Céu: A Reactive Language for Wireless Sensor Networks

ABSTRACT

Céu is a system-level language targeting Wireless Sensor Networks that aims to unify the features of dataflow and imperative synchronous reactive languages, posing an alternative to the predominating event-driven and threaded-based systems.

Céu supports concurrent lines of execution that run in time steps and are allowed to share variables. However, the synchronous and static nature of Céu enable a compile time analysis that ensures deterministic and memory-safe programs. For time consuming operations, Céu provides asynchronous blocks that do not share variables with other parts of a program.

The Céu compiler generates single threaded *C* code, being comparable to handcrafted event-driven programs in terms of size and portability.

1. INTRODUCTION

Wireless sensor networks (WSNs) are composed of a large number of tiny devices (known as "motes") capable of sensing the environment and communicating among them. WSNs are usually employed to continuously monitor physical phenomena in large or unreachable areas, such as wildfire in forests and air temperature in buildings. Each mote features limited processing capabilities, a short-range radio link, and one or more sensors (e.g. light and temperature) [2].

Software developed for WSNs typically performs a succession of sensing, processing, actuating, and communicating. These activities are primarily reactive as they involve permanent interaction with the surrounding environment through timers, messages arrivals, sensor samplings, etc.

The first system languages and operating systems for WSNs follow the event-driven programming model [14, 9], which is efficient for the severe resource constraints of WSNs and, at the same time, quite versatile. However, the limitations of event-driven programming (namely, manual stack management) brought to the scenario multithreaded alternatives that are offered with a cooperative or preemptive scheduling policy (e.g. [10, 5]).

In the context of WSNs, little attention has been given to the family of synchronous languages, which is an established technology for safety-critical embedded systems [3]. We believe that the programming facilities found in these languages match perfectly with the reactive nature of WSNs and could be adopted in this context.

Two major styles of synchronous languages have evolved: in the *control–imperative* style (e.g. [4]), programs are organized as hierarchies of control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style (e.g. [13]), programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming efforts.

In this work, we present Céu, a reactive language targeting WSNs that unifies both control and dataflow synchronous programming styles. Although predominantly imperative, Céu greatly diverges from C-like procedural languages in both syntax and semantics. Céu is based on a small set of control primitives similar in functionality to Esterel's [4]. On top of this kernel, Céu provides disciplined side effects and introduces *internal events*, which together enable dataflow capabilities to the language.

Céu relies on a compile-time analysis to detect unbounded loops and concurrent access to variables. The static analysis precludes any dynamic support in the language, such as memory allocation, recursion, and dynamic loading. However, this trade-off seems to be favorable in the context of WSNs, as dynamic features are discouraged due to resource limitations and safety

requirements.

To evaluate the applicability of CÉU, we consider four key aspects when programming for WSNs: *memory usage*, *responsiveness*, *safety*, and *expressiveness*. In Section 2 we review these aspects considering three common programming models adopted in system-level programming languages: *event-driven programming*, *cooperative multithreading*, and *preemptive multithreading*.

In Section 3 we introduce CÉU: its imperative primitives, synchronous execution model, safety restrictions, dataflow support, first class timers, and asynchronous execution for long computations. In Section 4, we provide a quantitative analysis of CÉU, comparing memory usage, responsiveness, and source code size with existing system-level languages. In Section 5, we discuss the techniques we applied in the implementation of CÉU, what helps to understand the results of our evaluation. Section 6 presents related work, and is followed by the conclusion of our paper, in Section 7.

2. EVALUATION ASPECTS

As of 2011, the state-of-the-art in system-level development for WSNs lies somewhere in between the efficiency of event-driven programming and the expressiveness of cooperative and preemptive multithreading [11]. Each model has its advantages and drawbacks regarding the aspects we consider in this work.

Memory usage is the first aspect in our evaluation. Duffy et al. argue that the multithreaded model tends to use more memory than the event-driven due to the required per-thread stack space [8]. However, some alternatives [10, 18] alleviate the stack problem with tradeoffs, such as disallowing the use of local variables.

The second aspect, *responsiveness*, is the ability of a system to promptly acknowledge high-priority requests (e.g. radio messages). Duffy et al. argue that preemptive multithreading performs better, because schedulers can favor high-priority threads [8], while cooperative multithreading and event-driven programming require the programmer to manually split time consuming tasks, what might be complex in some situations [8].

Safety is also an important aspect, as motes have scarce resources, are deployed in remote locations, and must run for long periods without human intervention. In our discussion, the term safety is focused on ensuring deterministic and bounded execution (i.e. programs should not become unresponsive when executing long loops).

Although event-driven programming and cooperative multithreading are race-free due to the enforced serial code execution, they are susceptible to unbounded execution when programmers do not carefully set yield points in code.

Multithreaded programs with preemptive scheduling are nondeterministic by construction, and are also subject to deadlocks and race conditions given the required manual synchronization.

Hence, these models cannot prevent such unsafe properties, requiring the programmer to perform additional verification.

Expressiveness, the last aspect, means how programs can be written concisely in a language. Event-driven programming is intrinsically unstructured, being the least expressive when compared to multithreaded alternatives [1]. However, cooperative and preemptive mutilthreading require, respectively, explicit scheduling and synchronization, besides all exercise related to the life cycle of threads.

Table 1 summarizes the evaluation aspects with respect to the existing system-level programming models for WSNs.

