The Semantics of a Safe and Flexible Language for Embedded Systems

Abstract

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

We present a formal description of CÉU 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 advanced mechanisms considering the context of embedded systems, such as finalization 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 [20]. Alternative designs enforce cooperative scheduling to eliminate race conditions, but potentialize unbounded execution, breaking real-time responsiveness in programs [12]. Therefore, language designers have basically three options: not providing threads at all [16], affecting the productivity of programmers; providing restricted alternatives, such as disallowing locals in threads [13]; or preserving full support, but offering coarsed-grained concurrency only [9].

CÉU[1] is a reactive programming language that provides a reliable yet powerful programming environment for embedded systems. CÉU is based on Esterel [10] 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 for programs.

In this work, we focus on a formal description of CÉU that allows us to discuss safety warranties for programs, such as deterministic behavior. For an extensive and informal presentation, with examples of typical patterns found in embedded systems, refer to the technical report of CÉU [23].

CÉU distinguishes itself from Esterel in two basic aspects:

- 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:

- Based on 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 finalization blocks (*finally blocks* in Java), 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 form of subroutines that cannot express mutual runtime dependency, resulting in memory-bounded programs that preclude stack overflows.

The proposed new functionalities are compliant with the safety requirements and resource limitations of embedded systems and, arguably, do not dramatically reduce the expressiveness of the language. However, 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 [22], 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 minimum 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.

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. A declaration includes the type of value the event carries when it occurs. 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).

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

```
// primary primitives          // description
nop(v)                         (constant value)
mem                            (any memory access)
await(e)                       (await event ‘e’)
emit(e)                        (emit event ‘e’)
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
awaiting(e)                    (previously awaiting ‘e’)
stacked(i)                     (stack depth mark)
emitting(i,e)                  (emitting ‘e’)
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; in the example, 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, leading to more concise implementations. 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, while 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 a 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 awakened 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 occurrences 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.

$$
\begin{aligned}
pop(awaiting(e)) &= (-1, \{\}) \\
pop(stacked(i)) &= (i, \{\}) \\
pop(emitting(i, e)) &= (i, \{e\}) \\
pop(p\ ;\ q) &= pop(p) \\
pop(p\ @\ loop\ q) &= pop(p) \\
pop(p\ and/or\ q) &= (max(j, k), F \cup G) \\
&\qquad where\ (j, F) = pop(p) \\
&\qquad and\ (k, G) = pop(q)
\end{aligned}
$$

Other primitives (nop,mem,await,emit,break,if,loop) need not to be defined.

Figure 3. The recursive definition for *pop*.

$$
\begin{aligned}
isBlocked(awaiting(e)) &= true \\
isBlocked(stacked(i)) &= true \\
isBlocked(emitting(i, e)) &= true \\
isBlocked(p\ ;\ q) &= isBlocked(p) \\
isBlocked(p\ @\ loop\ q) &= isBlocked(p) \\
isBlocked(p\ and\ q) &= isBlocked(p) \wedge isBlocked(q) \\
isBlocked(p\ or\ q) &= isBlocked(p) \wedge isBlocked(q) \\
isBlocked(*) &= false\ \ (nop, mem, await, \\
&\qquad emit, break, if, loop)
\end{aligned}
$$

Figure 4. The recursive predicate $isBlocked$.

The semantic rules to be presented generate three statements that the programmer cannot write: the primitives $awaiting$, $stacked$, and $emitting$ avoid the immediate matching of emits and awaits 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 trails awaiting the broadcast external event. Then, we continuously apply small steps until all trails await and/or emit. The next big step awakes all previously awaiting trails matching the 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) remains at stack level i, as it can only execute after the program completely reacts to the event. Awakened trails may emit new events that will increase the stack depth ($i + 2$ and so on); hence, only after the stack unrolls to depth i that the original emitting trail continues.

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

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

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

For the initial big step, function *pop* returns the single external event emitted at starting depth 1. For further big steps, *pop* is defined in Figure 3 and returns the deepest stack level and all emitted internal events (if any) from the previous small-step sequence.

A small-step sequence is represented with the single arrow and is associated with the same depth i from the previous big step, which identifies 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 *pop* returns $(-1, \{\})$.
- The program terminates, i.e., the small-step rules transform the whole program into a nop.

In Section 3.1 we show that, by imposing syntactic restrictions to programs, 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.

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., $(i, E) = (1, \{\$\})$ 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 $awaiting$, $stacked$, and/or $emitting$ 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 the primary primitives:

$$mem \xrightarrow[i]{} nop(v),\ \ (v\ is\ nondet) \qquad \textbf{(mem)}$$

$$await(e) \xrightarrow[i]{} stacked(0)\ ;\ awaiting(e) \qquad \textbf{(await)}$$

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

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

An $await$ is stacked with the lowest possible depth, being transformed in an $awaiting$ only in the end of the reaction chain. This way, a trail that reaches an $await$ can only react to an $emit$ in the next reaction chain.

An $emit$ is deferred to a deeper depth to be applied in the next big step. The rule actually transforms an $emit$ in two primitives in sequence: the $emitting(i + 1, e)$ defers the immediate matching, while $stacked(i)$ holds the trail in the current depth, resuming only after the complete reaction to the event.

Given that small-step rules execute at the current deepest level and that rule **emit** is the only one that increases the stack depth, the next big step will necessarily take all deferred emits. This explains why the definition for *pop* in Figure 3 blindly takes the union of all emitted events without considering their depths.

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

The rules for conditionals and sequences are straightforward:

[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.

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

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

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

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

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

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

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. 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. Then, the whole composition terminates and the blocked side is *killed*, i.e., all its $awaiting$, $stacked$, and $emitting$ primitives are eliminated with the rule transformation.

A similar situation occurs when an and or or reach a $break$ in either of the sides. In this case, the enclosing $loop$ must terminate and both sides are killed, transforming the whole composition into a $break$. As the rules make explicit, a trail can only be killed after it terminates or blocks.

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 by transforming the whole composition becomes into 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 $emitting$ and $awaiting$ 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. 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.

We start with the rules to check if deferred emits match previously awaiting trails:

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

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

The $stacked$ and $emitting$ primitives are "popped" if they are at the deepest level i returned by function pop:

$$stacked\ (i) \underset{(i,E)}{\Longrightarrow} nop \qquad \textbf{(Stack)}$$

$$emitting\ (i, e) \underset{(i,E)}{\Longrightarrow} nop \qquad \textbf{(Emitting)}$$

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

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

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

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

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

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 [10], 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
end                                      await A;
                                     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 await (as desired) or break (reducing the whole expansion to a nop via rule **loop-brk**).

Interleaving big steps and small-step sequences also cannot lead to unbounded execution: all trails that $await$ are stacked at depth 0 before they actually become $awaiting$ (small-step rule **await**). This way, these trails can never awake, given that $emitting$ are always matched in depth levels higher than 0 (small-step rule **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 to C function calls. 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 in a well-marked way.

This approach is also adopted by Esterel, which supports the `call` primitive to execute code assumed to be instantaneous in the host language [8]. In CÉU, we take a step further and statically detect when such calls occur in parallel, as discussed in the next section.

Evidently, the programmer should only recur to C for I/O operations that are assumed to be instantaneous, but never for control activities (e.g. interrupt handling).

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 determines an order of execution among tasks. Even systems with preemptive multithreading can offer guarantees of determistic execution [21].

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 to rules **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 nondeterministic *execution* (scheduling). At compile time, we run a symbolic interpretation of the program that creates a directed 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 an 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 directed 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.

