ (i, e) \underset{(i,E)}{\Longrightarrow} cont\ (i) \qquad \textbf{(Defer)}$$

$$cont\ (i) \underset{(i,E)}{\Longrightarrow} nop \qquad \textbf{(Cont)}$$

A deferred emit cannot unblock immediately because the just matched trails (rule **Await-awk**) must all react before the emit continuation proceeds. Hence, rule **Defer** specifies that the emit continuation remains blocked at the same (low) depth. This way, awaken trails will execute in the next small-step sequence and new emits will be deferred at deeper depths (note the `i+1` in small-step rule **emit**). Eventually, no new emits will take place, and function $bcast$ will pop the continuations with the highest depth, which will proceed through rule **Cont**, providing the desired stacked execution for internal events.

To conclude the big-step semantics, the rules for compound statements advance their subparts all at once:

$$\frac{p \underset{(i,E)}{\Longrightarrow} p'}{(p\ ;\ q) \underset{(i,E)}{\Longrightarrow} (p'\ ;\ q)} \qquad \textbf{(Seq)}$$

$$\frac{p \underset{(i,E)}{\Longrightarrow} p'}{(p\ @\ loop\ q) \underset{(i,E)}{\Longrightarrow} (p'\ @\ loop\ q)} \qquad \textbf{(Loop)}$$

$$\frac{p \underset{(i,E)}{\Longrightarrow} p' \qquad q \underset{(i,E)}{\Longrightarrow} q'}{(p\ and\ q) \underset{(i,E)}{\Longrightarrow} (p'\ and\ q')} \qquad \textbf{(And)}$$

$$\frac{p \underset{(i,E)}{\Longrightarrow} p' \qquad q \underset{(i,E)}{\Longrightarrow} q'}{(p\ or\ q) \underset{(i,E)}{\Longrightarrow} (p'\ or\ q')} \qquad \textbf{(Or)}$$

Note that there are no rules for mem, $break$, $emit$, nop, and if because none of these represent blocked primitives, and hence, never appear in a big step.

3. Safety warranties

A primeval goal of CÉU is to ensure a reliable execution for shared-memory programs. In this section, we demonstrate how CÉU can ensure at compile time that reaction chains are deterministic and require bounded resources (memory and CPU time).

3.1 Bounded execution

Reactions to the environment should run in bounded time to guarantee that programs are responsive and can handle upcoming input events. Similarly to Esterel [9], CÉU requires that each possible path in a loop body contains at least one `await` or `break` statement, thus ensuring that loops never run in unbounded time.

Consider the examples that follow:

```
loop do                      loop do
    if cond then                 if cond then
        break;                       break;
    end                          else
    ...                              await A;
end                              end
                                 ...
                             end
```

The first example is refused at compile time, because the `if` true branch may never execute, resulting in a *tight loop* (i.e., an infinite loop that does not await). The second variation is accepted, because for every iteration, the loop either breaks or awaits.

Given that programs with tight loops are refused at compile time, it is easy to show that the small-step semantics always reaches a state in which all trails are either blocked or terminated: all small-step rules advance to the blocked conditions of Figure 4, except for rule **loop-expd** which expands code. However, the compile-time restriction ensures that all trails inside a loop either break (reducing the whole expansion to a nop via rule **loop-brk**) or block, as desired.

We still need to show that interleaving big steps and small steps does not lead to a runtime cycle involving emits and awaits in loops in parallel. For instance, the program that follows specifies mutual dependency among trails through non-tight loops:

```
01:   par/and do
02:       loop do
03:           emit e;
04:           await f;
05:       end
06:   with
07:       loop do
08:           await e;
09:           emit f;
10:       end
11:   end
```

As loops must contain at least one $await$, the only way to provoke runtime cycles is to include an $emit$ that awakes a trail in parallel, expecting that trail to awake the $await$ in sequence (lines 3-4 or 8-9).