Table 1: Comparison of system-level programming models for WSNs.

	mem.	resp.	safety	expr.
Event-driven	+	–	–	–
Cooperative M.T.	–	–	–	+
Preemptive M.T.	–	+	–	+

3. THE LANGUAGE CÉU

CÉU is a concurrent language in which multiple lines of execution—known as *trails*—continuously react to input events from the environment. The fundamental concept in CÉU, accounting for its reactive nature, is that of *events*. Waiting for an event halts the running trail until that event occurs. The environment broadcasts an occurring event to all active trails, which share a single global time reference (the event itself).

The following example executes two trails in parallel that show in leds the received values from a radio:

```
   (~Radio_recv ~> v)*      -- 1st trail
||
   (~v ~> Leds_set)*        -- 2nd trail
```

The first trail awaits (~) the external input event *Radio_recv*[1], then triggers (~>) the internal event v, and then loops (*), repeating the process. The second trail awaits the internal event v, then triggers the external output event *Leds_set*, and then loops back. In other words, whenever a radio message is received, the first trail resumes and awakes the second trail, passing the received value through the internal event v.[2]

[1] As a convention, we use uppercase letters to denote external events and lowercase letters to denote internal events.

[2] The example could be rewritten simply as

The use of trails in Céu allows the programmer to handle multiple events simultaneously (e.g. timers, radio messages arrivals, etc.). Furthermore, a trail awaits an event without loosing context information, such as locals and the program counter.

Figure 1 shows the core syntax for the Céu expressions.[3]

Figure 1: Syntax for Céu.

`e ::=`	`e ; e`	(*sequence*)
	`e ? e : e`	(*conditional*)
	`e || e`	(*parallel or*)
	`e && e`	(*parallel and*)
	`e*`	(*loop*)
	`e^`	(*loop break*)
	`{ e }`	(*scope block*)
	`@{ e }`	(*asynchronous block*)
	`ID`	(*variable read*)
	`e => ID`	(*variable attribution*)
	`~ID`	(*event await*)
	`~TIME`	(*time await*)
	`~> ID`	(*event trigger*, 0 *params*)
	`e ~> ID`	(*event trigger*, 1 *param*)
	`e -> ID`	(*operation call*, 1 *param*)
	`(e,e) -> ID`	(*operation call*, *N params*)

Most expressions are self-explanatory. Note the syntax for attributions, triggers, and calls, in which the source expressions come first (resembling a dataflow style). Asynchronous blocks are explained in Section 3.5.

Operations are side effect free, relying only on their call-by-value parameters. The expression `(a,b)->add` invokes the operation *add* over the current values of the variables *a* and *b*. Operations are defined in a host language following a standard interface (e.g. functions in *C*). Note that Céu has no syntactic support for arithmetic or logical operators, which must be defined as external operations.

A parallel expression executes its subexpressions in concurrent trails, terminating when one of them (*par/or*), or both (*par/and*) terminate. Only parallel expressions create new trails in Céu, and all bookkeeping of trails (e.g. space allocation and scheduling) is done by the language, promoting a fine-grained use of trails. For instance, when any subexpression in a *par/or* terminates, Céu automatically destroys all other sibling trails.

Céu is grounded on a precise definition of time as a discrete sequence of external input events: a sequence because only a single input event is handled at a time; discrete because a complete reaction always executes in bounded time (discussed in Section 3.1). The execution model for a Céu program is as follows:

1. The program initiates in a single trail.
2. Active trails execute until they await or terminate. This step is named a *reaction chain*, and always runs in bounded time.
3. If the program does not terminate, then it goes idle and the environment takes control.
4. On the occurrence of a new external input event, the environment awakes the program on its awaiting trails. It then goes to step 2.

If a new external input event happens while a reaction chain (step 2) is running, the environment enqueues it, as reaction chains must run to completion. When multiple trails are active at a time, Céu does not specify the order in which they should execute. The language runtime is allowed to serialize, interleave, or even parallelize their execution.

A reaction chain may involve triggers and reactions to multiple internal events (discussed in Section 3.3), but only a single external input event is handled at this step.

External events are platform dependent and are used to model I/O with the surrounding environment. An external event is either of type input or output, never being of both types.

3.1 Bounded execution

A reaction chain must run in bounded time to ensure that a program is responsive and can handle upcoming external events. In Céu, only *operators* and *loops* might cause a reaction chain to run in unbounded time.

As operators are typically simple functions that provide ordinary operations, Céu assumes that their implementation in the host language does not enter in loop. This responsibility is left to the programmer, and can be easily met by avoiding the use of loops and recursive calls in the host language.

To guarantee that Céu loops run in bounded time, we demand that each possible path in a loop body contains at least one *await* or *break* expression. Based on this restriction, the following loops are refused at compile time: `(1)*`, `(~A||v)*`, `(v?1:~A)*`; while the following are accepted: `(~A)*`, `(~A&&v)*`, `(~A?1:0)*`. By structural induction, it is trivial to infer whether a given loop body satisfies that restriction or not.

3.2 Determinism

Determinism is usually a desired safety property, making concurrent programs more predictable and easier to debug. Concurrency in Céu is characterized when two or more trails execute during the same reaction chain.

`(~Radio_recv ~> Leds_set)*`, but would not illustrate the concurrent facet of Céu.

[3]We omitted the part of the language that borrows from *C* type declarations, pointers, arrays, and constants.

For instance, in the *par/and* expression `(1=>a&&2=>a)` both assignments run concurrently, while in `(~A;1=>a && ~B;2=>a)` there is no possible concurrency between the assignments, as A and B are external events and cannot happen at the same time (by CÉU's definition of time).

