Francisco Figueiredo Goytacaz Sant'Anna

Safe System-level Concurrency on Resource-Constrained Nodes with Céu

PhD Thesis

Thesis presented to the Post–graduate Program in Computer Science of the Informática Department, PUC–Rio as partial fulfillment of the requirements for the degree of Doctor in Philosophy in Computer Science

Adviser : Prof. Roberto Ierusalimschy
Co–Adviser: Prof. Noemi de La Roque Rodriguez

Rio de Janeiro
September 2013

Francisco Figueiredo Goytacaz Sant'Anna

Safe System-level Concurrency on Resource-Constrained Nodes with Céu

Thesis presented to the Post–graduate Program in Computer Science of the Informática Department, PUC–Rio as partial fulfillment of the requirements for the degree of Doctor in Philosophy in Computer Science. Approved by the following commission:

Prof. Roberto Ierusalimschy
Adviser
Department of Informática — PUC–Rio

Prof. Noemi de La Roque Rodriguez
Co–Adviser
Department of Informática — PUC–Rio

Rio de Janeiro — September 12, 2013

All rights reserved.

Francisco Figueiredo Goytacaz Sant'Anna

Bibliographic data

Sant'Anna, Francisco Figueiredo Goytacaz

Safe System-level Concurrency on Resource-Constrained Nodes with Céu / Francisco Figueiredo Goytacaz Sant'Anna; adviser: Roberto Ierusalimschy; co–adviser: Noemi de La Roque Rodriguez. — Rio de Janeiro : PUC–Rio, Department of Informática, 2013.

v., 86 f: il. ; 29,7 cm

1. PhD Thesis - Pontifícia Universidade Católica do Rio de Janeiro, Department of Informática.

Bibliography included.

1. Informática – Thesis. 2. Concurrency. 3. Determinism. 4. Embedded Systems. 5. Esterel. 6. Reactivity. 7. Synchronous. 8. Wireless Sensor Networks. I. Ierusalimschy, Roberto. II. Rodriguez, Noemi de La Roque. III. Pontifícia Universidade Católica do Rio de Janeiro. Department of Informática. IV. Title.

CDD: 004

Acknowledgments

I would like to thank my supervisors Roberto Ierusalimschy and Noemi Rodriguez for their time and knowledge sharing.

TTT: Olaf / Philippas

I'm also thankful to Luiz Fernando Soares, for my period in the Telemídia Lab, and all my teachers in CAp-UERJ, responsible for my education before college.

My studies were funded by CNPq and SAAB.

Abstract

Sant'Anna, Francisco Figueiredo Goytacaz; Ierusalimschy, Roberto; Rodriguez, Noemi de La Roque. **Safe System-level Concurrency on Resource-Constrained Nodes with Céu**. Rio de Janeiro, 2013. 86p. PhD Thesis — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Despite the continuous research to facilitate Wireless Sensor Networks development, most safety analysis and mitigation efforts in concurrency are still left to developers, who must manage synchronization and shared memory explicitly. We propose a system language that ensures safe concurrency by handling threats at compile time, rather than at runtime. The synchronous and static foundation of our design allows for a simple reasoning about concurrency that enables compile-time analysis resulting in deterministic and memory-safe programs. As a trade-off, our design imposes limitations on the language expressiveness, such as doing computationally-intensive operations and meeting hard real-time responsiveness. To show that the achieved expressiveness and responsiveness is sufficient for a wide range of WSN applications, we implement widespread network protocols and the CC2420 radio driver. The implementations show a reduction in source code size, with a penalty of memory increase below 10% in comparison to *nesC*. Overall, we ensure safety properties for programs relying on high-level control abstractions that also lead to concise and readable code.

Keywords

Concurrency. Determinism. Embedded Systems. Esterel. Reactivity. Synchronous. Wireless Sensor Networks.

Contents

I Introduction **15**

II Overview of programming models **19**
II.1 Asynchronous model 19
II.2 Synchronous model 21
II.3 Programming models in WSNs 22

III The design of Céu **25**
III.1 The execution model of Céu 25
III.2 Shared-memory concurrency 31
III.3 Integration with *C* 33
III.4 Local scopes and finalization 34
III.5 First-class timers 36
III.6 Internal events 38
III.7 Differences to Esterel 40

IV Demo applications **43**
IV.1 WSN ring 43
IV.2 Spaceship game 46

V Evaluation **51**
V.1 Code size 52
V.2 Memory usage 53
V.3 Responsiveness 54
V.4 Discussion 57

VI The semantics of Céu **59**
VI.1 Abstract syntax 59
VI.2 Operational semantics 60
VI.3 Concrete language mapping 65

VII The implementation of Céu **69**
VII.1 Temporal analysis 70
VII.2 Memory layout 71
VII.3 Trail allocation 71
VII.4 The external *C* API 75

VIIIRelated work **79**

IX Conclusion **81**

List of Figures

II.1 Two blinking LEDs in OCCAM-PI, ChibiOS and Céu.
Each line of execution in parallel blinks a LED with a fixed (but different) frequency. (The LEDs are connected to I/O ports 11 and 12.) Every 3 seconds both LEDs should light on together. After a couple of minutes of execution, only the implementation in Céu remains synchronized. 20
II.2 "Blinking LED" in nesC, Protothreads, and Céu. 24

III.1 Syntax of Céu. 26
III.2 A Céu program to illustrate the scheduler behavior. 27
III.3 A sequence of reaction chains for the program in Figure III.2. 28
III.4 Start/stop behavior for the radio driver.
The occurrence of `CC2420_STOP` (line 5) seamlessly aborts the receiving loop (collapsed in line 9) and resets the driver to wait for the next `CC2420_START` (line 3). 30
III.5 Automatic detection for concurrent accesses to shared memory.
The first example is suspicious because `x` and `p` can be accessed concurrently (lines 11 and 14). The second example is safe because accesses to `y` can only occur in sequence. The third example illustrates a false positive in our algorithm. 32
III.6 A Céu program with embedded *C* definitions. The globals `I` and `inc` are defined in the `native` block (lines 3 and 4), and are imported by Céu in line 8. *C* symbols must be prefixed with an underline to be used in Céu (line 9). 33
III.7 Annotations for *C* functions.
Function `abs` is side-effect free and can be concurrent with any other function. The functions `_Leds_led0Toggle` and `_Leds_led1Toggle` can execute concurrently. The variables `buf1` and `buf2` can be accessed concurrently (annotations are also applied to variables). 34
III.8 Unsafe use of local references.
The period in which the radio driver manipulates the reference to `msg` passed by `_AMSend_send` (line 15) may outlive the lifetime of the variable scope, leading to an undefined behavior in the program. 35
III.9 A loop that awaits an internal event can emulate a subroutine.
The `send` "subroutine" (lines 16-19) is invoked from three different parts of the program (lines 5, 9, and 14). 39
III.10 Exception handling in Céu.
The `emit`'s in lines 10 and 14 raise an exception to be caught by the `await` in line 6. The `emit` continuations are discarded given that the surrounding `par/or` is aborted. 41

IV.1 Communicating trail for the WSN ring. 44
IV.2 Monitoring trail for the WSN ring. 45

IV.3 Retrying trail for the WSN ring. 46
IV.4 Retrying trail for the WSN ring. 47
IV.5 The “spaceship” game 47
IV.6 Outermost loop for the game. 48
IV.7 Sets the game attributes. 48
IV.8 The game central loop. 49
IV.9 The “game over” behavior for the game. 50

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. 52
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). 56
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. 57

VI.1 Reduced syntax of CÉU. 59
VI.2 The recursive predicate *isBlocked* is true only if all branches in parallel are hanged in *awaiting* or *emitting* expressions that cannot transit. 63
VI.3 The function *clear* extracts fin expressions in parallel and put their bodies in sequence. 64

VII.1 Compilation process: from the source code in CÉU to the final binary. 69
VII.2 A program with a corresponding AST describing the sets I and O. The program is safe because accesses to `y` in parallel have no intersections for I. 71
VII.3 A program with blocks in sequence and in parallel, with corresponding memory layout. 72
VII.4 Static allocation of trails and entry-point labels. 73
VII.5 Generated code for the program of Figure VII.4. 74
VII.6 The *TinyOS* binding for CÉU. 77

VIII.1 Table of features found in work related to CÉU.
The languages are sorted by the date they first appeared in a publication. A gray background indicates where the feature first appeared (or a contribution if it appears in a CÉU cell). 80

I

Introduction

Wireless sensor networks (WSNs) are composed of a large number of tiny devices (known as “motes”) capable of sensing the environment and communicating. They 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].

WSNs are usually designed with safety and (soft) 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 WSN-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, WSN languages either lack functionality or fail to offer a small and reliable programming environment.

System-level development for WSNs commonly follows one of three major programming models: *event-driven*, *multi-threaded*, and *synchronous*. In event-driven programming [23, 13], each external event can be associated with a short-lived function callback to handle a reaction to the environment. This model is efficient, but is known to be difficult to program [1, 14]. Multi-threaded systems emerged as an alternative in WSNs, providing traditional structured programming in multiple lines of execution [14, 9]. However, the development process still requires manual synchronization and bookkeeping of threads [30]. Synchronous languages [3] have also been adapted to WSNs and offer higher-level compositions of activities with a step-by-step execution, considerably reducing programming efforts [27, 28].