However, the proposed example contains no runtime cycles, behaving as follows:

- *small steps (i=0):* 1st trail emits e and blocks at $defer(e, 1)$ (rule **emit**). 2nd trail awaits e. Both trails are blocked.
- *big step (i=1):* The emit and await are matched: 1st trail blocks at $cont(1)$ (rule **Defer**). 2nd trail awakes.
- *small steps (i=1):* 1st trail remains blocked at $cont(1)$. 2nd trail emits f and blocks at $defer(f, 2)$ (rule **emit**). Both trails are blocked.
- *big step (i=2):* The emit does not match an await and is lost. 1st trail remains blocked at $cont(1)$. 2nd trail blocks at $cont(2)$ (rule **Defer**).
- *small steps (i=2):* Both trails are blocked.
- *big step (i=2):* 1st trail remains blocked at $cont(1)$. 2nd trail resumes (rule **Cont**).
- *small steps (i=2):* 1st trail remains blocked at $cont(1)$. 2nd trail loops an awaits e. Both trails are blocked.
- *big step (i=1):* 1st trail resumes (rule **Cont**). 2nd trail remains blocked at $await(e)$.
- *small steps (i=1):* 1st trail loops an awaits f. 2nd trail is blocked at $await(e)$. Both trails are blocked.
- *big step (i=0):* Both trails are awaiting, the reaction chain terminates.

Given the stacked execution for internal events, an $await$ that follows an $emit$ can never awake as a consequence of that $emit$, making runtime cycles impossible: From big-step rule **Defer**, a deferred $emit$ at depth i remains blocked at $cont(i)$ and cannot continue to an $await$ in sequence. Then, all trails that awake from that $emit$ will have the chance to execute before the continuation. Therefore, any new emits will be deferred at $i + 1$ (small-step rule **emit**) and the next big step will consume these emits before $cont(i)$ has the chance to reach the $await$ in sequence. When the $await$ is finally reached, it is impossible to have a pending $emit$ caused by the original $emit$.

Enforcing bounded execution makes CÉU inappropriate for algorithmic-intensive applications that require unrestricted loops (e.g., cryptography, image processing). However, CÉU is designed for real-time control-intensive applications and we believe this is a reasonable price to pay in order to achieve higher reliability.

Note that CÉU does not extend the bounded execution analysis for C function calls, which are left as responsibility for the programmer. On the one hand, C calls must be carefully studied in order to keep programs responsive. On the other hand, they also give the programmer means to circumvent the rigor of CÉU.

3.2 Deterministic behavior

Providing deterministic schedulers is a selling point of many concurrent designs. For instance, event-driven systems usually employ a *FIFO* policy for event handlers, while in cooperative multithreading the programmer himself determinates an order of execution among tasks. Even systems with preemptive multithreading can offer deterministic execution for programs [19].

As discussed in Section 2.2.1, the small-step semantic rules for parallel compositions do not specify the exact order in which trails execute, leading to nondeterministic execution in CÉU. Note that a slight modification to rules **and-adv1/or-adv1** or **and-adv2/or-adv2** could force one trail to execute before any advance on the other, thus enforcing a deterministic policy for the scheduler. However, we believe that any arbitrary order should be avoided, because an apparently innocuous reordering of trails would modify the semantics of the program.

CÉU takes a different approach and only accepts programs with deterministic *behavior*, regardless of their nondeterministic *execution* (scheduling). At compile time, we run a symbolic interpretation of the program that creates an acyclic graph of all mem operations that execute in a reaction chain. If any two mem operations access the same memory area and one is not ancestor of the other, then the program is nondeterministic and is refused. The interpretation is repeated for every possible reaction chain the program can reach.

In a reaction graph, nodes represent mem operations and are connected through edges representing causality in the semantics of CÉU:

- Sequences connect two subgraphs with an edge.
- Conditionals depend on mem operations which the symbolic interpreter cannot evaluate deterministically. For this reason, the whole graph is duplicated and each copy proceeds to one of the conditional branches.
- Loops behave as sequences during runtime.
- Parallel compositions spawn two subgraphs that are interpreted independently. The subgraphs are determinate, given that small-step rules for rejoins require both sides to be terminated or blocked.
- Emits, awaits and terminating trails rejoin the graph and represent the termination of a small-step sequence. The big step that follows re-forks the graph on each awaking trail.