There are four possible sources of nondeterminism in CÉU:

Concurrent access to variables: when the same variable is accessed (or triggered) in concurrent trails, e.g. `(1=>a && 2~>a)`;

Concurrent trigger on external events: when external events are triggered in concurrent trails, e.g. `(1~>A && 2~>B)` or `(1~>A && 2~>A)`;

Concurrent par/or *termination:* when two trails in a *par/or* terminate concurrently, e.g. `(1||2)=>a`;

Concurrent loop escape: when two trails escape a loop concurrently, e.g. `(1^ && 2^)*=>a`.

During compile time, CÉU performs a *temporal analysis* in programs that convert them into deterministic finite automata in order to detect the forms of nondeterminism. A DFA covers exactly all possible paths a program can reach during runtime. The following program is identified as nondeterministic, because the variable v is accessed concurrently on the 6th occurrence of the event A:

```
   (~A; ~A; 1=>v)*
&&
   (~A; ~A; ~A; v)*
```

Note that output external events are usually unrelated, and the exact order they execute may be irrelevant. As an example, in the program `(~>Leds_led0On && ~>Leds_led1On)`, although the triggers run concurrently, the order in which each led is turned on cannot be perceived in practice. Nonetheless, CÉU is strict about determinism and refuses this program by default. However, it is possible to specify sets of related events that cannot occur concurrently:

```
nondet Leds_led0On, Leds_led0Off,
       Leds_led0Toggle, Leds_set;
nondet Leds_led1On, Leds_led1Off,
       Leds_led1Toggle, Leds_set;
nondet Leds_led2On, Leds_led2Off,
       Leds_led2Toggle, Leds_set;
```

An event that is listed in `nondet` sets can be triggered concurrently with any other event not sharing a set with it. Hence, the previous example becomes deterministic when considering the sets defined above, i.e., both `Leds_led0On` and `Leds_led1On` are members of a set, but do not occur together in any set.

3.3 Dataflow support

In CÉU, every variable is also an internal event and vice-versa, hence, a trigger on an internal event with a value also assigns that value to it. For this reason, internal events are also known as *reactive variables*.

Internal events bring dataflow support to CÉU. The following program fragment specifies that whenever variable $v1$ changes, $v2$ is automatically updated to $v1 + 1$ (1st trail), which in turn, automatically updates $v3$ to $v2 + 1$ (2nd trail):

```
   (~v1 -> inc ~> v2)*     -- 1st trail
||
   (~v2 -> inc ~> v3)*     -- 2nd trail
```

In contrast with external events, which are handled in a queue, internal events follow a stack policy and react within the same reaction chain. In practical terms, this means that a trail that triggers an internal event halts until all trails awaiting that event completely react to it, continuing to execute afterwards (but still within the same time unit).

In the example, suppose $v1$ is updated twice in sequence with the code `(... ; 10~>v1 ; 15~>v1)` in a 3rd trail in parallel. The program behaves as follows (with the stack for internal events in emphasis):

1. 3rd trail triggers `10~>v1` and halts;
   stack: [3rd]
2. 1st trail awakes, triggers `11~>v2`, and halts;
   stack: [3rd,1st]
3. 2nd trail awakes, triggers `12~>v3`, and halts;
   stack: [3rd,1st,2nd]
4. no trails are awaiting $v3$, so 2nd trail resumes, loops, and awaits $v2$ again;
   stack: [3rd,1st]
5. 1st trail resumes, loops, and awaits $v1$ again;
   stack: [3rd]
6. 3rd trail resumes, triggers `15~>v1`, and halts;
   stack: [3rd]
7. ...

Note that by the time the second trigger `15~>v1` executes (step 6), the trails in parallel are already awaiting $v1$ and $v2$ again (steps 4,5), hence, they will react again during the same reaction chain (step 7 on). This behavior, which we consider to be the expected for nested triggers, is naturally achieved with the stack execution policy.

An intriguing issue in dataflow languages is when programs have to deal with mutual dependency among variables. Such specifications lead to dependency cycles in programs, which require the explicit placement of *delay* combinators to break cycles [7, 6].

As an example, suppose variables a and b must hold the constraint that a is always $b + 1$, so that whenever a changes, b must be recalculated, and vice-versa. The following code fragment implements this constraint:

```
   (~a -> dec ~> b)*    -- 1st trail
||
   (~b -> inc ~> a)*    -- 2nd trail
```

This code can be inserted in parallel with a larger program to apply the constraint. If, for example, the event a triggers, the first trail resumes, triggers b, and halts (before awaiting a again). Then, the second trail resumes and triggers a, what has no effect on the first trail. In the end, the trails await a and b again. Hence, due to the stack execution policy for internal events, the code fragment above does not contain dependency cycles, is deterministic and compiles fine in CÉU.

3.4 Physical Time

Physical time[4] is probably the most common input in WSN applications, as found in typical patterns like sensor sampling and watchdogs. However, language support for physical time is somewhat low-level, usually through timer callbacks or sleep blocking calls.

Furthermore, underlying systems cannot ensure that a timer expires precisely with zero-delay on the requested timeout. For a single iteration this *residual delta time* is insignificant, but the systematic use of timers might accumulate a considerable amount of *deltas* that could become perceptible.

In CÉU, physical time is treated as special kind of external input event: the expression `~1s500ms` awaits one second and a half.

CÉU handles *deltas* automatically, leading to more robust applications. As an example, in the expression `(~10ms)*`, after 1 second elapses, the loop iterated exactly 100 times, even if a given reaction chain during that period takes longer than $20ms$. The runtime of CÉU keeps the *delta* for the current reaction chain and decrements it from the following await.

