V
Evaluation

In this chapter we present a quantitative evaluation of Céu. Our assumption is that when considering Céu for system-level development, programmers would face a tradeoff between code simplicity and efficient resource usage. For this reason, we evaluate source code size, memory usage, event-handling responsiveness, and battery consumption for a number of standardized protocols in TinyOS [39]. We use code size as a metric for code simplicity, complemented with a qualitative discussion regarding the eradication of explicit state variables for control purposes. By responsiveness, we mean how quickly programs react to incoming events (to avoid missing them). Memory, responsiveness are important resource-efficiency measures to evaluate the negative impact with the adoption of a higher-level language. In particular, responsiveness (instead of total CPU cycles) is a critical aspect in reactive systems, specially those with a synchronous execution semantics where preemption is forbidden. We also discuss battery consumption when evaluating responsiveness.

Our criteria to choose which language and applications to compare with Céu are based on the following guidelines:

- Compare to a resource-efficient programming language in terms of memory and speed.
- Compare to the best available codebase, with proved stability and quality.
- Compare relevant protocols in the context of WSNs.
- Compare the control-based aspects of applications, as Céu is designed for this purpose.
- Compare the radio behavior, the most critical and battery-drainer component in WSNs.

Based on these criteria, we chose *nesC* as the language to compare, given its resource efficiency and high-quality codebase[1]. In addition, *nesC* is used as benchmark in many systems related to Céu [14, 27, 5, 4]. In particular,

[1] TinyOS repository: http://github.com/tinyos/tinyos-release/

			Code size				Céu features					Memory usage			
Component	Application	Language	tokens	*Céu vs nesC*	globals		local data variables	internal events	first-class timers	parallel comp.	max. number or trails	ROM	*Céu vs nesC*	RAM	*Céu vs nesC*
					state	data									
CTP	*TestNetwork*	nesC	383	-23%	4	5	2;5;6	2	3	5	8	18896	9%	1295	2%
		Céu	295		-	2						20542		1319	
SRP	*TestSrp*	nesC	418	-30%	2	8	2;2;2;-	1	-	1	3	12266	5%	1252	-3%
		Céu	291		-	4						12836		1215	
DRIP	*TestDissemination*	nesC	342	-25%	2	1	4	1	-	1	5	12708	8%	393	4%
		Céu	258		-	-						13726		407	
CC2420	*RadioCountToLeds*	nesC	519	-27%	1	2	3;3	1	-	2	4	10546	2%	283	3%
		Céu	380		-	-						10782		291	
Trickle	*TestTrickle*	nesC	477	-69%	2	2	2;5	-	2	3	6	3504	22%	72	22%
		Céu	149		-	-						4284		88	

Figure V.1: Comparison between Céu and *nesC* for the implemented applications.
The column group *Code size* compares the number of language tokens and global variables used in the sources; the group *Céu features* shows the number of times each functionality is used in each application; the group *Memory usage* compares ROM and RAM consumption.

the work on *Protothreads* [14] is a strong reference in the WSN community, and we adhere to similar choices in our evaluation. All chosen applications are reference implementations of open standards in the TinyOS community [39]: the receiving component of the *CC2420* radio driver; the *Trickle* timer; the *SRP* routing protocol; the *DRIP* dissemination protocol; and the routing component of the *CTP* collection protocol. They are representative of the realm of system-level development for WSNs, which mostly consists of network protocols and low-level system utilities: a radio driver is mandatory in the context of WSNs; the trickle timer is used as a service by other important protocols [31, 20]; routing, dissemination, and collection are the most common classes of protocols in WSNs.

We took advantage of the component-based model of TinyOS and all of our implementations use the same interface provided by the *nesC* counterpart. This approach has two advantages: first, we could reuse existing applications in the TinyOS repository to test the protocols (e.g. *RadioCountToLeds* or *TestNetwork*); second, sticking to the same interface forced us to retain the original architecture and functionality, which also strengths our evaluation.