Despite the increase in development productivity, WSN system languages still fail to ensure static safety properties for concurrent programs. However, given the difficulty in debugging WSN applications, it is paramount to push as many safety guarantees to compile time as possible [32]. Shared-memory concurrency is an example of a widely adopted mechanism that typically relies on runtime safety primitives only. For instance, current WSN languages ensure atomic memory access either through runtime barriers, such as mutexes and locks [9, 33], or by adopting cooperative scheduling which also requires explicit yield points in the code [27, 14]. In either case, there is no additional static guarantees or warnings about unsafe memory accesses.

We believe that programming WSNs can benefit from a new language that takes concurrency safety as a primary goal, while preserving typical multi-threading features that programmers are familiarized with, such as shared memory concurrency. We present CÉU[1], a synchronous system-level programming language that provides a reliable yet powerful set of abstractions for the development of WSN applications. CÉU is based on a small set of control primitives similar to Esterel's [10], leading to implementations that more closely reflect program specifications. As a main contribution, we propose a static analysis that permeates all language mechanisms and detects safety threats, such as concurrent accesses to shared memory and concurrent termination of threads, at compile time. In addition, we introduce the following new safety mechanisms: *first-class timers* to ensure that timers in parallel remain synchronized (not depending on internal reaction timings); *finalization blocks* for local pointers going out of scope; and *stack-based communication* that avoids cyclic dependencies. Our work focuses on *concurrency safety*, rather than *type safety* [11].[2]

In order to enable the static analysis, programs in CÉU must suffer some limitations. Computations that run in unbounded time (e.g., compression, image processing) cannot be elegantly implemented [36], and dynamic loading is forbidden. However, we show that CÉU is sufficiently expressive for the context of WSN applications. We successfully implemented the *CC2420* radio driver, and the *DRIP*, *SRP*, and *CTP* network protocols [39] in CÉU. In comparison to *nesC* [19], the implementations reduced the number of source code tokens by 25%, with an increase in ROM and RAM below 10%.

The rest of the thesis is organized as follows: Chapter II introduces CÉU through comparisons with state-of-the-art languages representing the prevailing concurrency models used in WSNs. Chapter III details the design of CÉU, motivating and discussing the safety aspects of each relevant language feature. Chapter IV presents two demo applications, exploring the safe and high-level programming style promoted by the language. Chapter V evaluates the implementation of some network protocols in CÉU and compares some of its aspects with *nesC* (e.g. memory usage and tokens count). We also evaluate the responsiveness of the radio driver written in CÉU. Chapter VI presents a formal semantics of CÉU restricted to its control primitives, which comprises most novelties and challenging parts of our design. Chapter VII discusses key aspects of the implementation of CÉU, such as the static analysis algorithm and stackless lines of execution. Chapter VIII discusses related work to CÉU. Chapter IX concludes the thesis and makes final remarks.

---

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

[2] We consider both safety aspects to be complimentary and orthogonal, i.e., type-safety techniques could also be applied to CÉU.

II
Overview of programming models

Concurrent languages can be generically classified in two major execution models. In the *asynchronous model*, the program activities (e.g. threads and processes) run independently of one another as result of non-deterministic preemptive scheduling. In order to coordinate at specific points, these activities require explicit use of synchronization primitives (e.g. mutual exclusion and message passing). In the *synchronous model*, the program activities (e.g. callbacks and coroutines) require explicit control/scheduling primitives (e.g. returning or yielding). For this reason, they are inherently synchronized, as the programmer himself specifies how they execute and transfer control.

In this chapter we give an overview of these models, focusing on the synchronous model, given that CÉU and most related work targeting WSNs [19, 14, 28, 33, 27, 4, 5] (detailed in Chapter VIII) are synchronous.

II.1 Asynchronous model

The asynchronous model of computation can be classified according to how independent activities coordinate. In *shared memory* concurrency, communication is via global state, while synchronization is via mutual exclusion. In *message passing*, both communication and synchronization happen via exchanging messages.

The default behavior of activities being independent hinders the development of highly synchronized applications. As a practical evidence, Figure II.1 shows a simple application that blinks two LEDs in parallel with different frequencies[1]. We implemented it in two asynchronous styles and also in CÉU. For *shared memory* concurrency, we used a multithreaded RTOS[2], while for message passing, we used an *occam* variation for Arduino [25].

The LEDs should blink together every 3 seconds (LCM between 600ms and 1s). As we expected, even for such a simple application, the LEDs in

---

[1] The complete source code and a video demos for the application can be found at http://www.ceu-lang.org/TR/#blink.

[2] http://www.chibios.org/dokuwiki/doku.php?id=start

```
// OCCAM-PI
PROC main ()
  CHAN SIGNAL s1,s2:
  PAR
    PAR
      tick(600, s1!)
      toggle(11, s1?)
    PAR
      tick(1000, s2!)
      toggle(12, s2?)
:
```

```
// ChibiOS
void thread1 () {
  while (1) {
    sleep(600);
    toggle(11);
  }
}
void thread2 () {
  while (1) {
    sleep(1000);
    toggle(12);
  }
}

void setup () {
  create(thread1);
  create(thread2);
}
```

```
// Ceu
par do
  loop do
    await 600ms;
    _toggle(11);
  end
with
  loop do
    await 1s;
    _toggle(12);
  end
end
```

Figure II.1: Two blinking LEDs in OCCAM-PI, ChibiOS and Céu.
Each line of execution in parallel blinks a LED with a fixed (but different) frequency. (The LEDs are connected to I/O ports 11 and 12.) Every 3 seconds both LEDs should light on together. After a couple of minutes of execution, only the implementation in Céu remains synchronized.

the two asynchronous implementations lost synchronism after some time of execution. The Céu implementation remained synchronized for all tests that we have performed.

The implementations are intentionally naive: they just spawn the activities to blink the LEDs in parallel. The behavior for the asynchronous implementations of the blinking application is perfectly valid, as the preemptive execution model does not ensure implicit synchronization among activities. In a synchronous language, however, the behavior must be predictable, and loosing synchronism is impossible by design. We used timers in this application, but any kind of high frequency input would also behave nondeterministically in asynchronous systems.

Note that even though the implementations are syntactically similar, with two endless loops in parallel, the underlying execution models between Céu and the two others are antagonistic, hence, the different execution behavior.

Although this application can be implemented correctly with an asynchronous execution model, it circumvents the language style, as timers need to be synchronized in a single thread. Furthermore, it is common to see similar naive blinking examples in reference examples of asynchronous systems[3], suggesting that LEDs are really supposed to blink synchronized, a guarantee that the language cannot provide.

---

[3] Example 1 in the RTOS *DuinOS v0.3*: `http://code.google.com/p/duinos/`. Example 3 in the occam-based *Concurrency for Arduino v20110201.1855*: `http://`

II.2 Synchronous model

In this section, we present a review of some synchronous languages and programming techniques that more closely relate to CÉU. We refer back to them in detail in Chapter VIII to discuss specific features and differences that require a deeper knowledge about CÉU.

Event-driven programming

At the lowest abstract level of the synchronous model, event-driven programming is usually employed as a technique in general-purpose languages with no specific support for reactivity. Because a single line of execution and stack are available, programmers need to deal with the burden of manual stack management and inversion of control. [1]

In the context of WSNs, the programming language *nesC* [19] offers event-driven programming for the TinyOS operating system. The concurrency model of *nesC* is very flexible, supporting the traditional serialization among callbacks (the default and recommended behavior), and also asynchronous callbacks that interrupt others. To deal with race conditions, *nesC* supports atomic sections with a similar semantics to mutual exclusion in asynchronous languages. We use *nesC* as the back end of CÉU for TinyOS [23].

Cooperative multithreading

Cooperative multithreading is an alternative approach to preemptive multithreading where the programmer is responsible for scheduling activities in the program (known as *coroutines* [34]). With this approach, there are no possible race conditions on global variables, as the points that transfer control in coroutines are explicit (and, supposedly, are never inside critical sections).

In the context of WSNs, Protothreads [14] offer very lightweight cooperative multithreading for embedded systems. Its stackless implementation reduces memory consumption but precludes support for local variables. Furthermore, Protothreads provide no static safety warranties: programs can loop indefinitely, and accesses to globals are unrestricted.

Finite state machines

The use of finite state machines (FSMs) is a classic technique to implement reactive applications, such as network protocols and graphical user interfaces. A contemporary work [28] targets WSNs and is based on the Statecharts formalism [22].

`concurrency.cc/`.

FSMs have some known limitations. For instance, writing purely sequential flow is tedious, requiring to break programs in multiple states with a single transition connecting each of them. Another inherent problem of FSMs is the state explosion phenomenon, which can be alleviated in some designs that support hierarchical FSMs running in parallel [28].

Synchronous languages

The family of reactive synchronous languages[4] is an established alternative to *C* in the field of safety-critical embedded systems [3]. Two major styles of synchronous languages have evolved: in the *control–imperative* style (e.g. Esterel [8]), programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style (e.g. Lustre [21]), programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming.