Also, CÉU takes into account the fact that time is a physical quantity that can be added and compared. For instance, in the expression `(~50ms;~49ms || ~100ms)`, if CÉU cannot guarantee that the left *par/or* subexpression terminates exactly in 99ms, it can at least ensure that it will terminate before the second subexpression.

Finally, the temporal analysis of CÉU (introduced in Section 3.2) also embraces the semantics for time. For instance, the expression `(~50ms;~49ms;1=>a || ~100ms;2=>a)` is deterministic, while `((~10ms;1=>a)* || ~100ms;2=>a)` is not.

[4] By physical time we mean the passage of time from the real world, measured in hours, minutes, milliseconds, etc.

3.5 Asynchronous Blocks

One of the main limitations of the synchronous execution model is its inability to perform long computations requiring unbounded loops. *Asynchronous blocks (asyncs)* fill this gap in CÉU. Expressions enclosed by `@{` and `}` can contain unbounded loops, and run asynchronously with the rest of the program (referred to as the *synchronous side*). All variables in an *async* are local to it. The following example shows the sum of the arithmetic progression from 1 to 100 in leds:

```
(
  @{
    0 => sum ;
    1 => i ;
    (
      (sum,i)->add => sum ;
      (i,100)->eq ? sum^ : i->inc=>i ;
    )*
  }
||
  ~10ms ; 0 ;
) ~> Leds_set ;
```

In the example, the sum loop must be inside an *async* as it contains no await expressions. We use a watchdog in parallel that sets leds to 0 and cancels the computation if it takes longer than $10ms$.

CÉU specifies that *asyncs* only execute when there are no pending input events in the synchronous side. Hence, it gives no warranty that an *async* will ever terminate.

From the synchronous perspective, an *async* is equivalent to a trigger followed by an await on new unique external events, i.e.,

`~>XXX_ini ; ~XXX_end`,

where the trigger expression (`~>XXX_ini`) requests the computation to start, which runs completely detached from the synchronous code; while the corresponding await expression (`~XXX_end`) awakes when the computation terminates, yielding its final result.

This equivalence emphasizes that *asyncs* have a clear and localized impact on the synchronous side of a program.

3.6 Simulation in Céu

Simulation is an important aspect in cross-compiling platforms, such as WSNs development. It is usually employed to test applications before deploying them on target platforms. However, simulators are usually inaccurate, may require additional knowledge to operate, and vary among different developing platforms.

CÉU provides ways to simulate programs in the language itself, not depending on any external tool to test its programs. *Asyncs* are allowed to trigger external

input events and the passage of time towards the synchronous side of a program. Input events from *asyncs* go through the same queue for "real" events—once in the input queue, there is no distinction among them.

This way, it is easy to simulate and test the execution of programs with total control and accuracy with regard to the order of input events—all is done with the same language and inside the programs themselves.

In the following example, the first trail awaits the input event *Start* and then increments v every 10 milliseconds. To test this code, we simulate, in a second trail, the occurrence of the event *Start* and then the passage of $1s35ms$.

```
  ~Start=>v ;                      -- 1st trail
  (~10ms ; v->inc=>v)*
||
  @{ 10~>Start ; ~>1s35ms } ;   -- 2nd trail
  (v,113)->eq->assert ;
```

The program starts with both trails in parallel. However, the second trail enters an *async*, allowing the first trail to progress and await the event *Start*. No more pending synchronous code remains, so the *async* progresses and triggers `10~>Start`, what makes the first trail to resume and await $10ms$. Then, the *async* resumes and generates the passage of $1s35ms$. Only after the first trail completely reacts to it (the loop iterates exactly 103 times), the *async* terminates, proceeding to the assertion test that terminates the program successfully.

From the example it should be clear that simulation does not test true I/O, only the program behavior given an arbitrary input sequence. Also, simulation can be employed—with the exact same behavior—in the developing platform (given Céu is available), or in the target platform.

Note that in a reactive language a program execution depends solely on the input events it receives from the environment. Also, in a deterministic language, the exact timings for the incoming events are irrelevant to the application outcome, only the order they arrive.

3.7 Examples

The following examples show some program fragments in Céu that explore common patterns found in WSN applications. These fragments are extracted from real applications we have ported to evaluate Céu, as discussed in the next section.

The first example executes three trails in parallel, and each one blinks a led with a different frequency forever:

```
(
  ( ~250ms  ; ~>Leds_led0Toggle )*
||
  ( ~500ms  ; ~>Leds_led1Toggle )*
||
  ( ~1000ms ; ~>Leds_led2Toggle )*
)
```

The next example handles the process of starting a radio for communication. The output event `Radio_start` (line 5) requests the radio initialization, while the input event `Radio_startDone` (line 8) awaits the completion status. The code deals with error codes and timeouts:

```
 1:  (
 3:     ~1s
 4:  ||
 5:     ( ~>Radio_start => radio_err ?
 6:        ~> radio_err
 7:     :
 8:        ( ~Radio_startDone => radio_err ?
 9:           ~> radio_err
10:        :
11:           0^
12:        )
13:     ) ; ~1s
14:  )*
```

The whole process is the outer loop that runs until no errors are returned. The $1s$ watchdog (line 3) restarts the process if no feedback from the environment is received. The expression `~>Radio_start=>radio_err` (line 5) triggers the initialization event and saves its immediate return status in the variable `radio_err`. The await expression (line 8) awakes on the initialization completion. Every error (a nonzero value) is signalized to the rest of the application through the `radio_err` reactive variable (lines 6, 9). The break expression (line 11) is reached only if no errors occur. Otherwise the program awaits $1s$ (line 13) before retrying.