```
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
```

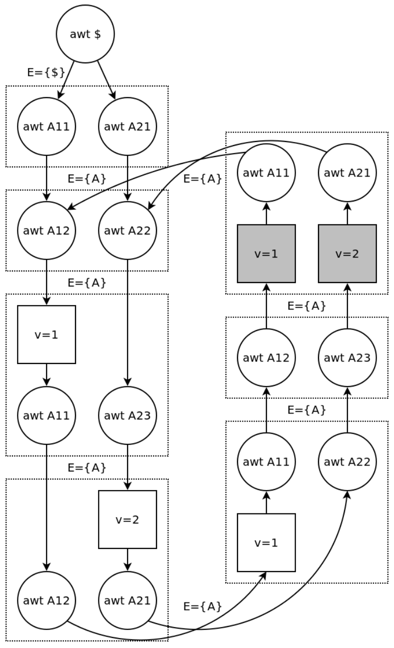

Figure 5. A nondeterministic program in CÉU and its reaction graphs.

- Parallel compositions connect two subgraphs that are interpreted independently until they rejoin. The subgraphs are determinate, given that small-step rules for rejoins require both sides to be terminated or blocked (having nowhere to proceed).
- Blocked trails (awaiting, stacked, or emitting) 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. Then, the process repeats.

A graph is evidently acyclic (the rules above create no cycles) and finite (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 always terminates, given that each 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 main event $\$$, the algorithm computes reaction chains to all external events (the example only uses event A). After six occurrences of event A, the algorithm detects two assignments in parallel paths to the variable v and refuses the program 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.

Our experience shows that the analysis is indeed practical. We have been using CÉU in the context of Wireless Sensor Networks [2] and applications with around 500 lines of code (generating images around 20 Kbytes of ROM) are verified in less than 5 minutes (in a *core-duo 2.2 GHz* laptop).

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 parallel path 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. Regarding C calls, they cannot appear in parallel paths, as CÉU has no knowledge about their side effects. Also, passing variables as parameters counts as read accesses to them, while passing pointers counts as write accesses 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 a C function that does not cause side effects, allowing it to be called concurrently with any other function in the program.
- The `det` modifier (for *deterministic*) declares a pair of variables (e.g., pointers) or functions that do not affect each other, allowing them 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*, i.e., programs in which any two mem operations identified as incompatible (by the prevailing policy) execute only as a causal relation between each other in all possible executions. 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 and terminate. 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. However, 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. 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. In order to support 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 advanced control-flow mechanisms in the language without requiring new primitives. We first describe a restricted form of subroutines that is used as the basis for the more elaborate mechanisms.

Although the described mechanisms involve thoughtful techniques, 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.

4.1 Subroutines

Internal events bring support for a limited form of subroutines. In the example that follows, we define a function `inc` that increments the value passed as reference. A trail in parallel calls this function in reaction to external event `A`:

```
 1: event int* inc;     // function ‘inc’ receives an int
 2: par/or do
 3:     loop do         // function definitions are loops
 4:         int* v = await inc;  // that await the event
 5:         *v = *v + 1;         // to execute the body
 6:     end
 7: with
 8:     int v = 1;
 9:     await A;
10:     emit inc(&v);            // call ‘inc’
11:     _assert(v==2);           // assert after return
12: end
```

A subroutine is represented as a loop that awaits an identifying event (`await inc` in line 4). A subroutine is called in a parallel trail through an emit on the corresponding event (`emit inc` in line 10). The parameter of a subroutine is the type of its corresponding event (`event int* inc` in line 1).

In the example, the trails start awaiting events `inc` and `A`. Once the input event `A` occurs, the second trail awakes and invokes `emit inc(&v)` to "call" subroutine `f`. Given the stacked execution for internal events, the calling trail pauses and the subroutine awakes. The subroutine increments the parameter, loops, and awaits to be called again. Finally, the calling trail resumes and passes the assertion test, terminating the program.

This form of subroutines has some significant limitations:

***Single calling*:** Further calls to a subroutine in a reaction chain have no effect. Suppose that after the assertion test in line 11, the trail tries to call the function again. Given that $await$ statements must be previously awaiting to react to an $emit$, the second call is ignored. Remember that events are unbuffered in CÉU.

***No recursion*:** Recursive calls to a subroutine have no effect. Suppose that after the increment in line 5, the subroutine tries to call itself. As the trail cannot be awaiting itself while running, the recursive call is ignored.

***Single instance*:** Calls to a running subroutine have no effect. Suppose that after the increment in line 5, the subroutine awaits an event (e.g. an external event `B`). While this event does not occur, the subroutine is hanged and cannot serve other requests.

***No concurrency*:** If two trails in parallel try to call the same subroutine passing a parameter, the compiler complains about nondeterminism. An $emit$ that passes a parameter needs to be translated to $mem; emit(e)$ in order to describe parameter passing (where mem accesses e). This way, concurrent emits with parameters are detected as nondeterministic by the static analysis.

CÉU provides no support for standard functions for a number of reasons:

- The interaction with other control primitives is not obvious (e.g., executing an $await$ or a par/or inside a function).
- They would still be restricted in some ways given the embedded context (e.g. no recursion or closures).
- Programs can always recur to C when absolutely necessary.

Regardless of the limitations, this form of subroutines is widely adopted in CÉU programs, given that they were designed to work with the other control mechanisms. Keep in mind that the typical reactive organization of programs (awaiting an external stimulus, reacting to it, and going back to awaiting) does not demand unrestricted subroutines. 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 Finalization blocks

Finalization 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’
    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 6, which forces the execution of the finalization 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`.

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

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

```
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 6. do-finally code and corresponding translation.

3. The first par/and trail emits $fin on termination to invoke the finalization block.
4. The second par/and trail (the finalization block) awaits $fin to start executing.
5. All trails that terminate a par/or or escape a loop emit $fin to also invoke the finalization 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 finalization blocks when a third trail in parallel encloses a do-finally construct and kills it:

- *3rd trail terminates before the do-finally starts to execute.* In this case, 3rd trail emits the corresponding $fin, which is not yet being awaited for, and the finalization block 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 finalization block, 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 finalization block only once, as expected.

do-finally constructs have the restriction that finalization code 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 START;  // callback to start handling the log
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
await START;
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 like subroutines, as discussed in Section 4.1. The operations that perform the actual low-level system calls are placed in parallel and may 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 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 stacked continuations.
   stack: []

This mechanism for exceptions can also support resumption 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 a file operation cannot be called twice within a reaction chain and that exception handlers cannot await other events, which are related to the single-call and 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. Mutual dependency is a known issue in dataflow languages, requiring the explicit placement of a specific delay operator to avoid runtime cycles [11].This solution is somewhat *ad hoc* and splits an internal dependency problem across two reactions to the environment.