Céu is strongly influenced by Esterel [10] in its support for hierarchical compositions of activities and reactivity to events. However, some fundamental differences exist, and we discuss them in detail in Section III.7.

II.3 Programming models in WSNs

A WSN application has to handle a multitude of concurrent events, such as timers and packet transmissions, keeping track of them according to its specification. From a control perspective, programs are composed of two main patterns: *sequential*, i.e., an activity with two or more sub-activities in sequence; and *parallel*, i.e., unrelated activities that eventually need to synchronize. As an example, an application that alternates between sampling a sensor and broadcasting its readings has a sequential pattern (with an enclosing loop); while using a 1-minute timeout to interrupt an activity denotes a parallel pattern.

Figure II.2 presents the three synchronous programming (sub-)models commonly used in WSNs through a simple concurrent application. It shows the implementations in *nesC* [19], *Protothreads* [14], and Céu for an application that continuously turns on a LED for 2 seconds and off for 1 second. After 1 minute of activity, the application turns off the LED and proceeds to another activity (marked in the code as `<...>`). The diagram on the bottom-right of Figure II.2 describes the overall control behavior for the application. The sequential programming pattern is represented by the LED alternating between the two states, while the parallel pattern is represented by the 1-minute timeout.

---

[4] The term *synchronous* is convoluted here: *Synchronous languages* evidently follow the *synchronous programming model*, but multi-purpose languages (e.g., *Java* and *C*) can also behave synchronously by applying techniques such as event-driven programming and state machines.

The first implementation, in *nesC*, which represents the *event-driven* model, spawns two timers "in parallel" at boot time (`Boot.booted`): one to make the LED blink and another to wait for 1 minute. The callback `T1.fired` continuously toggles the LED and resets the timer according to the state variable `on`. The callback `T2.fired` executes only once, canceling the blinking timer, and proceeds to `<...>`. Overall, we argue that this implementation has little structure: the blinking loop is not explicit, but instead relies on a static state variable and multiple invocations of the same callback. Furthermore, the timeout handler (`T2.fired`) requires specific knowledge about how to stop the blinking activity, and the programmer must manually terminate it (`T1.cancel()`).

The second implementation, in *Protothreads*, which represents the *multi-threaded* model [14, 9], uses a dedicated thread to make the LED blink in a loop. This brings more structure to the solution. The main thread also helps a reader to identify the overall sequence of the program, which is not easily identifiable in the event-driven implementation without tracking the dependencies among callbacks. However, it still requires much bookkeeping for initializing, scheduling and rejoining the blinking thread after the timeout (inside the `while` condition).

The third implementation, in CÉU, which represents the *synchronous-language model*, uses a `par/or` construct to run the two activities in parallel: an endless loop to blink the LED, and a single statement that waits for 1 minute before terminating. The `par/or` stands for *parallel-or* and rejoins automatically when any of its trails terminates. We argue that the hierarchical structure of CÉU more closely reflects the control diagram and ties the two activities together, implying that (a) they can only exist together; (b) they always start together (c) they always terminate together. Besides the arguably cleaner syntax, the additional control-flow information that can be inferred from the program is the base for all features and safety guarantees introduced by CÉU.

/ nesC */*

```
event void Boot.booted() {
  call T1.start(0)
  call T2.start(60000)
}
event void T1.fired() {
  static int on = 0;
  if (on) {
    call Leds.led0Off();
    call T1.start(1000);
  } else {
    call Leds.led0On();
    call T1.start(2000);
  }
  on = !on
}
event void T2.fired() {
  call T1.cancel();
  call Leds.led0Off();
  <...> // CONTINUE
}
```

/ Protothreads */*

```
int main() {
  PT_INIT(&blink);
  timer_set(&timeout,60000);
  while (
    PT_SCHEDULE(blink()) &&
    !timer_expired(timeout)
  );
  leds_off(LEDS_RED);
  <...> // CONTINUE
}
PT_THREAD blink() {
  while (1) {
    leds_on(LEDS_RED);
    timer_set(&timer,2000);
    PT_WAIT(expired(&timer));
    leds_off(LEDS_RED);
    timer_set(&timer,1000);
    PT_WAIT(expired(&timer));
  }
}
```

/* CÉU */

```
par/or do
   loop do
      _Leds_led0On();
      await 2s;
      _Leds_led0Off();
      await 1s;
   end
with
   await 1min;
end
_Leds_led0Off();
<...> // CONTINUE
```

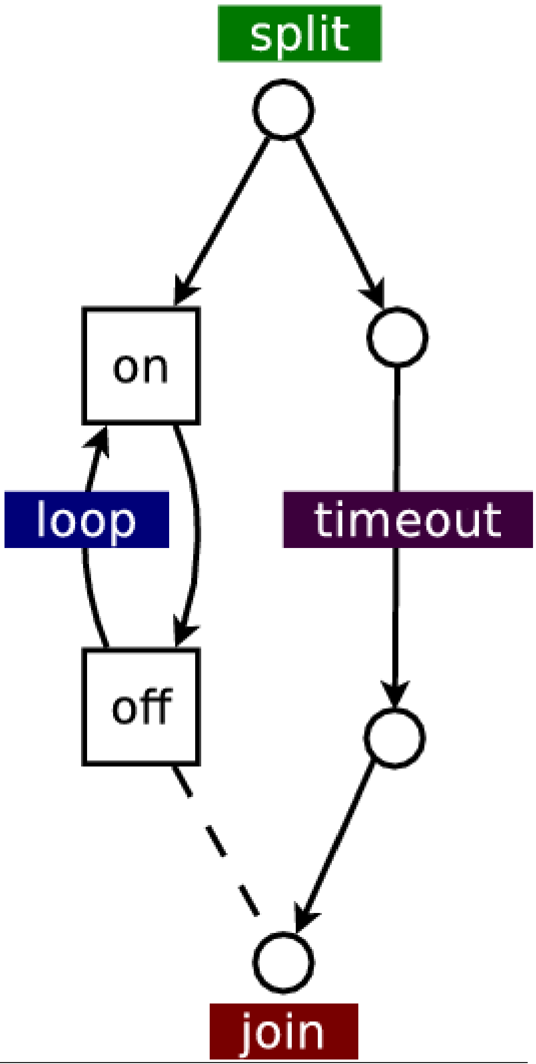

Figure II.2: “Blinking LED” in nesC, Protothreads, and CÉU.

III
The design of 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 an occurring event to all active trails, which share a single global time reference (the event itself). The fundamental distinction between CÉU and prevailing multi-threaded designs is the way threads are combined in programs. CÉU provides Esterel-like syntactic hierarchical compositions, while most multi-threaded systems typically only support top-level definitions for threads. Figure III.1 shows a compact reference of CÉU, which helps to follow the examples in this chapter.

We start the chapter with the fundamental design decisions behind CÉU's execution model, namely the uniqueness of external events and deterministic scheduler (Section III.1). Then, we discuss how they enable safe concurrency support for shared memory and *C* calls (Sections III.2 and III.3). We further introduce some new programming features that match CÉU's synchronous and safety-oriented design: local scope finalization (Section III.4), first-class timers (Section III.5), and a stack-based communication mechanism (Section III.6). We finish with a discussion that summarizes the chapter by comparing CÉU with Esterel (Section III.7).

III.1 The execution model of Céu

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 reactions to events are guaranteed to execute in bounded time (to be discussed further). The execution model for CÉU programs is as follows:

1. The program initiates the "boot reaction" 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.

```
// DECLARATIONS
input <type> <id>;      // external event
event <type> <id>;      // internal event
var   <type> <id>;      // variable

// EVENT HANDLING
await <id>;             // awaits event
emit  <id>;             // emits  event

// COMPOUND STATEMENTS
<...> ; <...> ;         // sequence
if <...> then <...>     // conditional
         else <...> end
loop do <...> end       // repetition
    break               //   (escape loop)
finalize <...>          // finalization
    with <...> end

// PARALLEL COMPOSITIONS
par/and do <...>        // rejoins on termination of both sides
      with <...> end
par/or do  <...>        // rejoins on termination of any side
      with <...> end
par do <...>            // never rejoins
  with <...> end

// C INTEGRATION
_f();                   // C call (prefix '_')
native do <...> end     // block of native code
pure <id>;              // pure annotation
safe <id> with <id>;    // safe annotation
```

Figure III.1: Syntax of Céu.

3. The program goes idle and the environment takes control.

4. On the occurrence of a new external input event, the environment awakes *all* trails awaiting that event. It then goes to step 2.

The synchronous execution model of Céu is based on the hypothesis that internal reactions run *infinitely faster* in comparison to the rate of external events [36]. Conceptually, a program takes no time on step 2 and is always idle on step 3. In practice, if a new external input event occurs while a reaction chain is running (step 2), it is enqueued to run in the next reaction. When multiple trails are active at a time (i.e. awaking on the same event), Céu schedules them in the order they appear in the program text. This policy is somewhat arbitrary, but provides a priority scheme for trails, and also ensures deterministic and reproducible execution for programs, which is important for simulation purposes. A reaction chain may also contain emissions and reactions to internal events, which are presented in Section III.6.

```
1  input void A, B, C;
2  par/and do
3      // trail 1
4      <...>         // <...> represents non—awaiting statements
5      await A;
6      <...>
7  with
8      // trail 2
9      <...>
10     await B;
11     <...>
12 with
13     // trail 3
14     <...>
15     await A;
16     <...>
17     await B;
18     par/and do
19         // trail 3
20         <...>
21     with
22         // trail 4
23         <...>
24     end
25 end
```

Figure III.2: A Céu program to illustrate the scheduler behavior.