We can use the previous fragment as a kind of library to be used in programs. The following example blinks a led on every occurrence of `radio_err`. Once the radio successfully starts, the *par/or* terminates, proceeding to the code in sequence, which will actually use the radio for communications.

```
(
  RADIO_START() -- copy the previous fragment
||
  (~radio_err ; ~>Leds_led0Toggle)*
) ;
...             -- proceed to use the radio
```

4. EVALUATION OF CÉU

In order to evaluate three of the aspects we considered in our work—*memory usage*, *responsiveness*, and *expressiveness*—we have performed some experiments with real applications and present a quantitative analysis of Céu in this section.

Unfortunately, it is not trivial to evaluate the *safety* aspect in a quantitative manner. Nonetheless, all programs written in Céu for the following experiments are 100% free of unbounded execution and nondeterministic behavior.

In the first experiment, we ported existing *nesC* [12] applications to Céu. Our goal is to compare *nesC* and Céu with respect to *memory usage*, and lines of code (as an indicative of *expressiveness*). By using existing applications in our experiment, we intend not to choose specific scenarios that favor one language or the other.

We chose *TinyOS*/*nesC*[5] in this comparison for a number of factors:

- *nesC* is the *de-facto* standard programming language for WSNs.
- *nesC* is stable and freely available.
- The Céu compiler generates code for TinyOS, meaning that examples in *nesC* will also run in any platform in which Céu works.
- The *nesC* distribution provides a wide range of sample applications that explore the domain of WSNs.
- Related works (e.g. [10, 16, 17]) already include comparisons to *nesC*, allowing (at least) an indirect comparison of Céu with them.

Table 2 shows the amount of ROM, RAM, and LOCs (lines of code) for the same applications written in *nesC* and Céu. The third line for each application shows the ratio $^{\text{Céu}}/_{nesC}$ for a given measure, for example: the AntiTheft written in Céu uses 1.40 times more RAM than its *nesC* counterpart.

Our experiment suggests that as application complexity grows, the difference in memory consumption decreases, reaching around 30-35% for the BaseStation application. This behavior is a consequence of the memory footprint of Céu, which requires specialized code for the runtime bookkeeping of timers, trails, events, etc.

When evaluating LOCs of programs, we considered only their core implementation file (*modules* in *nesC*), and extracted from it all comments, interface declarations, and extra spaces.[6] With this approach we focus on the logic of programs, where programmers spend most of their time and rely on the expressiveness of the language in use. The Céu numbers are quite satisfactory, being around 50% smaller for all applications.

[5] We used $TinyOS-2.1.1$ and *micaz* motes, and the sample applications found in `/opt/tinyos-2.1.1/apps/`.

[6] The original and modified sources for the experiment can be found at ??? (hidden in this version due to the double blind submission policy).

Table 2: Céu vs TinyOS: sample applications

		ROM	RAM	LOC
Blink	*nesC*	2052 bytes	51 bytes	17 lines
	Céu	4168 bytes	247 bytes	5 lines
	$^{\text{Céu}}/_{nesC}$	**2.03**	**4.84**	**0.29**
Sense	*nesC*	4370 bytes	84 bytes	24 lines
	Céu	6742 bytes	348 bytes	11 lines
	$^{\text{Céu}}/_{nesC}$	**1.54**	**4.14**	**0.46**
AntiTheft	*nesC*	22424 bytes	1663 bytes	85 lines
	Céu	27014 bytes	2325 bytes	45 lines
	$^{\text{Céu}}/_{nesC}$	**1.20**	**1.40**	**0.53**
BaseStation	*nesC*	15216 bytes	1735 bytes	144 lines
	Céu	19844 bytes	2373 bytes	57 lines
	$^{\text{Céu}}/_{nesC}$	**1.30**	**1.37**	**0.40**

In the second experiment we evaluate the *responsiveness* aspect. The general idea of the experiment is to check if motes can promptly answer radio requests when subjected to long computations. We chose to compare Céu with MantisOS [5], given that multithreaded systems perform better in this aspect [8]. Table 3 summarizes the results of this experiment, which is described next.

Table 3: Céu vs MantisOS: responsiveness

		no comp.	sort,cypher	inf. loops
1 sender	MantisOS	23.2s	23.3s	23.3s
	Céu	23.3s	23.3s	23.3s
2 senders	MantisOS	19.8ms	19.8s	19.9s
	Céu	12.3ms	12.8s	12.4s

(the measures are the average of three consecutive executions)

We first created two simple applications to send and receive radio messages to measure how fast they exchange 3000 messages without losses. We varied the sending speed, and the fastest the receiving side could sustain without losses was around $7ms$ for each message (coincidently, in both implementations), resulting in $23s$ for the full process (in Table 3, see *"1 sender/ no comp."*).

In order to evaluate the responsiveness of the receiving side, we changed it to also execute in parallel two distinct long computations that run forever. In both implementations, the 3000 messages were received without losses, while the increase in the total receiving time was negligible (see *"1 sender/sort,cypher"*).

For the long computations, we implemented an in-place heap sort and *XTEA* cryptography[7] to represent

[7] `http://en.wikipedia.org/wiki/XTEA`.

two different algorithms.

In MantisOS, we had to change the priority of the receiving thread to be higher than the others. In CÉU the receiving part (which is synchronous) already runs with higher priority than long computations (which run inside *asyncs*), and this cannot be changed.

As a more extreme test, we included five computations with infinite loops (e.g. `while (1);`) to run in parallel. Again, both implementations took around 23*s* to receive all 3000 messages without losses (see *"1 sender/inf.loop"*).