Figure V.1 shows the comparison for *Code size* and *Memory usage* between the implementations in *nesC* and Céu. For memory usage, detailed in Section V.2, we compare the binary code size and required RAM. For code size, detailed in Section V.1, we compare the number of tokens used in the source code. For responsiveness, detailed in Section V.3, we evaluate the capacity to promptly acknowledge radio packet arrivals in the *CC*2420 driver.

V.1 Code size

We use two metrics to compare code complexity between the implementations in CÉU and *nesC*: the number of language tokens and global variables used in the source code. Similarly to comparisons in related work [5, 14], we did not consider code shared between the *nesC* and CÉU implementations (e.g. predicates, `struct` accessors, etc.), as they do not represent control functionality and pose no challenges regarding concurrency aspects.

Note that the languages share the core syntax for expressions, calls, and field accessors (based on *C*), and we removed all verbose annotations from the *nesC* implementations for a fair comparison (e.g. `signal`, `call`, `command`, etc.). The column *Code size* in Figure V.1 shows a considerable decrease in the number of tokens for all implementations (around at least 25%).

Regarding the metrics for number of globals, we categorized them in *state* and *data* variables.

State variables are used as a mechanism to control the application flow (on the lack of a better primitive). Keeping track of them is often regarded as a difficult task, hence, reduction of state variables has already been proposed as a metric of code complexity in a related work [14]. The implementations in CÉU, not only reduced, but completely eliminated state variables, given that all control patterns could be expressed with hierarchical compositions of activities assisted by internal-event communication.

Data variables in WSN programs usually hold message buffers and protocol parameters (e.g. sequence numbers, timer intervals, etc.). In event-driven systems, given that stacks are not retained across reactions to the environment, all data variables must be global[2]. Although the use of local variables does not imply in reduction of lines of code (or tokens), the smallest the scope of a variable, the more readable and less susceptible to bugs the program becomes. In the CÉU implementations, most variables could be nested to a deeper scope. The column *local data variables* in Figure V.1 shows the depth of each new local variable in CÉU that was originally a global in *nesC* (e.g. "2;5;6" represents globals that became locals inside blocks in the 2nd, 5th, and 6th depth level).

The columns under *Céu features* in Figure V.1 point out how many times each functionality has been used in the implementations in CÉU, helping to identify where the reduction in size comes from. As an example, Trickle uses 2 timers and 3 parallel compositions, resulting in at most 6 trails active at the same time. The use of six coexisting trails for such a small application

[2]In the case of *nesC*, we refer to globals as all variables defined in the top-level of a component implementation block, which are visible to all functions inside it.

is justified by its highly control-intensive nature, and the almost 70% code reduction illustrates the huge gains with Céu in this context.

V.2 Memory usage

Memory is a scarce resource in motes and it is important that Céu does not pose significant overheads in comparison to *nesC*. We evaluate ROM and RAM consumption by using available testing applications for the protocols in the TinyOS repository. Then, we compiled each application twice: first with the original component in *nesC*, and then with the new component in Céu. Column *Memory usage* in Figure V.1 shows the consumption of ROM and RAM for the generated applications. With the exception of the Trickle timer, the results in Céu are below 10% in ROM and 5% in RAM, in comparison with the implementations in *nesC*. Our method and results are similar to those for Protothreads [14], which is an actively supported programming system for the Contiki OS [13].

Note that the results for Trickle illustrate the footprint of the runtime of Céu. The RAM overhead of 22% actually corresponds to only 16 bytes: 1 byte for each of the maximum 6 concurrent trails, and 10 bytes to handle synchronization among timers. As the complexity of the application grows, this basic overhead tends to become irrelevant. The SRP implementation shows a decrease in RAM, which comes from the internal communication mechanism of Céu that could eliminate a queue. Note that both TinyOS and Céu define functions to manipulate queues for timers and tasks (or trails). Hence, as our implementations use components in the two systems, we pay an extra overhead in ROM for all applications.