To illustrate the behavior of the scheduler of Céu, the execution of the program in Figure III.2 is depicted in the diagram of Figure III.3. The program starts in the boot reaction and is split in three trails. Following the order of declaration, the scheduler first executes *trail 1* until it awaits A in line 5; then *trail 2* executes until it awaits B in line 10; then *trail 3* is scheduled and also awaits A, in line 15. As no other trails are pending, the reaction chain terminates and the scheduler remains idle until the occurrence of A: *trail 1* awakes, executes and terminates; and then *trail 3* executes and waits for B in line 17. *Trail 2* remains suspended, as it is not awaiting A. During this reaction, new instances of events A, B, and C occur and are enqueued to be handled in the reactions that follow. As A happened first, it is used in the next reaction. However, no trails are awaiting it, so an empty reaction chain takes place. The next reaction dequeues event B: *trail 2* awakes, executes and terminates; then *trail 3* is split in two and both terminate. The program terminates and does not react to the pending event C. Note that each step in the time line ($t0$, $t1$, etc.) is identified by the event it handles. Inside a reaction, trails only react to that identifying event (or remain suspended).

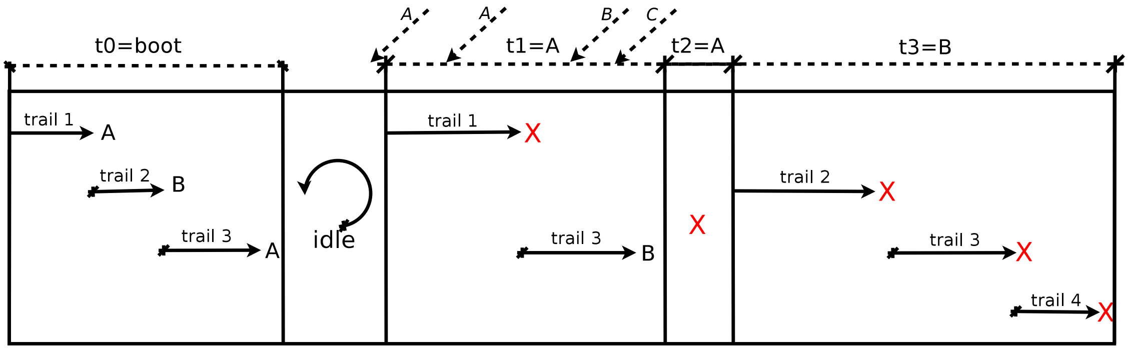

Figure III.3: A sequence of reaction chains for the program in Figure III.2.

(a) Bounded execution

Reaction chains should run in bounded time to guarantee that programs are responsive and can handle upcoming input events from the environment. 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
    if <cond> then
        break;
    end
end
```

```
loop do
    if <cond> then
        break;
    else
        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.

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

(b) Parallel compositions and abortion

The use of trails in parallel allows that programs wait for multiple events at the same time. Furthermore, trails await without loosing context information, such as locals and the program counter, what is a desired behavior in concurrent applications. [1]

Céu supports three kinds of parallel constructs regarding how they rejoin in the future: a `par/and` requires that all trails in parallel terminate before proceeding to the next statement; a `par/or` requires that any trail in parallel terminates before proceeding to the next statement, aborting all awaiting sibling trails; finally, a `par` never rejoins and should be used when trails in parallel are supposed to run forever (if they terminate, the scheduler forcedly halts them forever). To illustrate how trails rejoin, consider the two variations of the following archetype:

```
loop do
    par/and do
        <...>
    with
        await 100ms;
    end
end
```

```
loop do
    par/or do
        <...>
    with
        await 100ms;
    end
end
```

In the `par/and` variation, the block marked as `<...>` in the first trail (which may contain nested compositions with `await` statements) is repeated every 100 milliseconds at minimum, as both sides must terminate before re-executing the loop. In the `par/or` variation, if the block does not terminate within 100 milliseconds, it is restarted. These archetypes represent, respectively, the *sampling* and *timeout* patterns, which are very common in reactive applications.

The code in Figure III.4 is extracted from our implementation of the *CC2420* radio driver and uses a `par/or` to control the start/stop behavior of the radio. The input events `CC2420_START` and `CC2420_STOP` (line 1) represent the external interface of the driver with a client application (e.g. a protocol). The driver enters the top-level loop and awaits the starting event (line 3); once the client application makes a start request, the driver spawns two other trails: one to await the stopping event (line 5), and another to actually receive radio messages in a loop (collapsed in line 9). As compositions can be nested, the receiving loop can be as complex as needed, including other loops and parallel constructs. However, once the client requests to stop the driver, the trail in line 5 awakes and terminates, causing the `par/or` also terminate and abort the receiving loop. In this case, the top-level loop restarts and waits for the next request to start the radio (line 3, again).

The `par/or` construct of Céu is regarded as an *orthogonal preemption primitive* [7] because the two sides in the composition need not to be tweaked with synchronization primitives or state variables in order to affect each other. In contrast, it is known that traditional (asynchronous) multi-threaded languages cannot express thread abortion safely [7, 35].

```
 1  input void CC2420_START, CC2420_STOP;
 2  loop do
 3      await CC2420_START;
 4      par/or do
 5          await CC2420_STOP;
 6      with
 7          // loop with other nested trails
 8          // to receive radio packets
 9          <...>
10      end
11  end
```

Figure III.4: Start/stop behavior for the radio driver.
The occurrence of `CC2420_STOP` (line 5) seamlessly aborts the receiving loop (collapsed in line 9) and resets the driver to wait for the next `CC2420_START` (line 3).

(c) Reasoning about concurrency

The blinking LED of Figure II.2 in Céu illustrates how synchronous parallel constructs lead to a simpler reasoning about concurrency aspects in comparison to the other implementations. As reaction times are assumed to be instantaneous, the blinking loop takes exactly 3 seconds (i.e., $2s + 1s$). Hence, after 20 iterations, the accumulated time becomes 60 seconds and the loop terminates concurrently with the 1-minute timeout in parallel. Given that the loop appears first in the code, the scheduler will restart it and turn on the LED for the last time. Then, the 1-minute timeout is scheduled, aborts the whole `par/or`, and turns off the LED. This reasoning is actually reproducible in practice, and the LED will light on exactly 21 times for every single execution of this program. First-class timers are discussed in more detail in Section III.5. Note that this static control inference is impossible in asynchronous languages, given that internal reactions take an unpredictable time (as illustrated in Figure II.1). Even in the other implementations of Figure II.2, this control inference cannot be easily extracted, specially considering the presence of two different timers.

The behavior for the LED timeout just described denotes a *weak abortion*, because the blinking trail had the chance to execute for the last time. By inverting the two trails, the `par/or` would terminate immediately, and the blinking trail would not execute, denoting a *strong abortion* [7]. Céu not only provides means to choose between weak and strong abortion, but also detects the two conflicting possibilities and issues a warning at compile time (to be discussed in Section III.2).

III.2 Shared-memory concurrency

WSN applications make extensive use of shared memory, such as for handling memory pools, message queues, routing tables, etc. Hence, an important goal of Céu is to ensure a reliable execution for concurrent programs that share memory. Concurrency in Céu is characterized when two or more trail segments in parallel execute during the same reaction chain. A trail segment is a sequence of statements followed by an `await` (or termination).

In the first code fragment that follows, the two assignments to `x` run concurrently, because both trail segments are spawned during the same reaction chain. However, in the second code fragment, the assignments to `y` are never concurrent, because `A` and `B` are different external events and the respective segments can never execute during the same reaction chain:

```
var int x=1;
par/and do
    x = x + 1;
with
    x = x * 2;
end
```

```
input void A, B;
var int y=0;
par/and do
    await A;
    y = y + 1;
with
    await B;
    y = y * 2;
end
```

Note that although the variable `x` is accessed concurrently in the first example, the assignments are both atomic and deterministic: the final value of `x` is always 4 (i.e. $(1 + 1) * 2)$). Remember from Section III.1 that trails are scheduled in the order they appear and run to completion (i.e., until they await or terminate). However, programs with concurrent accesses to shared memory are suspicious, because an apparently innocuous reordering of trails modifies the semantics of the program; for instance, the previous example would yield 3 with the trails reordered, i.e., $(1 * 2 + 1)$.

We developed a compile-time temporal analysis for Céu in order to detect concurrent accesses to shared variables, as follows: *if a variable is written in a trail segment, then a concurrent trail segment 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. The algorithm for the analysis holds the set of all events in preceding `await` statements for each variable access. Then, the sets for all accesses in parallel trails are compared to assert that no events are shared among them. Otherwise the compiler warns about the suspicious accesses.