In another test, we kept the single receiver and used two senders to measure how fast the receiving side receives 3000 messages (now ignoring the losses) while running long computations in parallel.

Although CÉU (through TinyOS) performs better than MantisOS (see *"2-senders/no comp."*), our objective is to measure the *increase* in the total time due to the long computations running in parallel. For MantisOS the increase among the three variations is negligible, while in CÉU it is below 5%.

From the second experiment, we conclude that CÉU is comparable to a multithreaded implementation in terms of responsiveness, both having nearly optimal behavior for the tests we performed. Although not in the scope of this work, we asserted that, for all tests, both implementations performed a fair scheduling among long computations.

5. THE IMPLEMENTATION OF CÉU

As a static language, much of the complexity of the implementation of CÉU resides in the compile phase. Nonetheless, some complexity is left to the runtime phase, which has to handle multiple queues for upcoming input events, active trails, internal events, *asyncs*, and timers.

We use the following program as our guiding example for this section:

```
(
   (~A,~B)->add^    -- 1st trail
||
   (~B && ~C)       -- 2nd trail
)* ;
...                 -- code after the loop
```

5.1 Compile phase

The compile phase can be subdivided into *parsing*, *temporal analysis*, *memory layout*, *gate allocation*, and *code generation*.

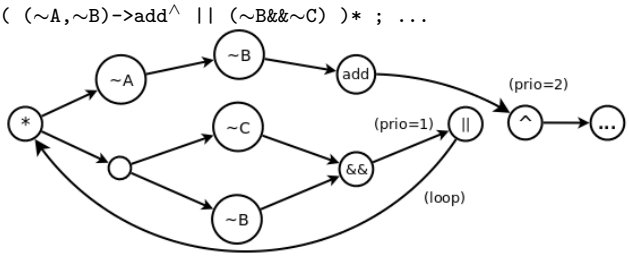
The CÉU parser is written in *LPeg* [15], and converts a program into an *abstract syntax tree (AST)* to be used in the following phases.

5.1.1 Temporal analysis

The *temporal analysis* detects inconsistencies in CÉU programs, such as tight loops and the forms of nondeterminism, as discussed in Sections 3.1 and 3.2. It is also responsible for setting the priorities for trails (see further) and determining the sizes of the queues that are used during runtime.

The program AST is first converted into a graph that represents its execution flow. Figure 2 shows the corresponding graph for our example.

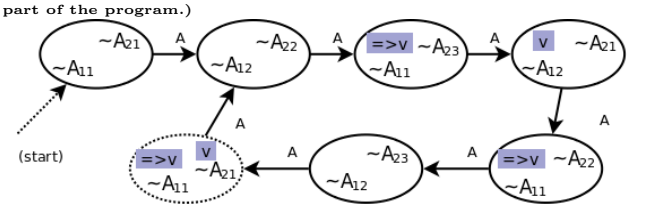
Figure 2: Flow graph for the program `( (~A,~B)->add^ || (~B&&~C) )* ; ...`



By default all nodes in a flow graph have priority 0 (highest). However, as the figure shows, nodes that represent the termination of *par/ors* and loops have lower priorities (the outer, the lower). The priority scheme is needed to avoid glitches during runtime, and is equivalent to traversing a dependency graph in topological order, as employed in functional reactive programming implementations [6].

The flow graph is then converted to a DFA, as introduced in Section 3.2. Currently, the DFA conversion mimics exactly the execution of a program for every applicable input sequence. Figure 3 shows the resulting DFA for the nondeterministic example in Section 3.2.

Figure 3: DFA for the nondeterministic program $(\sim A_{11};\ \sim A_{12};\ 1{=}{>}v)*\ ||\ (\sim A_{21};\ \sim A_{22};\ \sim A_{23};\ v)*$ **(The subscripts help identifying each ~A, but are not part of the program.)**



Each state contains what to execute and what to await. In the figure, the first state represents both trails awaiting the event A. After six occurrences of A, the variable v is accessed concurrently (in the dotted state), qualifying a nondeterministic behavior in the program, which is refused at compile time.

The algorithm is exponential, but most flow nodes can be ignored, reducing considerably the size of graphs. In an average laptop[8], it takes no longer than a few mil-

[8]Laptop with a Intel Core-Duo processor.

liseconds to convert any program used in our evaluation.

5.1.2 Memory layout

CÉU favors a fine-grained use of trails, being common to find trails that await a single event. For this reason, CÉU does not allocate stacks for trails; instead, all data resides in fixed memory slots—this is true for the program variables as well as for temporary values and flags needed during runtime. For instance, the first trail in the guiding example requires temporary slots to hold the parameters to `add`, while the second trail must keep flags to remember which sides of the *par/and* have already terminated.

The memory for trails in parallel must coexist, while expressions in sequence can reuse it. In the example, the code following the loop (identified as ...) reuses all memory from the loop.

CÉU statically allocates a one dimension vector to hold all memory slots, whose size is the maximum the program uses at a given time. A given position in the vector may hold different data (with variable sizes) during runtime.

5.1.3 Gate allocation

Each await expression has an associated *gate* that holds whether the expression is currently active (awaiting) or not. Gates for the same event are grouped in a list that is traversed whenever the event occurs, awaking the expressions whose gates are active. In contrast with memory slots, gates are not reused in different parts of the program.

In the example, there is one gate for each of the four await expressions. When the event B occurs, its list of two gates is traversed to awake the currently awaiting trails.