A graph is evidently acyclic and finite, because the presented rules create no cycles and programs execute in bounded time (as discussed in previous section). Also, a graph is univocally represented by the set of external and internal events the program is awaiting after the symbolic interpretation. Therefore, the algorithm

```
input void A;
int v;
par/and do
  loop do
    await A; // A11
    await A; // A12
    v = 1;
  end
with
  loop do
    await A; // A21
    await A; // A22
    await A; // A23
    v = 2;
  end
end
```

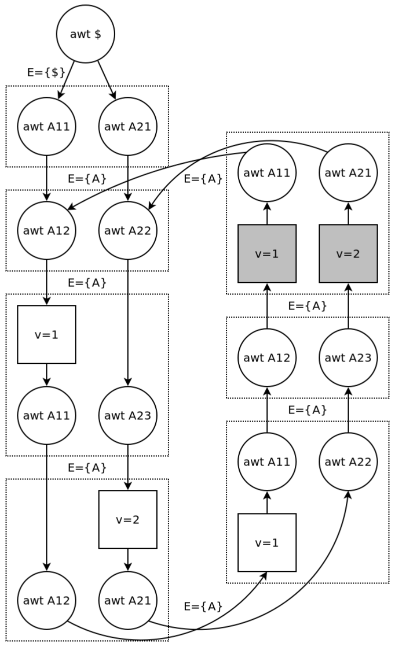

Figure 5. A nondeterministic program in CÉU and its reaction graphs.

always terminates, given that a graph construction runs in bounded time and the maximum number of graphs is finite (the number of combinations of $await$ statements).

As an example, consider the program and corresponding reaction graphs in Figure 5. Each dashed box represents a reaction graph; each set of circles identifies a graph; and each square represents a mem operation. Starting from the program awaiting the event \$, the algorithm computes reaction chains to all external events (the example only uses event A). The algorithm detects that variable v is assigned in parallel paths after six occurrences event A, and the program is refused at compile time.

Unfortunately, the described algorithm is exponential on the number of conditionals and awaits in a program. Even so, it is applicable for many reasons:

- Embedded programs are usually small, not being affected by the exponential growth.
- Many programs are safety-critical and must provide as much warranties as possible.
- The algorithm is easily parallelizable, given that reaction chains do not depend on each other.
- The development phase *per se* does not require safety warranties, reducing considerably the number of times the algorithm has to be executed.

An orthogonal problem to building reaction graphs is to classify mem operations that can be safely executed in parallel paths, avoiding false positives in the analysis. For instance, mem operations that accesses different variables can obviously execute concurrently. However, remember from Section 2.1 that the mem primitive represents not only read & write access to variables, but also C function calls. Moreover, CÉU also supports pointers, which are required for low-level manipulation (e.g., accessing buffers from device drivers).

CÉU enforces a default policy for mem operations as follows: If a variable is written in a path, then a path in parallel cannot read or write to that variable, nor dereference a pointer of that variable type. An analogous policy is applied for pointers vs variables and pointers vs pointers. Any two C calls cannot appear in parallel paths, as CÉU has no knowledge about their side effects. Also, passing variables as parameters count as read access to them, while passing pointers count as write access to those types (because functions may dereference and assign to them).

This policy may still yield some false positives in the analysis. For instance, the rule for C calls is particularly restrictive, as many functions can be safely called concurrently. Therefore, CÉU supports syntactic annotations that the programmer can use to relax the policy explicitly:

- The `pure` modifier declares C functions that do not perform side effects, being allowed to be called concurrently with any other function in the program.
- The `det` modifier (for *deterministic*) declares pairs of variables (e.g., pointers) or functions that do not affect each other, being allowed to be used concurrently.