We focused most of the language implementation efforts on RAM optimization, as it has been historically considered more scarce than ROM [32]. Although we have achieved competitive results, we expected more gains with memory reuse for blocks with locals in sequence, because it is something that cannot be done automatically by the *nesC* compiler. However, we analyzed each application and it turned out that we had no gains *at all* from blocks in sequence. Our conclusion is that sequential patterns in WSN applications come either from split-phase operations, which always require memory to be preserved (between the request and the answer); or from loops, which do reuse all memory, but in the same way that event-driven systems do.

V.3 Responsiveness

A known limitation of languages with synchronous and cooperative execution is that they cannot guarantee hard real-time deadlines [12, 29]. For instance, the rigorous synchronous semantics of Céu forbids non-deterministic preemption to serve high priority trails. Even though Céu ensures bounded execution for reactions, this guarantee is not extended to *C* function calls, which are usually preferred for executing long computations (due to performance and existing code base). The implementation of a radio driver purely in Céu raises questions regarding its responsiveness, therefore, we conduct two experiments in this section. The experiments use the *COOJA* simulator [15] running images compiled to *TelosB* motes.

In the first experiment, we "stress-test" the radio driver to compare its performance in the Céu and *nesC* implementations. We use 10 motes that broadcast 100 consecutive packets of 20 bytes to a mote that runs a periodic time-consuming activity. The receiving handler simply adds the value of each received byte to a global counter. The sending rate of each mote is 200ms (leading to a receiving average of 50 packets per second considering the 10 motes), and the time-consuming activity in the receiving mote runs every 140ms. Note that these numbers are much above typical WSN applications: 10 neighbours characterizes a dense topology; 20 bytes plus header data is close to the default limit for a TinyOS packet; and 5 messages per second is a high frequency on networks that are supposedly idle most of the time. We run the experiment varying the duration of the lengthy activity from 1 to 128 milliseconds, covering a wide set of applications (summarized in Table V.1). We assume that the lengthy operation is implemented directly in *C* and cannot be easily split in smaller operations (e.g., recursive algorithms [12, 29]). So, we simulated them with simple busy waits that would keep the driver in Céu unresponsive during that period.

Figure V.2 shows the percentage of handled packets in Céu and *nesC* for each duration. Starting from the duration of 6ms for the lengthy operation, the responsiveness of Céu degrades in comparison to *nesC* (5% of packet loss). The *nesC* driver starts to become unresponsive with operations that take 32ms, which is a similar conclusion taken from TOSThreads experiments with the same hardware [29]. Table V.1 shows the duration of some lengthy operations specifically designed for WSNs found in the literature. The operations in the group with timings up to 6*ms* could be used with real-time responsiveness in Céu (considering the proposed high-load parameters).

Although we did not perform specific tests to evaluate CPU usage, the

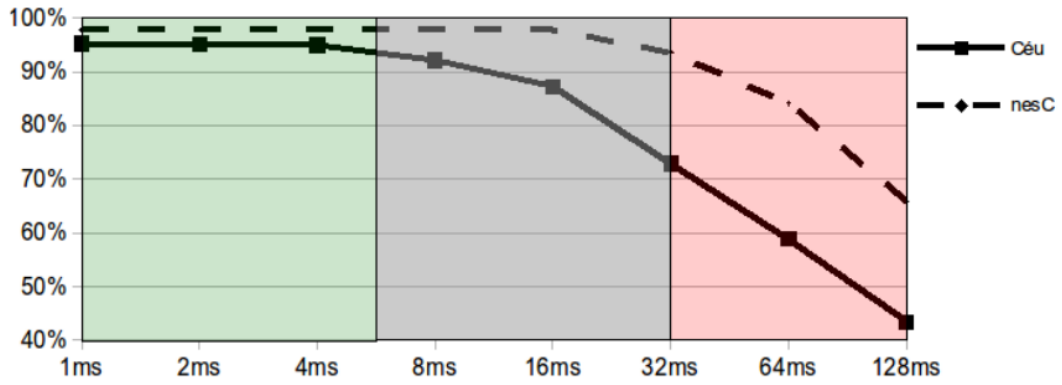

Figure V.2: Percentage of received packets depending on the duration of the lengthy operation.
Note the logarithmic scale on the x-axis. The packet arrival frequency is 20ms. The operation frequency is 140ms. In the (left) green area, Céu performs similarly to *nesC*. The (middle) gray area represents the region in which *nesC* is still responsive. In the (right) red area, both implementations become unresponsive (i.e. over 5% packet losses).