All gates are set to inactive when a program starts. Once an await expression is reached, its corresponding gate is turned on. Once an await expression awakes, its corresponding gate is turned off.

In CÉU, there is a strict relation between gates and trails. A trail can be seen as a sequence of atomic operations with await expressions separating them. If a trail is active in between two reaction chains, it must be awaiting a single event, and, therefore, only one of its gates can be active at a time. Hence, a trail can be destroyed by simply setting all of its gates to inactive. This is exactly what CÉU does to sibling trails when a *par/or* or *loop* terminates.

5.1.4 Code generation

The final output of the compiler is *nesC* code, as CÉU uses TinyOS as its underlying platform. For instance, a trigger on an external output event is converted to a *command call* in *nesC* (i.e., `7~>Leds_set` becomes `call Leds.set(7)`). However, except for the initialization code and I/O, the generated code is essentially C.

For most CÉU expressions, the conversion to C is straightforward: operators are defined as inline functions and invoked through normal C calls; variable access manipulates the memory vector; and conditional, sequencing, and loops also pose no challenges.

The biggest semantic mismatch between C and CÉU resides in await and parallel expressions. Considering the expression `(~A,~B)->add` from the example, it is clear that before invoking the `add` operator, the program must yield control to the environment twice to await the input events A and B. Hence, the generated code must be split in three parts: before awaiting A, before awaiting B, and invoking `add`. Follows the pseudo-code generated for that expression:

```
Bef_A:
  GTES[A1] = Bef_B;       -- activate gate A1
  halt;                   -- await A
Bef_B:
  GTES[A1] = 0;           -- deactivate gate A1
  DATA[P1] = DATA[A];     -- save 1st param
  GTES[B1] = Add;         -- activate gate B1
  halt;                   -- await B
Add:
  GTES[B1] = 0;           -- deactivate gate B1
  add(DATA[P1],DATA[B]); -- invoke add
  halt;
```

The labels `Bef_A`, `Bef_B`, and `Add` represent entry points held in gates that the CÉU runtime execute according to the current input event and the state of gates. Note that the first parameter to `add` must be saved in a temporary slot, because the event A may occur again before B.

As Section 5.2 shows, the real implementation uses C switch case labels enclosed by a loop to traverse a queue holding pending entry points. With this technique, a parallel expression simply inserts into this queue entry points to its subexpressions:

```
Par_1:
   enqueue Sub_1;
   enqueue Sub_2;
   halt;
Sub_1:
   ...
Sub_2:
   ...
```

5.2 Run-time phase

As a reactive language, the execution of a CÉU program is guided by the occurrence of external events. Hence, the CÉU runtime is basically an infinite loop that awaits events from the environment.

Céu uses three main queues to conduct the execution of a reaction chain. The first queue, `Q_EXTS`, is responsible for serializing external events into the program. The second queue, `Q_TRAILS`, holds entry points triggered from active gates for the current event. The third queue, `Q_INTRA`, holds internal events triggered in the current reaction chain, whose execution is delayed until `Q_TRAILS` is emptied.

The whole process is illustrated in Figure 4 and detailed in the following pseudo-code:

```
 1:  while (1)
 2:  {
 3:     ext = await(Q_EXTS);
 4:     trigger(ext);
 5:
 6:  _TRAILS_:
 7:     while (label = remove(Q_TRAILS))
 8:     {
 9:        switch (label) {
10:           case Init:     // init label
11:              ...
12:              break;
13:           case Awt_exp1: // gate label
14:              ...
15:              break;
16:           case Awt_expN:
17:              ...
18:              break;
19:           ...
20:     }
21:
22:     if (isEmpty(Q_INTRA))
23:        continue;
24:
26:     while (intl = remove(Q_INTRA))
27:        trigger(intl);
28:     goto _TRAILS_;
29:  }
```

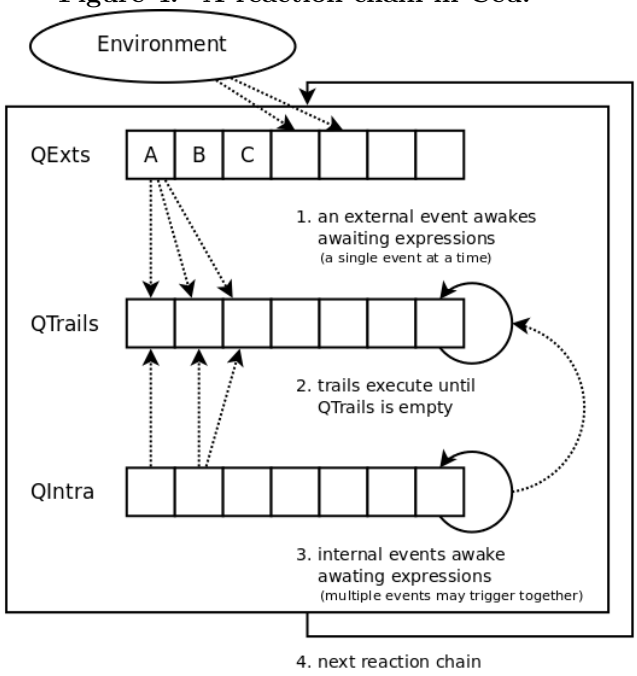
The infinite loop (line 1) takes an external event at a time (line 3). The command `trigger` (line 4) traverses the list of gates for the external event, inserting the entry points for active gates into `Q_TRAILS`.

In line 7, another loop continuously switches to entry points identified as case labels, executing their corresponding code, until `Q_TRAILS` is empty. The execution may unroll new insertions into `Q_TRAILS` (e.g. parallel expressions).