The following code illustrates CÉU annotations:

```
pure  _abs;             // 'abs' is side-effect free
det   _led1 with _led2; // 'led1' vs 'led2' is ok
int*  buf1, buf2;       // point to different memory
det   buf1 with buf2;   // 'buf1' vs 'buf2' is ok
```

To summarize this section, CÉU only accepts programs with *deterministic behavior*, requiring that any two mem operations identified as incompatible (by the prevailing policy) can only execute as a causal relation between each other in any possible state the program can reach during runtime. Therefore, deterministic behavior, as we define, does not require deterministic scheduling and can be statically inferred with the presented algorithm.

3.3 Bounded memory

CÉU favors a fine-grained use of trails, being common to use trails that await a single event. For this reason, CÉU does not allocate per-trail stacks; instead, all locals reside in fixed memory slots held in a static one-dimension vector. Locals for trails in parallel must coexist in memory, while statements in sequence can share space.

The memory in CÉU can be precisely calculated, given that programs are defined as hierarchies of control-flow statements with explicit forks and joins for trails. This contrasts with threads, which are defined detached from the program hierarchy (e.g., a function defined in separate) and requires manual bookkeeping (e.g. creation, synchronization, etc.), hindering automatic memory prediction and management.

Another concern regarding memory consumption is the runtime stack for internal events. Note that during runtime, a trail can only occupy one position in the stack, given that an emit pauses the trail until the stack unwinds and recursion is impossible. Hence, in the worst case, the runtime stack size is the maximum number of trails in parallel containing an `emit` statement, which is also trivially calculated from the program text.

Besides C calls, which are not under control of CÉU, the other possible point of failure regarding memory consumption is the queue for external events. High-frequency external events may fill up the queue before the program can react to them, even with the guaranteed bounded execution. For projects that must deal with event bursts, CÉU delegates the queue management to the underlying system, which can provide its own policy for adjusting the queue size, prioritizing events, or signaling the program about overflows (e.g., through a custom event).

4. Advanced control mechanisms

In this section we explore the stacked execution for internal events in CÉU, demonstrating how it enables many advanced control-flow mechanisms without requiring new primitives in the language.

Although the described mechanisms involve thoughtful techniques, keep in mind that they can be easily abstracted with compile-time macros taking advantage of the structured style of CÉU[4]. As an exception, the `do-finally` construct to be presented in Section 4.2 makes slight global additions to the program tree and requires a dedicated syntax.

```
event int f;         event void g;       event void h,i;
par/or do            par/or do           par/or do
  loop do              loop do             loop do
    int v=await f;       await g;            await h;
    ...                  ...                 ...
  end                    emit g;             await i;
with                   end                 end
  emit f(1);         with                with
  emit f(2);           emit g;             emit h;
end                    emit g;             emit h;
                     end                 end

    (6.a)                (6.b)               (6.c)
  subroutine         non-recursive       single instance
```

Figure 6. Subroutines in CÉU.

4.1 Subroutines

Internal events introduce support in CÉU for a limited form of subroutines, as illustrated in Figure 6.a and depicted as follows:

- The subroutine is represented as a loop that awaits an identifying event (e.g., `await f`).
- The subroutine is called in a parallel trail through an emit on the corresponding event (e.g., `emit f`).
- The parameter of the subroutine is the type of its corresponding event (e.g., `event int f`).

In the example, the second trail invokes `emit f(1)` to "call" subroutine `f`. Given the stacked execution, the calling trail pauses and awakes the subroutine in the first trail. The subroutine executes its body, loops, and awaits function `f` to be "called" again. Then, the second trail resumes and calls `f` in sequence, repeating the behavior just described. In the examples, we assume the subroutine bodies represented as `...` do not contain `await` statements.

The other examples in Figure 6 show how this form of subroutines is non-recursive (recursive calls have no effect), and can only run a single instance at a time (i.e., calls to running subroutines have no effect).