Operation	Duration
Block cypher [26, 18]	1ms
MD5 hash [18]	3ms
Wavelet decomposition [41]	6ms
SHA-1 hash [18]	8ms
RLE compression [38]	70ms
BWT compression [38]	300ms
Image processing [37]	50–1000ms

Table V.1: Durations for lengthy operations is WSNs.
Céu can perform the operations in the green rows in real-time and under high loads.

experiment suggests that the overhead of Céu over *nesC* is very low. When the radio driver is the only running activity (column $1ms$, which is the same result for an addition test we did for $0ms$), both implementations loose packets with a difference under 3 percentage points. This difference remains the same up to 4-ms activities, hence, the observed degradation for longer operations is only due to the lack of preemption, not execution speed. Note that for lengthy operations implemented in *C*, there is no runtime overhead at all, as the generated code is the same for Céu and *nesC* (i.e. Céu and *nesC* just call *C*).

In the second experiment, instead of running a long activity in parallel, we use a 8-ms operation tied in sequence with every packet arrival to simulate an activity such as encryption. We now run the experiment varying the rate

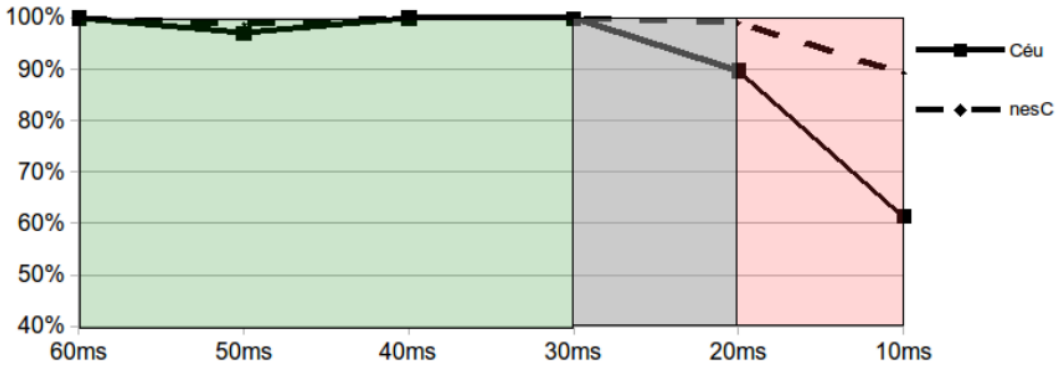

Figure V.3: Percentage of received packets depending on the sending frequency. Each received packet is tied to a 8-ms operation. Céu is 100% responsive up to a frequency of 30ms per packet.

in the 10 sending motes from 600ms to 100ms (i.e., 60ms to 10ms receiving rate if we consider the 10 motes). Figure V.3 shows the percentage of handled packets in Céu and *nesC* for each rate of message arrival. The results show that Céu is 100% responsive up to frequency of 33 packets per second, while *nesC* up to 50 packets.

The overall conclusion from the experiments is that the radio driver in Céu performs as well as the original driver in *nesC* under high loads for programs with lengthy operations of up to 4ms, which is a reasonable time for control execution and simple processing. The range between 6ms and 16ms offers opportunities for performing more complex operations, but also requires careful analysis and testing. For instance, the last experiment shows that the Céu driver can process in real time messages arriving every 33ms in sequence with a 8-ms operation.

Note that our experiments represent a "stress-test" scenario that is atypical to WSNs. Protocols commonly use longer intervals between message transmissions together with mechanisms to avoid contention, such as randomized timers [31, 20]. Furthermore, WSNs are not subject to strict deadlines, being not classified as hard real-time systems [32].