Consider the three examples in Figure III.5. The first code is detected

```
1  input void A;
2  var int x;
3  var int* p;
4  par/or do
5    loop do
6      await A;
7      if <cnd> then
8        break;
9      end
10   end
11   x = 1;
12 with
13   await A;
14   *p = 2;
15 end
```

```
input void A, B;
var int y;
par/or do
  await A;
  y = 1;
with
  await B;
  y = 2;
end
await A;
y = 3;
```

```
input void A;
var int z;
par/and do
  await A;
  z = 1;
with
  await A;
  await A;
  z = 2;
end
```

Figure III.5: Automatic detection for concurrent accesses to shared memory. The first example is suspicious because `x` and `p` can be accessed concurrently (lines 11 and 14). The second example is safe because accesses to `y` can only occur in sequence. The third example illustrates a false positive in our algorithm.

as suspicious, because the assignments to `x` and `p` (lines 11 and 14) may be concurrent in a reaction to `A` (lines 6 and 13); In the second code, although two of the assignments to `y` occur in reactions to `A` (lines 4-5 and 10-11), they are not in parallel trails and, hence, are safe. Note that the assignment in reaction to `B` (line 8) is safe given that reactions to different events cannot overlap (due to the single-event rule). The third code illustrates a false positive in our algorithm: the assignments to `z` in parallel can only occur in different reactions to `A` (lines 5 and 9), as the second assignment awaits two occurrences of `A`, while the first trail assigns and terminates in the first occurrence.

Conflicting weak and strong abortions, as introduced in Section III.1, are also detected with the proposed algorithm. Besides accesses to variables, the algorithm also keeps track of trail terminations inside a `par/or`, issuing a warning when they can occur concurrently. This way, the programmer can be aware about the conflict existence and choose between weak or strong abortion.

The proposed static analysis is only possible due to the uniqueness of external events within reactions and support for syntactic compositions, which provide precise information about the flow of trails (i.e., which run in parallel and which are guaranteed to be in sequence). Such precious information cannot be inferred when the program relies on state variables to handle control, as typically occurs in event-driven systems.

We also implemented an alternative algorithm that converts a CÉU program into a deterministic finite automata. The resulting DFA represents all possible points a program can reach during runtime and, hence, eliminates all

```
1  native do
2      #include <assert.h>
3      int I = 0;
4      int inc (int i) {
5          return I+i;
6      }
7  end
8  native _assert(), _inc(), _I;
9  _assert(_inc(_I));
```

Figure III.6: A Céu program with embedded *C* definitions. The globals `I` and `inc` are defined in the `native` block (lines 3 and 4), and are imported by Céu in line 8. *C* symbols must be prefixed with an underline to be used in Céu (line 9).

false positives. However, the algorithm is exponential and may be impractical in some situations. That being said, the simpler static analysis does not detect false positives in any of the implementations to be presented in Section V and executes in negligible time, suggesting that the algorithm is practical.

III.3 Integration with *C*

Most existing operating systems and libraries for WSNs are based on *C*, given its omnipresence and level of portability across embedded platforms. Therefore, it is fundamental that programs in Céu have access to all functionality the underlying platform already provides.

In Céu, any identifier prefixed with an underscore is repassed *as is* to the *C* compiler that generates the final binary. Therefore, access to *C* is seamless and, more importantly, easily trackable. Céu also supports *native blocks* to define new symbols in *C*, as Figure III.6 illustrates. Code inside "`native do ... end`" is also repassed to the *C* compiler for the final generation phase. As Céu mimics the type system of *C*, values can be easily passed back and forth between the languages.

C calls are fully integrated with the static analysis presented in Section III.2 and cannot appear in concurrent trails segments, because 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 increases considerably the number of false positives in the analysis, given that many functions can actually be safely called concurrently. Therefore, Céu supports explicit syntactic annotations to relax the policy. They are illustrated in Figure III.7, and are described as follows:

- The `pure` modifier declares a *C* function that does not cause side effects,

```
pure  _abs();                 // side-effect free
safe _Leds_led0Toggle with  // 'led0' vs 'led1' is safe
     _Leds_led1Toggle;
var int* buf1, buf2;        // point to different buffers
safe buf1 with buf2;        // 'buf1' vs 'buf2' is safe
```

Figure III.7: Annotations for *C* functions.
Function `abs` is side-effect free and can be concurrent with any other function. The functions `_Leds_led0Toggle` and `_Leds_led1Toggle` can execute concurrently. The variables `buf1` and `buf2` can be accessed concurrently (annotations are also applied to variables).

allowing it to be called concurrently with any other function in the program.

- The `safe` modifier declares a pair of variables or functions that do not affect each other, allowing them to be used concurrently.

Céu does not extend the bounded execution analysis to *C* function calls. On the one hand, *C* calls must be carefully analyzed in order to keep programs responsive. On the other hand, they also provide means to circumvent the rigor of Céu in a well-marked way (the special underscore syntax). Evidently, programs should only resort to *C* for simple operations that can be assumed to be instantaneous, such as non-blocking I/O and `struct` accessors, but never for control purposes.

III.4 Local scopes and finalization

Local declarations for variables bring definitions closer to their use in programs, increasing the readability and containment of code. Another benefit, specially in the context of WSNs, is that blocks in sequence can share the same memory space, as they can never be active at the same time. The syntactic compositions of trails allows the Céu compiler to statically allocate and optimize memory usage: memory for trails in parallel must coexist; trails that follow rejoin points reuse all memory.

However, the unrestricted use of locals may introduce subtle bugs when dealing with pointers and *C* functions interfacing with device drivers. Given that hardware components outlive the scope of any local variable, a pointer passed as parameter to a system call may be held by a device driver for longer than the scope of the referred variable, leading to a dangling pointer.

The code snippet in Figure III.8 was extracted from our implementation of the CTP collection protocol [39]. The protocol contains a complex control hierarchy in which the trail that sends beacon frames (lines 11-16) may be

```
 1 <...>
 2 par/or do
 3    <...>         // stops the protocol or radio
 4 with
 5    <...>         // neighbour request
 6 with
 7    loop do
 8       par/or do
 9          <...>   // resends request
10       with
11          await (dt) ms; // beacon timer expired
12          var _message_t msg;
13          payload = _AMSend_getPayload(&msg,...);
14          <prepare the message>
15          _AMSend_send(..., &msg, ...);
16          await CTP_ROUTE_RADIO_SENDDONE;
17       end
18    end
19 end
```

Figure III.8: Unsafe use of local references.
The period in which the radio driver manipulates the reference to `msg` passed by `_AMSend_send` (line 15) may outlive the lifetime of the variable scope, leading to an undefined behavior in the program.

aborted from multiple `par/or` trails (all collapsed in lines 3, 5, and 9). Now, consider the following behavior: The sending trail awakes from a beacon timer (line 11). The local message buffer (line 12) is prepared and passed to the radio driver (line 13-15). While waiting for an acknowledgment from the driver (line 16), the protocol receives a request to stop (line 3) that aborts the sending trail and makes the local buffer go out of scope. As the radio driver runs asynchronously and still holds the reference to the message (passed in line 15), it may manipulate the dangling pointer. A possible solution is to cancel the message send in all trails that can abort the sending trail (through a call to `AMSend_cancel`). However, this would require expanding the scope of the message buffer, adding a state variable to keep track of the sending status, and duplicating the code, increasing considerably the complexity of the application.

CÉU provides a safer and simpler solution with the following rule: *C calls that receive pointers require a finalization block to safely handle referred variables going out of scope.* This rule prevents the previous example to compile, forcing the relevant parts to be be rewritten as

```
1  native nohold _AMSend_getPayload();
2      <...>
3          var _message_t msg;
4          <...>
5          finalize
6              _AMSend_send(..., &msg, ...);
7          with
8              _AMSend_cancel(&msg);
9          end
10    <...>
```

First, the `nohold` annotation informs the compiler that the referred *C* function does not require finalization code because it does not hold references (line 1). Second, the `finalize` construct (lines 5-9) automatically executes the `with` clause (line 8) when the variable passed as parameter in the `finalize` clause (line 6) goes out of scope. Therefore, regardless of how the sending trail is aborted, the finalization code politely requests the OS to cancel the ongoing send operation (line 8).

All network protocols that we implemented in CÉU use this finalization mechanism for message sends. We looked through the TinyOS code base and realized that among the 349 calls to the `AMSend.send` interface, only 49 have corresponding `AMSend.cancel` calls. We verified that many of these *sends* should indeed have matching *cancels* because the component provides a *stop* interface for clients. In *nesC*, because message buffers are usually globals, a send that is not properly canceled typically results in an extra packet transmission that wastes battery. However, in the presence of dynamic message pools, a misbehaving program can change the contents of a (not freed) message that is actually about to be transmitted, leading to a subtle bug that is hard to track.

The finalization mechanism is fundamental to preserve the orthogonality of the `par/or` construct, i.e., an aborted trail does not require clean up code outside it.

III.5 First-class timers

Activities that involve reactions to *wall-clock time*[1] appear in typical patterns of WSNs, such as timeouts and sensor sampling. However, support for wall-clock time is somewhat low-level in existing languages, usually through timer callbacks or sleep blocking calls. In any concrete system implementation, however, a requested timeout does not expire precisely with zero-delay, a fact that is usually ignored in the development process. We define the difference between the requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in sequence (or in a loop) may accumulate a considerable amount of deltas that can lead to incorrect behavior in programs.