The example 6.b fails to make the recursive call in the first trail, because the subroutine is not awaiting event `g` in the moment its body calls itself. Remember that events are unbuffered in CÉU. The recursive call fails, but the subsequent call in the second trail behaves normally.

In example 6.c, the first trail calls subroutine `h`, which awaits event `i` in its body. Then, control returns to the second trail, which calls the subroutine again. However, the subroutine did not complete and the second call is missed.

In Section 4.4, we show that we can even take advantage of non-recursive subroutines to properly describe mutual dependency among trails in parallel.

4.2 Finally blocks

Finally blocks (as found in $Java$ and $C\#$) are often useful to handle dynamic resource allocation in a structured way. As an example, the naive program in CÉU that follows allocates a block of memory and uses it across reactions to events before freeing it:

```
input void A,F;
par/or do
    _t* ptr = _malloc(...);
    ... // use 'ptr'
```

[4] Our programs in CÉU make extensive use of the *m4* preprocessor.

```
input void A,F;                input void A,F;
                               event void $fin;      (1)
par/or do                      par/or do
  do                             par/and do          (2)
    _t* ptr = _malloc();           _t* ptr = _malloc();
    ... // use ‘ptr’               ... // use ‘ptr’
    await A;                       await A;
    ... // use ‘ptr’               ... // use ‘ptr’
                                   emit $fin;        (3)
  finally                        with
                                   await $fin;       (4)
    _free(ptr);                    _free(ptr);
  end                            end
with                           with
  await F;                       await F;
                                 emit $fin;          (5)
end                            end
```

Figure 7. do-finally code and corresponding translation.

```
        await A;
        ... // use ‘ptr’
        _free(ptr);
    with
        await F;
    end
    ...     // program continues
```

In the code, if event F occurs before A, the par/or composition terminates and does not free the allocated memory, leading to a leak in the program.

CÉU provides a do-finally construct to ensure the execution of a block of code to safely release resources. The previous example can be rewritten as the code in the left side of Figure 7, which forces the execution of the block after the finally keyword, even when the outer par/or terminates.

do-finally constructs do not add any complexity to the semantics of CÉU, relying only on the set of primitives already presented in Section 2.1. For instance, the example is translated at compile time into the code shown in the right side of the figure, as follows:

1. A unique global internal event $fin is declared.[5]
2. The do-finally is converted into a par/and.
3. The first par/and trail emits $fin on termination to invoke the finally block.
4. The second par/and trail (the finally block) awaits $fin to start executing.
5. All trails that terminate a par/or or escape a loop emit $fin to also invoke the finally block.

We opted for a dedicated syntax given that the transformation is not self-contained, affecting the global structure of programs.

The cases that follow illustrate the precise behavior of finally blocks when a third trail in parallel encloses a do-finally construct and kills it:

- *3rd trail terminates before the finally block starts to execute.* In this case, 3rd trail emits the corresponding $fin, which is not yet being awaited for, and the finally does not execute.
- *3rd trail terminates while the do-finally is blocked.* In this case, the resource has been acquired but not released. The corresponding $fin is emitted and holds 3rd trail to awake the finally, which safely releases the resource before resuming the terminating trail.
- *3rd trail terminates concurrently with the do-finally.* (Suppose they react to the same event.) In this case, both trails emit $fin, executing the finally only once, as expected.

[5] Each do-finally is associated to an unique event (e.g., $fin_1, $fin_2, etc.).

finally blocks have the restriction that they cannot await events, otherwise they would be killed by the terminating trail before releasing the acquired resources. However, releasing resources does not typically involve awaiting.

4.3 Exception handling

Exception handling can be provided by specialized programming language constructs (e.g., try-catch blocks in Java), but also with techniques using standard control-flow primitives (e.g., setjmp/longjmp in C). CÉU can naturally express different forms of exception handling without a specific construct.

