Céu: A Reactive Language for Wireless Sensor Networks

Francisco Sant'Anna
Departamento de Informática - PUC-Rio
fsantanna@inf.puc-rio.br

Abstract

CÉU is a system-level reactive language targeting Wireless Sensor Networks that poses an alternative to the predominating event-driven and threaded-based systems. CÉU supports concurrent lines of execution that are allowed to share variables. However, the static nature of CÉU enables a compile time analysis that ensures safe and deterministic execution. The CÉU compiler generates single-threaded *C* code that is comparable in size to handcrafted event-driven code.

1 Biography

Francisco Sant'Anna is a second year Ph.D. student at PUC-Rio with expected graduation date in September, 2013. His advisor is Roberto Ierusalimschy, associate professor at PUC-Rio in the field of programming languages, and the creator of the Lua language. His co-advisor is Noemi de La Roque Rodriguez, associate professor at PUC-Rio in the field of distributed systems.

2 Introduction

Three aspects have been used to compare system programming languages for Wireless Sensor Networks: *memory usage*, event processing (*responsiveness*), and *energy consumption* [3].

We consider that *safety* is also an important aspect, as motes must run for long periods without human intervention. In our discussion, we confine the term safety to deterministic and bounded execution (i.e. programs should not execute long loops). Current system languages for WSNs (e.g. [6, 4, 2]) do not detect such safety properties, requiring the programmer to perform exhaustive testing. For instance, preemptive multithreading is non-deterministic by design, while event-driven and cooperative multithreading are susceptible to unbounded execution.

Finally, *expressiveness* is another key aspect, regarding how programmers are able to write concise and maintainable programs. Event-driven programming is intrinsically unstructured, while cooperative and preemptive mutilthreading require, respectively, explicit scheduling and synchronization, besides all exercise related to the life cycle of threads.

In our thesis, we present CÉU, a reactive language inspired in Esterel [1] and FRP [5] that aims to improve the safety and expressiveness of current system languages for WSNs.

CÉU relies on a compile-time analysis to detect unbounded loops and concurrent access to variables. The static analysis forbids 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 (such as *malloc*) are discouraged due to the resource limitations and safety requirements.

WSNs applications typically react to multiple external events concurrently (e.g. timers, radio message arrivals, etc). CÉU supports multiple lines of execution that can handle different events in parallel. A line of execution can await an event without loosing context information, such as locals and the program counter.

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. Waiting for an event halts the running trail until that event occurs. The environment broadcasts occurring events to all active trails, which share a single global time reference (an event itself).

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

```
(~Radio_recv ~> v)* || (~v ~> Leds_set)*
```

The first trail (on the left of the `||` parallel operator) awaits (~) the external input event *Radio_recv*, 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*.

The example could be simplified and rewritten just as `(~Radio_recv ~> Leds_set)*`, but would not illustrate the concurrent nature of CÉU, though.

The core BNF-like syntax of CÉUis the following:[1]

```
e ::=  e;e    |   e?e:e    (sequence, conditional)
   |   e||e   |   e&&e     (parallel or, and)
   |   e*     |   e^       (loop, loop escape)
   |   {e}                 (scope block)
   |   ID     |   e=>ID    (variable read, write)
   |   ~ID    |   e~>ID    (event await, trigger)
   |   e,e    |   e->ID    (operation params, call)
```

Note the syntax for attributions, triggers, and calls, where the source expressions come first (resembling a dataflow style). Operators are defined conforming to a standard interface in a host language (e.g. functions in C).

A parallel expression executes its subexpressions in concurrent trails, terminating when one of them (*par/or*), or both (*par/and*) terminate. 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 expression 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 4). 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 as a *reaction chain*, and always runs in bounded time.
3. If the program does not terminate, then it goes idle and the environment takes the control.
4. On the occurrence of a new input event, the environment awakes the program on its awaiting trails. Then, goes to step 2.

If a new 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.