---

[1] By wall-clock time we mean the passage of time from the real world, measured in hours, minutes, etc.

The `await` statement of Céu supports wall-clock time and handles deltas automatically, resulting in more robust applications. As an example, consider the following program:

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

Suppose that after the first `await` request, the underlying system gets busy and takes 15ms to check for expiring awaits. The Céu scheduler will notice that the `await 10ms` has not only already expired, but delayed with `delta=5ms`. Then, the awaiting trail awakes, sets `v=1`, and invokes `await 1ms`. As the current delta is higher than the requested timeout (i.e. $5ms > 1ms$), the trail is rescheduled for execution, now with `delta=4ms`.

Céu also takes into account the fact that time is a physical quantity that can be added and compared. For instance, for the program that follows, although the scheduler cannot guarantee that the first trail terminates exactly in 11ms, it can at least ensure that the program always terminates with `v=1`:

```
par/or do
    await 10ms;
    <...>          // any non—awaiting sequence
    await  1ms;
    v = 1;
with
    await 12ms;
    v = 2;
end
```

Remember that any non-awaiting sequence is considered to take no time in the synchronous model. Hence, the first trail is guaranteed to terminate before the second trail, because $10 + 1 < 12$. A similar program in a language without first-class support for timers, would depend on the execution timings for the code marked as `<...>`, making the reasoning about the execution behavior more difficult. The importance of synchronized timers becomes more evident in the presence of loops, like in the introductory example of Figure II.2 in which the first trail is guaranteed to execute exactly 21 times before being aborted by the timer in the second trail.

Note that in extreme scenarios, small timers in sequence (or in a loop) may never "catch up" with the external clock, resulting in a *delta* that increases indefinitely. To deal with such cases, the current *delta* is always returned from an `await` and can be used in programs:

```
loop do
    var int late = await 1ms;
    if late < 1000 then
        <...>   // normal behavior
    else
        <...>   // abnormal behavior
    end
end
```

III.6 Internal events

Céu provides internal events as a signaling mechanism among parallel trails: a trail that invokes `await e` can be awoken in the future by a trail that invokes `emit e`.

In contrast with external events, which are handled in a queue, internal events follow a stack policy. In practical terms, this means that a trail that emits an internal event pauses until all trails awaiting that event completely react to it, continuing to execute afterwards. Another difference to external events is that internal events occur in the same reaction chain they are emitted, i.e., an `emit` instantaneously matches and awakes all corresponding `await` statements that were invoked in *previous reaction chains*[2].

The stacked execution for internal events introduces support for a restricted form of subroutines that cannot express recursive definitions (either directly or indirectly), resulting in bounded memory and execution time. Figure III.9 shows how the dissemination trail from our implementation of the DRIP protocol simulates a function and can be invoked from different parts of the program (lines 16-19), just like a subroutine. The `await send` (line 17) represents the function entry point, which is surrounded by a `loop` so that it can be invoked repeatedly. The DRIP protocol distinguishes data and metadata packets and disseminates one or the other based on a request parameter. For instance, when the trickle timer expires (line 8), the program invokes `emit send=>1` (line 9), which awakes the dissemination trail (line 17) and starts sending a metadata packet (collapsed in line 18). Note that if the trail is already sending a packet, then the `emit` will not match the `await` and will have no effect (the *nesC* implementation uses an explicit state variable to attain this same behavior).

---

[2] In order to ensure bounded reactions, an `await` statement cannot awake in the same

```
1   event int send;
2   par do
3       <...>
4           await DRIP_KEY;
5           emit send => 0;     // broadcast data
6   with
7       <...>
8           await DRIP_TRICKLE;
9           emit send => 1;     // broadcast meta
10  with
11      <...>
12          var _message_t* msg = await DRIP_DATA_RECEIVE;
13          <...>
14          emit send => 0;     // broadcast data
15  with
16      loop do
17          var int isMeta = await send;
18          <...>   // send data or metadata (contains awaits)
19      end
20  end
```

Figure III.9: A loop that awaits an internal event can emulate a subroutine. The `send` "subroutine" (lines 16-19) is invoked from three different parts of the program (lines 5, 9, and 14).

This form of subroutines has some significant limitations:

***Single instance*:** Calls to a running subroutine have no effect. As noted in the example of Figure III.9, a subroutine that awaits on its body may miss further calls to it (in some cases this behavior is actually desired).

***Single calling*:** Further calls to a subroutine in a reaction chain also have no effect. Even if a subroutine terminates within a reaction chain (i.e. reaches the `await` again), other `emit` invocations are ignored until the next reaction chain. Remember that *await* statements must be awaiting before the reaction chain starts to be awoken and that `emit` statements are immediately broadcast (i.e., they are not buffered).

***No recursion*:** Recursive calls to a subroutine also have no effect. For the same reason of the *single instance* property, a trail cannot be awaiting itself while running and the recursive call is ignored.

***No concurrency*:** If two trails in parallel try to call the same subroutine, the static analysis warns about non-determinism. Even if the warning is ignored, the call from the first trail takes effect (based on deterministic scheduling), while the second call fails on the *single call* property.

reaction chain it is invoked.

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

- The interaction with other Céu 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 recursive subroutines.

Internal events also provide means for describing more elaborate control structures, such as *exceptions*. The code in Figure III.10 handles incoming packets for the CC2420 radio driver in a loop. After awaking from a new packet notification (line 4), the program enters in a sequence to read the bytes from the hardware buffer (lines 8-16). If any anomaly is found on the received data, the program invokes `emit next` to discard the current packet (lines 10 and 14). Given the execution semantics of internal events, the `emit` continuation is stacked and awakes the trail in line 6, which terminates and aborts the whole `par/or` in which the emitting trail is paused. Therefore, the continuation for the `emit` never resumes, and the loop restarts to await a next packet.

III.7 Differences to Esterel

A primary goal of Céu is to support reliable shared-memory and *C* concurrency on top of a deterministic scheduler and effective safety analysis (Sections III.1, III.2 and III.3). Esterel, however, does not support shared-memory concurrency because *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [6]. Furthermore, Esterel is deterministic only with respect to reactive control, i.e., *"the same sequence of inputs always produces the same sequence of outputs"* [6]. However, the order of execution for side-effect operations within a reaction is non-deterministic: *"if there is no control dependency and no signal dependency, as in "`call f1() || call f2()`", the order is unspecified and it would be an error to rely on it"* [6].

In Esterel, an external reaction can carry simultaneous signals, while in Céu, a single event defines a reaction. The notion of time in Esterel is similar to that of digital circuits, in which multiple wires (signals) can be queried for

```
 1  <...>
 2  event void next;
 3  loop do
 4      await CC_RECV_FIFOP;
 5      par/or do
 6          await next;
 7      with
 8          <...>   // (contains awaits)
 9          if rxFrameLength > _MAC_PACKET_SIZE then
10              emit next;  // packet is too large
11          end
12          <...>   // (contains awaits)
13          if rxFrameLength == 0 then
14              emit next;  // packet is empty
15          end
16          <...>   // (contains awaits)
17      end
18  end
```

Figure III.10: Exception handling in Céu.
The `emit`'s in lines 10 and 14 raise an exception to be caught by the `await` in line 6. The `emit` continuations are discarded given that the surrounding `par/or` is aborted.

their status (*present* or *absent*) on each clock tick. Céu more closely reflects event-driven programming, in which occurring events are sequentially and uninterruptedly handled by the program. This design decision is fundamental for the temporal analysis of Section III.2.

Esterel makes no semantic distinctions between internal and external signals, both having only the notion of presence or absence during the entire reaction [7]. In Céu, however, 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 within a reaction.
- External events are handled in a queue, while internal events follow a stacked execution policy.

In particular, the stack-based execution policy for internal events in Céu enables a limited but safe form of subroutines and an exception-handling mechanism, as discussed in Section III.6.

Apart from these fundamental differences to Esterel, Céu introduces first-class timers with a convenient syntax and predictable behavior (Section III.5), and also finalization blocks to safely handle memory going out of scope (Section III.4).

IV
Demo applications

In this chapter, we present two demos that explore the high-level and safety capabilities of CÉU described in the previous chapter. Our goal is to present full commented applications that help understanding and getting familiar with the language. The applications are somewhat simple (70 and 170 lines), but complete enough to expose the programming techniques promoted by CÉU.

The first demo targets commercially available 16-bit WSN nodes, such as *micaZ* and *telosb*[1]. The second demo uses the Arduino open-source platform[2], in order to experiment with custom third-party hardware. Both platforms have low processing power and memory capacity (16Mhz CPU, 32Kb Flash, and 4Kb SRAM), showing that CÉU is applicable to highly constrained platforms.

IV.1 WSN ring

In the first demo, we implement a fixed-ring topology with N motes placed side-by-side which should all follow the same behavior: receive a message with an integer counter, show it on the LEDs, wait for 1 second, increment the counter, and forward it to the mote on its right. Because the topology constitutes a ring, the counter will be incremented forever while traversing the motes. If a mote does not receive a message within 5 seconds, it should blink the red LED every 500 milliseconds until a new message is received. The mote with *id=0* is responsible for initiating the process at boot time and recovering the ring from failures. On perceiving a failure, it should wait for 10 seconds before retrying the communication.