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 7 applies the temperature conversion formula between Celsius and Fahrenheit, so that whenever the value in one unit is set, the other is automatically recalculated.

We first define the external events that request changes and also variables to hold the temperatures and corresponding internal events (lines 1-3). 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 internal changes and update the dependency relations among the temperatures (lines 5-8 and 10-13). For instance, the first trail is a `loop` that waits for changes on `tc_evt` (line 6) and signals the conversion formula to `tf_evt` (line 7). The behavior for the second trail that awaits `tf_evt` (lines 10-13) is analogous. The third and fourth trails (lines 15-19 and 21-25) await external updates in loop to notify the changes; The program behaves as follows (with the stack in emphasis):

1. 1st and 2nd trail await `tc_evt` and `tf_evt`;
   stack: []

```
 1:  event int TC, TF;
 2:  int tc, tf;
 3:  event int tc_evt, tf_evt;
 4:  par/or do
 5:     loop do                // 1st trail
 6:        tc = await tc_evt;
 7:        emit tf_evt(9 * tc / 5 + 32);
 8:     end
 9:  with
10:     loop do                // 2nd trail
11:        tf = await tf_evt;
12:        emit tc_evt(5 * (tf-32) / 9);
13:     end
14:  with
15:     loop do
16:        int v = await TC;   // 3rd trail
17:        emit tc_evt(v);
18:        ...   // use ‘tc’ or ‘tf’
19:     end
20:  with
21:     loop do
22:        int v = await TF;   // 4th trail
23:        emit tf_evt(v);
24:        ...   // use ‘tc’ or ‘tf’
25:     end
26:  end
```

Figure 7. A dataflow program with mutual dependency.

2. On TC, 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 *with all dependencies resolved* and awaits the next external change;
   stack: []
8. ... (analogous behavior for further external occurrences)

The example size is disproportionate to the problem it solves, but illustrates the circular dependency issue (similar examples appear in other references [4, 11]). The bottom line is that dataflow techniques permit that complex dependency patterns are handled internally, providing well-defined entry points to application programmers (i.e. they would be required to write only the 3rd and 4th trails in the example).

5. Related work

CÉU is strongly influenced by Esterel [10], 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 in CÉU is a prerequisite for its static analysis that enables safe shared-memory concurrency. However, in Esterel, *if a variable is written by some thread, then it can neither be read nor be written by concurrent threads* [8].

Although CÉU provides some dataflow functionality, it is not intended for data-intensive applications. For instance, the *Functional Reactive Programming (FRP)* paradigm brings dataflow behavior to functional languages and supports the dynamic creation of signals at runtime [24]. We believe that dataflow and imperative reactivity are complementary, but the latter is more suitable for control-intensive embedded systems that deal with low-level I/O and handle explicit state.

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 [13, 18, 19].

Protothreads [13] offer lightweight cooperative multithreading for embedded systems. Its stackless implementation reduces memory consumption but precludes support for local variables. CÉU also avoids the use of stacks for trails, but preserves support for locals by calculating the required memory at compile time.

SOL [18] and OSM [19] 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-machine formalism.

In common among the referred works is the agreement in providing low-level access (e.g., systems calls and shared-memory) and lock-free concurrency that precludes race conditions on programs. However, they do not specify an execution order for tasks reacting to the same external stimulus [8, 13, 19]. Hence, if two tasks access the same resource concurrently, even if the accesses are race free, the final outcome is nondeterministic. As discussed in Section 3.2 CÉU refuses programs with such behavior.

On the opposite side of the spectrum of concurrency models, asynchronous languages for embedded systems [9, 17] assume time independence among processes and are more appropriate for applications with a low synchronization rate or for those involving algorithmic-intensive problems.

Asynchronous models are also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [9, 14, 15]. 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].

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

6. Conclusion

In this work, we presented a formal description of the control aspects of the reactive programming language CÉU and discussed how to detect unsafe properties of programs at compile time.

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

We consider providing safe shared-memory concurrency a fundamental design choice of CÉU, given that low-level I/O is indispensable in the context of embedded systems (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).

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. `http://soft.vub.ac.be/~ebainomu/publications.html`. (To appear in ACM Computing Surveys).

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

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

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

[11] 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.

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

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

[14] M. Farooq and T. Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.

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

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

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

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