V.4 Battery consumption

Battery consumption is critical in WSNs, given that motes usually have no other source of energy and, in the case of being deployed in remote locations, cannot have the batteries replaced.

In order to evaluate battery consumption in Céu in comparison to *nesC*, we adapted the experiments of Section V.3. The parameters were adjusted to make the implementations in the two languages behave the same, i.e., the

	Experiment 1			Experiment 2		
	nesC	Céu	**nesC/Céu**	nesC	Céu	**nesC/Céu**
Total (J)	1.53	1.52	1.01	2.28	2.27	1.00
Active (J)	0.04	0.03	~~1.56~~	1.38	1.38	1.00
Idle (J)	1.49	1.49	1.00	0.89	0.89	1.00

Figure V.4: Battery consumption for *nesC* and CÉU in the two experiments. The consumption line "Active" for the Experiment 1 is negligible, hence, the ratio between *nesC* and CÉU should not be considered.

receiving node should receive the same amount of packets during the same period.

For the first experiment, we made each sending node transmit 75 messages during 150s, resulting in around 625 received packets (considering the losses) in the receiving mote, which also performs a 1-ms heavy activity every 1.5 seconds. In this experiment, the CPU is idle most of the time and the battery is consumed by the radio hardware. For the second experiment, we included a 2-ms heavy activity after every received packet, making the battery to be also consumed by the CPU. Figure V.4 shows the battery consumption (total and with active and idle CPU) for the two experiments and for both implementations.

We did not expect a noticeable difference in battery usage between CÉU and *nesC*, because, even considering the support for multiple lines of execution in CÉU, the compiler generates simple event-driven code in *C*, not requiring threads or complex runtime apparatus. In fact, the results are virtually the same for both I/O and CPU-bound experiments.

V.5 Discussion

CÉU targets control-intensive applications and provides abstractions that can express program flow specifications concisely. Our evaluation shows a considerable decrease in code size that comes from logical compositions of trails through the `par/or` and `par/and` constructs. They handle startup and termination for trails seamlessly without extra programming efforts. We believe that the small overhead in memory qualifies CÉU as a realistic option for constrained devices. Furthermore, our broad safety analysis, encompassing all proposed concurrency mechanisms, ensures that the high degree of concurrency in WSNs does not pose safety threats to applications. As a summary, the following safety properties hold for all programs that successfully compile in CÉU:

– Time-bounded reactions to the environment (Sections III.1 and III.6).

- Reliable weak and strong abortion among activities (Sections III.1 and III.2).
- No concurrency in accesses to shared variables (Section III.2).
- No concurrency in system calls sharing a resource (Section III.3).
- Finalization for blocks going out of scope (Section III.4).
- Auto-adjustment for timers in sequence (Section III.5).
- Synchronization for timers in parallel (Section III.5).

These properties are desirable in any application and are guaranteed as preconditions in Céu by design. Ensuring or even extracting these properties from less restricted languages requires significant manual analysis.

Even though the achieved expressiveness and overhead of Céu meet the requirements of WSNs, its design imposes two inherent limitations: the lack of dynamic loading which would forbid the static analysis, and the lack of hard real-time guarantees. Regarding the first limitation, dynamic features are already discouraged due to resource constraints. For instance, even object-oriented languages targeting WSNs forbid dynamic allocation [4, 40].

To deal with the second limitation, which can be critical in the presence of lengthy computations, we can consider the following approaches: (1) manually placing `pause` statements in unbounded loops; (2) integrating Céu with a preemptive system. The first option requires the lengthy operations to be rewritten in Céu using `pause` statements so that other trails can be interleaved with them. This option is the one recommended in many related work that provide a similar cooperative primitive (e.g. `pause` [6], `PT_YIELD` [14], `yield` [27], `post` [19]). Considering the second option, Céu and preemptive threads are not mutually exclusive. For instance, TOSThreads [29] proposes a message-based integration with *nesC* that is safe and matches the semantics of Céu external events.