Figure IV.1 implements the communicating trail, which continuously receives and forwards the messages. The code is an endless loop that first awaits a radio message (line 2), gets a pointer to its data buffer (line 3), shows the received counter on the LEDs (line 4), and then awaits 1s (line 5) before

---

[1] `http://www.xbow.com`

[2] `http://arduino.cc`

```
1  loop do
2      var _message_t* msg = await RADIO_RECEIVE;
3      var int* cnt = _Radio_getPayload(msg);
4      _Leds_set(*cnt);
5      await 1s;
6      *cnt = *cnt + 1;
7      finalize
8          _Radio_send((_NODE_ID+1)%N, msg);
9      with
10         _Radio_cancel(msg);
11     end
12     await RADIO_SENDDONE;
13 end
```

Figure IV.1: Communicating trail for the WSN ring.

incrementing the counter in the message (line 6) and forwarding it to the next mote (line 7-12).

The program uses several services provided by the underlying operating system ([23]), which are all non-blocking *C* functions for LEDs and radio manipulation.

The finalization block (lines 7-11) ensures that regardless of how the communicating trail is composed with the rest of the application (and eventually aborted by it), the `msg` buffer will be safely released while waiting for a `RADIO_SENDDONE` acknowledge from the radio driver.

Because this code does not handle failures, it is straight to the point and easy to follow. Actually, this is the final code for this task, as error handling is placed in a parallel trail.

To handle failures, we define in Figure IV.2 a monitoring trail (lines 4-22) in parallel with the communicating trail. Lines 8 to 20 describe the network-down behavior. After 5 seconds of inactivity are detected in the sub-trails in parallel (lines 6 and 8), two new activities run in parallel: one that retries communication every 10 seconds by signaling the internal event `retry` (lines 8-11); and another that blinks the red LED every 500 milliseconds (lines 13-17).

The trick to restore the normal behavior of the network is to await event `RADIO_RECEIVE` (line 6) in the `par/or` (line 5) with the network-down behavior to abort it whenever a new message is received. By surrounding everything with a `loop` (line 4), we ensure that the error detection is continuous.

Finally, we implement in Figure IV.3 the initiating/retrying process that sends the first message from mote with *id=0*. Again, we place the code (lines 6-20) in parallel with the other activities. As this process is only handled by the mote with $id = 0$, we start by checking it (line 6). If this is not the case, we simply await forever on this trail (line 19). Otherwise, the `loop` (lines 7-17)

```
1  par do
2      <...> // COMMUNICATING TRAIL (previous code)
3  with
4      loop do
5          par/or do
6              await RADIO_RECEIVE;
7          with
8              await 5s;
9              par do
10                 loop do
11                     emit retry; // only captured by mote 0
12                     await 10s;
13                 end
14             with
15                 _Leds_set(0);   // clear LEDs
16                 loop do
17                     _Leds_led0Toggle();
18                     await 500ms;
19                 end
20             end
21         end
22     end
23 end
```

Figure IV.2: Monitoring trail for the WSN ring.

sends the first message as soon as the mote is turned on (line 12). It then waits for a `retry` emit (line 16) to loop and resend the initial message. Remind that event `retry` is emitted on network-down every 10 seconds (line 10 of previous code).

The static analysis of Céu correctly warns about concurrent calls to `_Radio_send` (line 12) *vs.* `_Leds_set` and `_Leds_led0Toggle` (lines 13,15 of previous code), which all execute after the program detects 5 seconds of inactivity (line 6 of previous code). However, because these functions affect different devices (i.e. radio *vs.* LEDs), they can be safely executed concurrently. The following annotation (to be included in the program) states that these specific functions can be called concurrently with deterministic behavior, allowing the program to be compiled without warnings:

```
safe _Radio_send with
     _Leds_set, _Leds_led0Toggle;
```

This example shows how complementary activities in an application can be written in separate and need not to be mixed in the code. In particular, error handling (monitoring trail) need not to interfere with regular behavior (communicating trail), and can even be incorporated later. To ensure that parallel activities exhibit deterministic behavior, the Céu compiler rejects harmful concurrent *C* calls by default.

```
par do
    <...> // COMMUNICATING TRAIL
with
    <...> // MONITORING TRAIL
with
    if _NODE_ID == 0 then
        loop do
            var _message_t msg;
            var int* cnt = _Radio_getPayload(&msg);
            *cnt = 1;
            finalize
                _Radio_send(1, &msg);
            with
                _Radio_cancel(&msg);
            end
            await retry;
        end
    else
        await FOREVER;
    end
end
```

Figure IV.3: Retrying trail for the WSN ring.

As a final consideration, we can extend the idea of compositions by combining different *applications* together. In the context of WSNs, it is usually difficult to physically recover motes in a deployed network, and by combining multiple applications in a single image, we can switch their execution remotely via radio. The archetype in Figure IV.4 illustrates this idea. The input event `SWITCH` (line 1) is used to request application switches remotely.[3] Initially, the code behaves as application 1 (lines 7-9), but is also waiting for a `SWITCH` request in parallel (line 5). Whenever a new request occurs, the `par/or` terminates, aborts the running application, and restarts as the requested application. The `await FOREVER` statement (line 13) ensures that a terminating application does not reach the end of the `par/or` and restarts itself.

The same idea can be used to *reboot* a mote remotely, in the case of a strange behavior in an application.

IV.2 Spaceship game

In the next demo, a spaceship game, we control a ship that moves through space and has to avoid collisions with meteors until it reaches the finish line. Although this application is not networked, it is still embedded and reactive, using timers, buttons, and an LCD with real-time feedback. We use an Arduino

---

[3] We are assuming the existence of an hypothetical high-level event `SWITCH` that abstracts the radio protocol for requests to change the current running application.

```
input int SWITCH;
var int cur_app = 1;
loop do
    par/or do
        cur_app = await SWITCH;
    with
        if cur_app == 1 then
            <...>  // CODE for APP1
        end
        if cur_app == 2 then
            <...>  // CODE for APP2
        end
        await FOREVER;
    end
end
```

Figure IV.4: Retrying trail for the WSN ring.

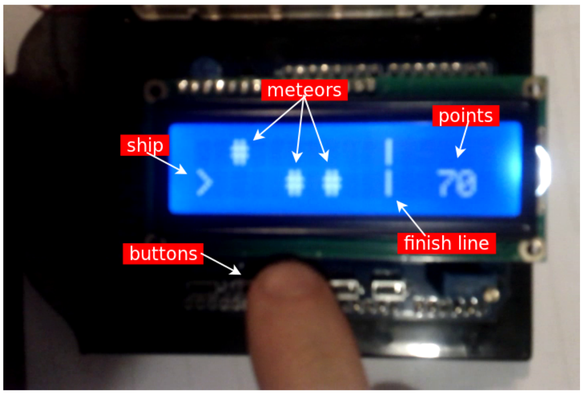

Figure IV.5: The “spaceship” game

connected to a third-party two-row LCD display with two buttons to exhibit and control the spaceship. Figure IV.5 shows the picture of a running quest.

We describe the behavior of the game, along with its implementation, following a top-down approach. The outermost loop of the game, in Figure IV.6, is constituted of `CODE 1`, which sets the game attributes such as globals `code` and `dt`; `CODE 2` with the central game loop; and `CODE 3` with the “game over” animation. Every time the loop is executed, it resets the game attributes (line 5), generates a new map (line 7), redraws it on screen (line 8), and waits for a starting key (line 9). Then, the program executes the main logic of the game (line 11), until the spaceship reaches the finish line or collides with a meteor. Depending on the result held in `win`, the “game over” code (line 13) may display an animation before restarting the game.

The game attributes (`CODE 1` in Figure IV.7) change depending on the result of the previous iteration of the outermost loop. For the first game execution and whenever the spaceship collides with a meteor, variable `win` is

```
var int dt;       // inverse of game speed
var int points;   // number of steps alive
var int win = 0;  // starting the game

loop do
    <...> // CODE 1: set game attributes

    _map_generate();
    _redraw(x, y, points);
    await KEY;  // starting key

    <...> // CODE 2: the central loop

    <...> // CODE 3: game over
end
```

Figure IV.6: Outermost loop for the game.

```
1      // CODE 1: set game attributes
2      var int y = 0;          // ship coordinates
3      var int x = 0;          //  restart every phase
4
5      if not win then
6          dt     = 500;
7          points = 0;
8      else
9          if dt > 100 then
10             dt = dt - 50;
11         end
12     end
```

Figure IV.7: Sets the game attributes.

false, hence, the attributes are reset to their initial values (lines 6-7) Otherwise, if the player reached the finish line, then the game gets faster, keeping the current points (lines 9-11).

The central loop of the game (`CODE 2` in Figure IV.8) moves the spaceship as time elapses and checks whether the spaceship reaches the finish line or collides with a meteor. The code is actually split in two loops in parallel: one that runs the game steps (lines 3-19), and the other that handles input from the player to move the spaceship (lines 21-29). Note that we want the spaceship to move only during the game action, this is why we did not place the input handling in parallel with the whole application.