Once the trails queue is empty, Céu checks `Q_INTRA` for triggered internal events (line 22). If the queue is empty, the current reaction chain terminates and the next external event is handled (line 23). Otherwise, the corresponding switch labels for the awaiting trails are inserted into `Q_TRAILS` (line 27), which is traversed again (line 28).

The `Init` label (line 9) points the beginning of the program and executes before any external event.

Figure 4: A reaction chain in Céu.



The implementation of *asyncs* pose some challenges, as they execute in unbounded time and must be preempted by incoming events and also other *asyncs*.

Currently, before every loop iteration in an *async*, Céu checks if there are pending events in `Q_EXTS`. If it is the case, the single *async*'s gate is set to the next loop iteration and the execution halts.

In order to switch control among multiple active *asyncs*, Céu generates pseudo external events every $10ms$ that change control to the next *async* (in `Q_ASYNCS`)[9]. As Céu does not use stacks for *asyncs*, no additional efforts for context switches is required.

Céu also maintains an auxiliary queue, `Q_TIMERS`, for physical time awaits, and keeps a system timer for the earliest await to expire. When the timer expires, Céu awakes the awaiting expression(s) and creates a new timer for the next await in the queue.

Except for external events (which cannot be predicted), the required size for all queues is precisely calculated at compile time. For every state in the program DFA, Céu counts the number of parallel expressions, internal triggers, awaits, and *asyncs* to properly set the queue sizes. Although the resulting sizes are currently slightly overestimated, it is not possible that these queues overflow.

[9]The value of $10ms$ was copied from the MantisOS implementation.

The size for the external queue is arbitrarily set to 20, and overflows have the effect that new events are discarded.

6. RELATED WORK

Our work is strongly influenced by the Esterel language [4]. In Esterel, time is defined as discrete reaction steps in which multiple external signals (events in CÉU) can be queried on their presence status. This semantics has more proximity with that of electronic circuits, but simultaneous events would inhibit the temporal analysis of CÉU. For instance, variable manipulation in Esterel is restricted to a single process (trail in CÉU).

Karpinski and Cahill present a language (also based on Esterel) targeting WSNs, and perform a throughout quantitative and qualitative comparison with *nesC*[16]. CÉU differs from these languages with the introduction of internal events and deterministic support for concurrent access to variables, which are fundamental for the dataflow capabilities of CÉU.

The *Functional Reactive Programming (FRP)* paradigm brings dataflow behavior to functional languages [19]. CÉU borrows some ideas from a FRP implementation [6], such as push-driven evaluation and glitch prevention. However, the dynamic nature of FRP does not match the efficiency and safety requirements of WSNs, what motivated the development of Flask, a restricted FRP implementation [17]. With a powerful set of stream combinators, Flask is more expressive than CÉU for data oriented applications. However, FRP may not be the best programming abstraction for control intensive applications.

Protothreads [10] offer very lightweight threads with blocking support. Its stackless implementation reduces memory consumption but prevents automatic variables. Furthermore, Protothreads provide no safety support besides atomic execution of threads: a program can loop indefinitely, and access to globals is unrestricted.

7. CONCLUSION

We presented CÉU, a language targeting WSNs that unifies imperative and dataflow reactive programming. CÉU is based on a small synchronous kernel that provides reaction to events and imperative primitives (e.g. awaits, parallel blocks, and loops). In order to support dataflow, CÉU provides *internal events* as a communication mechanism among trails. The stack execution policy for internal events better expresses nesting of triggers and also avoids cycles for mutual dependency among data.

In this work we evaluated key aspects in WSN development. We showed that CÉU is comparable to current system-level languages for WSNs with respect to memory usage and responsiveness. Furthermore, we believe that CÉU poses concrete advantages in terms of safety and expressiveness when compared to these languages.

In the design of CÉU we favored safety over power, since we restricted the language to static capabilities only. However, this limitation can be considered (to some extent) advantageous for WSNs, given that CÉU enforces the prevailing discipline in this context. We propose a temporal analysis in programs that prevents unresponsiveness and enforces deterministic behavior.

In terms of expressiveness, our initial experiments show a 50% decrease in LOCs when comparing CÉU to *nesC*. Besides supporting imperative and declarative reactive programming, CÉU provides native support for *physical time*, *asynchronous blocks* for long computations, and simulation from within the programs themselves.

We are aware of the limitations of evaluating the expressiveness of CÉU based solely on lines of code. On the way to a more in-depth qualitative approach, we are currently teaching CÉU as an alternative to *nesC* in a hands-on WSN course in a high-school. The students successfully implemented a simple multi-hop communication protocol in CÉU. Also, the same format is being employed in an undergraduate course, but still in an early stage. We will compare the achievements of the students with both languages and use the results in our evaluation.

At this point, we did not evaluate battery consumption, which we also consider a key aspect in WSNs and plan to include in our evaluation.

8. REFERENCES

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

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91:64–83, Jan 2003.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10:563–579, August 2005.

[6] 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.

[7] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

[8] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. A comprehensive experimental comparison of event driven and multi-threaded sensor node operating systems. *JNW*, 3(3):57–70, 2008.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

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

[12] D. Gay, M. Welsh, P. Levis, E. Brewer, R. von Behren, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI'03*, pages 1–11, 2003.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, September 1991.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, November 2000.

[15] R. Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39:221–258, March 2009.

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

[17] G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 335–346, New York, NY, USA, 2008. ACM.

[18] R. Sugihara and R. K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4:8:1–8:29, April 2008.

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