As an illustrative example, suppose an external entity periodically writes to a log file and notifies the program through the event ENTRY, which carries the number of available characters to read.

We start from a simple and straightforward specification to handle log entries assuming no errors occur. The normal flow is to open the file and wait in a loop for ENTRY occurrences. In our implementation, the low-level file operations open and read are internal events working as subroutines:

```
// DECLARATIONS
input int ENTRY;  // callback event for log entries
_FILE*    f;      // holds a reference to the log
char[10] buf;     // holds the current log entry
event char* open; // opens filename into ‘f’
event int read;   // reads a number of bytes into ‘buf’
event int excpt;  // callback event for exceptions

// NORMAL FLOW
do
  emit open("log.txt");
  loop do
    int n = await ENTRY;
    emit read(n);               // reads into global ‘buf’
    _printf("log: %s\n", buf); // handles the log string
  end
finally
  if f != _NULL then
    _fclose(f);
  end
end
```

We use a finally block to safely close the file in the case of abrupt terminations, as discussed in previous section.

Emits to open and read behave just like conventional subroutines, as discussed in Section 4.1. The operations that perform the actual low-level system calls are placed in parallel and possibly emit exceptions through event excpt:

```
// DECLARATIONS (as in previous code)
par/or do
    // NORMAL FLOW (as in previous code)
with
    loop do     // OPEN subroutine
        char* filename = await open;
        f = _open(filename);
        if f == _NULL then
            emit excpt(1);  // 1 = open exception
        end
    end
with
    loop do     // READ subroutine
        int n = await read;
        if (n > 10) || (_read(f,buf,n) != n) then
            emit excpt(2);  // 2 = read exception
        end
    end
end
```

To handle exceptions, we enclose the normal flow with another par/or to terminate it on any exception thrown by file operations:

```
// DECLARATIONS
par/or do
    par/or do
        // NORMAL FLOW
    with
        await excpt;    // catch exceptions
    end
with
    // OPERATIONS       // throw exceptions
end
```

To illustrate how the program behaves on an exception, suppose the normal flow tries to read a string and fails. The program behaves as follows (with the stack in emphasis):

1. Normal flow invokes the read operation (`emit read`) and pauses;
   stack: [norm]
2. Read operation awakes, throws an exception (`emit excpt`), and pauses;
   stack: [norm, read]
3. Exception handler (`await excpt`) awakes, invokes the `finally` (through implicit `emit $fin`), and pauses;
   stack: [norm, read, hdlr]
4. The `finally` block executes, closes the file, and terminates;
   stack: [norm, read, hdlr]
5. The exception continuation terminates the `par/or`, cancelling all remaining paused continuations.
   stack: []

Exceptions in CÉU can also be recoverable if the handler does not terminate its surrounding `par/or`. For instance, the new handler that follows waits for exceptions in a loop and recovers from each type of exception:

```
...
    par/or do
        // NORMAL FLOW
    with
        loop do
            int err = await excpt;  // catch exceptions
            if err == 1 then        // open exception
                f = <creates a new file>
            else/if err == 2 then   // read exception
                buf = <assigns a default string>
            end
        end
    end
...
```

Now, step 3 in the previous execution trace would not fire the `finally` block, but instead, assign a default string to `buf`, loop and await the next exception. Then, the exception continuation would loop and await further file operations. In the end, the read operation would resume as if no exceptions had occurred.

Note that throughout the example, the normal flow remained unchanged, with all machinery to handle exceptions placed around it.

In terms of memory usage, switching from the original normal flow (without exception throws) to the last example (with recovery) incurred extra 450 bytes of ROM and 24 bytes of RAM.

The presented approach for exceptions has the limitation that file operations and exception handlers cannot await other events, which is related to the single-instance property of subroutines in CÉU.