Every variable in CÉU is also an internal event and vice-versa. By triggering an internal event with a value also assigns that value to it. For this reason, internal events are also known as *reactive variables*. The following program fragment specifies that whenever the variable $v1$ changes, $v2$ is automatically updated to the increment of $v1$, which in turn, automatically updates $v3$ to the increment of $v2$:

```
(~v1->inc ~> v2)* || (~v2->inc ~> v3)*
```

In contrast with external events, which are handled in a queue, internal events follow a stack policy and react within the same propagation 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 within the same time unit.

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

4 Safety

A reaction chain must run in bounded time to ensure that a program is responsive and can handle upcoming 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 do not enter in loop.

For CÉU *loops*, we restrict that they must contain at least one *await* or *break* expression on theirs bodies for each possible path within them. For instance, 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 expression holds this restriction or not.

Determinism is usually a desired safety property, making programs more predictable and easier to debug. In CÉU, there are three possible sources of non-determinism: *concurrent access to variables* (e.g. `(1=>a&&2=>a)`), *concurrent* par/or *termination* (e.g. `(1||2)=>a`, might yield 1 or 2), and *concurrent loop escape*[2] (e.g. `(1^ && 2^)* =>a`).

During compile time, CÉU converts programs into deterministic finite automatons in order to detect the three forms of non-determinism. This conversion is the reason why CÉU is a static language. A DFA unequivocally represents a CÉU program, covering exactly all possible paths it can reach during runtime. For instance, the following program is identified as non-deterministic, because the variable v is accessed concurrently on the 6th occurrence of the event A:

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

5 Physical Time

Physical time[3] is probably the most common input in WSN applications, as found in typical patterns, such as sensor sampling, and watchdogs. However, system languages support for physical time is somewhat low-level, usually through *timer* callbacks or *sleep* blocking calls. CÉU provides a first-class support for physical time: the expression `~1s500ms` awaits one second and a half.

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 does. Likewise, 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 20*ms*.

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

[2]The token ^ escapes the innermost loop with its preceding expression.

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

6 Evaluation

From the aspects we want to evaluate in CÉU—*memory* and *battery* consumption, *responsiveness*, *safety*, and *expressiveness*— we already have quantitative measures for memory usage and expressiveness (in terms of source code size).

We ported existing TinyOS/nesC[4] [6] applications to CÉU to support our experiments. The following table shows the measures for 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.

		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**

Our experiments suggest that as the applications 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.[5] With this approach we focused 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.

7 Related Work

Our work is influenced by the Esterel language [1], an imperative reactive language with similar constructs. Karpinski and Cahill [7] present a language targeting WSNs (also based on Esterel), and perform a throughout quantitative and qualitative comparison with nesC. CÉU differs from these languages with its semantics for internal events, physical time, and a more consistent support for concurrent access to variables.

Protothreads [4] offer very lightweight threads with blocking support. Its stackless implementation reduces memory consumption but prevents automatic variables. Protothreads provide no safety support besides atomic execution of threads: a program can loop indefinitely, and access to globals is unrestricted.

[4]We chose to use TinyOS due to its simplicity and acceptance in the WSN community. We are using *TinyOS* − 2.1.1 and *micaz* motes in our experiments.

[5]The original and modified sources for the experiments can be found at `www.lua.inf.puc-rio.br/~francisco/sensys_11.html`.

8 Conclusion

We believe that CÉU poses concrete advantages in terms of safety and expressiveness when compared to current system languages for WSNs. Regarding safety, we propose a temporal analysis of 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.

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.

At this point, we did not evaluate battery consumption and responsiveness aspects, but we plan to perform quantitative analysis for them. Responsiveness, in particular, deals with long running tasks that currently require explicit yields inside loops (e.g. with `~1ms`). We are working on a lightweight asynchronous extension to CÉU to address this limitation.

We intend to port more complex applications to CÉU to improve our evaluation. We are aware of the limitations of evaluating the expressiveness of CÉU based solely on LOCs, though. On the way to a more in-depth qualitative approach, we are teaching CÉU as an alternative to nesC in introductory courses on WSNs for undergraduate and also high school students. We will compare the achievements of the students with both models and use the results in our evaluation.

9 References

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

[2] 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.

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

[4] 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.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, New York, NY, 1997. ACM.

[6] 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.

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