4.4 Dataflow programming

Reactive dataflow programming [4] provides a declarative style to express dependency relationships among data. A known issue in dataflow languages is on handling mutual dependency, which requires the explicit placement of a specific delay operator to avoid runtime cycles [10, 21]. This solution is somewhat *ad hoc* and splits an internal dependency problem across two reactions to the environment.

```
 1:   int tc, tf;
 2:   event int tc_evt, tf_evt;
 3:   par/or do
 4:      loop do            // 1st trail
 5:         tc = await tc_evt;
 6:         emit tf_evt(9 * tc / 5 + 32);
 7:      end
 8:   with
 9:      loop do            // 2nd trail
10:         tf = await tf_evt;
11:         emit tc_evt(5 * (tf-32) / 9);
12:      end
13:  with
14:      emit tc_evt(0);     // 3rd trail
15:      emit tf_evt(100);
16:  end
```

Figure 8. A dataflow program with mutual dependency.

CÉU can naturally express safe mutual dependencies, making it impossible to implement recursive definitions (as shown in Section 3.1). For instance, the program in Figure 8 applies the temperature conversion formula between Celsius and Fahrenheit [4], so that whenever the value in one unit is set, the other is automatically recalculated.

We first define the variables to hold the temperatures and corresponding internal events (lines 1-2). Any change to a variable in the program must be signalled by an emit on the corresponding event so that dependent variables can react. Then, we create two trails to await for changes and update the dependency relations among the temperatures. For instance, the first trail is a `loop` (lines 4-7) that waits for changes on `tc_evt` (line 5) and signals the conversion formula to `tf_evt` (line 6). The behavior for the second trail that awaits `tf_evt` (lines 9-12) is analogous. The third trail (lines 14-15) updates the temperatures twice in sequence. The program behaves as follows (with the stack in emphasis):

1. 1st and 2nd trail await `tc_evt` and `tf_evt`;
   stack: []
2. 3rd trail signals a change to `tc_evt` and pauses;
   stack: [3rd]
3. 1st trail awakes, sets `tc=0`, emits `tf_evt`, and pauses;
   stack: [3rd,1st]
4. 2nd trail awakes, sets `tf=32`, emits `tc_evt`, and pauses;
   stack: [3rd,1st,2nd]
5. no trails are awaiting `tc_evt` (1st trail is paused), so 2nd trail (on top of the stack) resumes, loops, and awaits `tf_evt` again;
   stack: [3rd,1st]
6. 1st trail resumes, loops, and awaits `tc_evt` again;
   stack: [3rd]
7. 3rd trail resumes and now signals a change to `tf_evt`;
   stack: [3rd]
8. ... (analogous behavior)

5. Related work

CÉU is strongly influenced by Esterel [9], but they are different in the fundamental aspect of dealing with events (signals in Esterel). The stacked execution for internal events employed by CÉU greatly improves the expressiveness of the language as shown in Section 4.

Furthermore, Esterel is commonly used in hardware design, and its notion of time is similar to that of digital circuits, where multiple signals can be active at a clock tick. In CÉU, instead of clock ticks, occurrences of external events define time units. We believe that for software design, this approach simplifies the reasoning about concurrency. For instance, the uniqueness of external events is a prerequisite for the static analysis that enables safe shared-memory concurrency in CÉU.

More recently, Wireless Sensor Networks (WSNs) emerged as an active research area for highly constrained embedded concurrency, resulting in the development of many synchronous languages [12, 16, 17].

Protothreads [12] offer lightweight cooperative multithreading for embedded systems. Its stackless implementation reduces memory consumption but precludes support for local variables. CÉU also avoids the use stacks for trails, but preserves support for locals by calculating the required memory at compile time.

SOL [16] and OSM [17] provide parallel state machines for WSNs, offering a formal and mature model for programming embedded systems. However, the main contributions of CÉU, stacked execution for internal events and safe support for shared-memory concurrency, do not directly adapt to the state machines formalism.

In common among the referred works is the agreement in providing low-level access for tasks (e.g., systems calls and shared-memory) and lock-free concurrency that precludes race conditions on programs. However, they do not propose a reliable strategy for concurrent tasks accessing shared resources, as quoted from the references:

- *The protothreads mechanism does not specify any specific method to invoke or schedule a protothread, this is defined by the system using protothreads.* [12]
- *A single write access will always completely execute before the next write access can occur. However, the order in which write accesses are executed is arbitrary.* [17]
- *The parallel operator executes all its threads in a round-robin manner according to the order of their declaration in the program.* [16]

On the opposite side of concurrent designs, asynchronous languages for embedded systems [8, 15] assume time independence among processes and are more appropriate for applications with a low synchronization rate or for those involving algorithmic-intensive problems.

The described techniques for the `do-finally` construct and exception handling heavily rely on `par/or` compositions, which cannot be precisely defined in asynchronous languages without tweaking processes with synchronization mechanisms [7].

Asynchronous models are also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [2, 8, 13]. Even though CÉU ensures bounded execution for reactions, it cannot provide hard real-time warranties. For instance, assigning different priorities for trails would break lock-free concurrency (i.e., breaking correctness is worse than breaking timeliness).

Fortunately, CÉU and RTOSes are not mutually exclusive, and we can foresee a scenario in which multiple CÉU programs run in different RTOS threads and communicate asynchronously via external events, an architecture known as GALS (*globally asynchronous–locally synchronous*) [5].

6. Conclusion

In this work, we presented the design of the reactive programming language CÉU targeting highly constrained embedded systems. We formally described the control aspects of CÉU and discussed how to detect unsafe properties of programs at compile time.

CÉU achieves a high degree of reliability while embracing practical aspects, such as support for lock-free concurrency, low-level access to the platform, and advanced control-flow mechanisms.

We consider a fundamental design choice of CÉU to provide safe shared-memory concurrency for the context of embedded systems, given that low-level I/O is indispensable (e.g. interfacing with sensors and actuators).

Embedded systems are still predominantly developed in the "bare metal", regardless of existing alternatives, probably due to the flexibility and popularity of C. We believe that CÉU is an attractive alternative, given its unrestricted access to C and rich set of concurrent control primitives (e.g., parallel compositions and internal events).

Currently, CÉU is not intended for use in other reactive scenarios, such as desktop applications and games: besides the impracticability of the static analysis for larger applications, CÉU does not support the dynamic creation of trails, which is essential for virtualizing resources (e.g., graphical widgets, AI units, etc.).

References

[1] A. Adya et al. Cooperative task management without manual stack management. In *ATEC'02*, pages 289–302. USENIX Association, 2002.

[2] I. F. Akyildiz et al. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[3] Arduino. Arduino homepage. `http://arduino.cc`.

[4] E. Bainomugisha et al. A survey on reactive programming. *ACM Computing Surveys*, to appear.

[5] A. Benveniste. Some synchronization issues when designing embedded systems from components. In *Proceedings of EMSOFT*, pages 32–49, London, UK, 2001. Springer-Verlag. ISBN 3-540-42673-6.

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

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

[8] S. Bhatti et al. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10: 563–579, August 2005.

[9] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.

[10] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *15th European Symposium on Programming*, pages 294–308, 2006. LNCS 3924.

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

[12] Dunkels et al. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys '06*, pages 29–42. ACM, 2006.

[13] FreeRTOS. Freertos homepage. `http://www.freertos.org`.

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

[15] C. L. Jacobsen et al. Concurrent event-driven programming in occam-pi for the Arduino. In *CPA '11*, volume 68, pages 177–193, June 2011.

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

[17] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of IPSN '05*, pages 45–52, April 2005.

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

[19] M. Olszewski et al. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09*, pages 97–108, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5.

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

[21] F. Sant'Anna and R. Ierusalimschy. Luagravity, a reactive language based on implicit invocation. In *Proceedings of the 2009 Brazilian Symposium on Programming Languages (SBLP)*, pages 89–102, 2009.

[22] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Céu: Embedded, Safe, and Reactive Programming. Technical Report 12/12, PUC-Rio, 2012.