The game steps run periodically, depending on the current speed of the game (line 4). For each loop iteration, `x` is incremented and the current state is redrawn on screen (lines 5-6). Then, the spaceship is checked for collision with meteors (lines 8-11), and also with the finish line (lines 13-16). In either

```
1  // CODE 2: the central loop
2  par/or do
3      loop do
4          await (dt)ms;
5          x = x + 1;
6          _redraw(x, y, points);
7  
8          if _MAP[y][x] == '#' then
9              win = 0;  // a collision
10             break;
11         end
12 
13         if x == _FINISH then
14             win = 1;  // finish line
15             break;
16         end
17 
18         points = points + 1;
19     end
20 with
21     loop do
22         var int key = await KEY;
23         if key == _KEY_UP then
24             y = 0;
25         end
26         if key == _KEY_DOWN then
27             y = 1;
28         end
29     end
30 end;
```

Figure IV.8: The game central loop.

of the cases, the central loop terminates with `win` set to the proper value, also canceling the input handling activity. The points are incremented before each iteration of the loop (line 18).

To handle input events, we wait for key presses in another loop (line 22) and change the spaceship position accordingly (lines 24, 27). Note that there are no possible race conditions on variable `y` (i.e., lines 6,8 *vs.* 24,27) because the two loops in the `par/or` statement react to different events (i.e., time and key presses).

After escaping the central loop, we run the code of Figure IV.9 for the "game over" behavior, which starts an animation if the spaceship collides with a meteor. The animation loop (lines 6-13) continuously displays the spaceship in the two directions, suggesting that it has hit a meteor. The animation is interrupted when the player presses a key (line 3), proceeding to the game restart. Note the use of the `_lcd` object, available in a third-party *C++* library shipped with the LCD display, and used unmodified in the example.

This demo makes extensive use of global variables, relying on the de-

```
1   // CODE 3: game over
2   par/or do
3       await KEY;
4   with
5       if !win then
6           loop do
7               await 100ms;
8               _lcd.setCursor(0, y);
9               _lcd.write(’<’);
10              await 100ms;
11              _lcd.setCursor(0, y);
12              _lcd.write(’>’);
13          end
14      end
15  end
```

Figure IV.9: The “game over” behavior for the game.

terministic concurrency analysis guaranteed by the CÉU compiler. We used a top-down approach to illustrate the hierarchical compositions of blocks of code. For instance, the “game over” animation (lines 6-13) is self-contained and can be easily adapted to a new behavior without considering the other parts of the program.

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, and event-handling responsiveness 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 and 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/`

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 CC2420 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, as they do not represent control functionality and pose no challenges regarding concurrency aspects (i.e. they are basically predicates, `struct` accessors, etc.).

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 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].

---

[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.

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; 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.

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.

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, 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 $6ms$ 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 and battery consumption, the experiment suggests that the overhead of Céu over

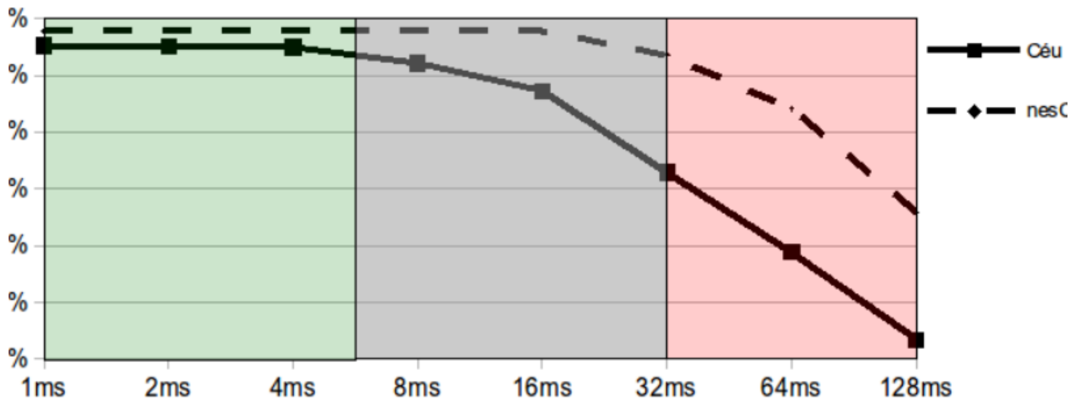

Figure V.2: Percentage of received packets depending on the duration of the lengthy operation.

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).

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 or battery consumption overhead at all, as the generated code is the same for Céu and *nesC*.

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 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

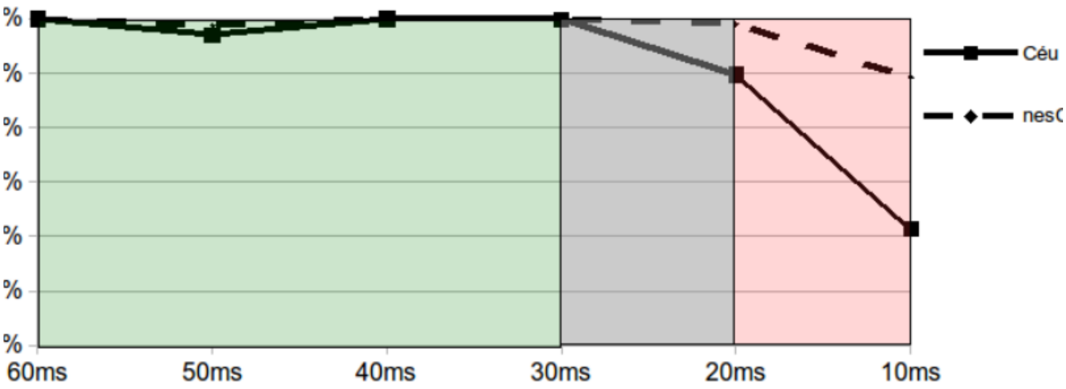

Figure V.3: Percentage of received packets depending on the sending frequency. Each received packet is tied to a 8-ms operation. Céu is 100% responsive up to a frequency of 30ms per packet.

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 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.

VI
The semantics of Céu

The disciplined synchronous execution of Céu, together with broadcast communication and stacked execution for internal events, may raise doubts about the precise execution of programs. In this chapter, we introduce a reduced syntax of Céu and propose an operational semantics in order to formally describe the language, eliminating imprecisions with regard to how a program reacts to an external event. For the sake of simplicity, we focus on the control aspects of the language, leaving out side effects and *C* calls (which behave like in any conventional imperative language).

VI.1 Abstract syntax

```
                                // primary expressions
p ::= mem(id)                   (any memory access to ‘id’)
    | await(id)                 (await event ‘id’)
    | emit(id)                  (emit event ‘id’)
    | break                     (loop escape)
                                // compound expressions
    | if mem(id) then p else p  (conditional)
    | p ; p                     (sequence)
    | loop p                    (repetition)
    | p and p                   (par/and)
    | p or p                    (par/or)
    | fin p                     (finalization)
                                // derived by semantic rules
    | awaiting(id,n)            (awaiting ‘id’ since sequence number ‘n’)
    | emitting(n)               (emitting on stack level ‘n’)
    | p @ loop p                (unwinded loop)
```

Figure VI.1: Reduced syntax of Céu.

Figure VI.1 shows the BNF-like syntax for a subset of Céu that is sufficient to describe all semantic peculiarities of the language. The $mem(id)$ primitive represents all accesses, assignments, and *C* function calls that affect a memory location identified by *id*. 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. The special notation *nop* is used to represent an innocuous *mem* expression (it can be though as a synonym for $mem(\epsilon)$, where ϵ is an unused identifier). Except for the fin and semantic-derived expressions, which are discussed further, the other expressions map to their counterparts in the concrete language in Figure III.1. Note that *mem* expressions cannot share identifiers with *await*/*emit* expressions.

VI.2 Operational semantics

The core of our semantics is a relation that, given a sequence number n identifying the current reaction chain, maps a program p and a stack of events S in a single step to a modified program and stack:

$$\langle S, p \rangle \xrightarrow[n]{} \langle S', p' \rangle$$

where

$$\begin{aligned} S, S' &\in id^* && (sequence\ of\ event\ identifiers: [id_{top}, ..., id_1]) \\ p, p' &\in P && (as\ described\ in\ Figure\ VI.1) \\ n &\in \mathbb{N} && (univocally\ identifies\ a\ reaction\ chain) \end{aligned}$$

At the beginning of a reaction chain, the stack is initialized with the occurring external event *ext* ($S = [ext]$), but *emit* expressions can push new events on top of it (we discuss how they are popped further).

We describe this relation with a set of *small-step* structural semantics rules, which are built in such a way that at most one transition is possible at any time, resulting in deterministic reaction chains. The transitions rules for the primary expressions are as follows:

$$\langle S,\ await(id) \rangle \xrightarrow[n]{} \langle S,\ awaiting(id, n) \rangle \qquad \textbf{(await)}$$

$$\langle id : S,\ awaiting(id, m) \rangle \xrightarrow[n]{} \langle id : S,\ nop \rangle, \ \ m < n \qquad \textbf{(awaiting)}$$

$$\langle S,\ emit(id) \rangle \xrightarrow[n]{} \langle id : S,\ emitting(|S|) \rangle \qquad \textbf{(emit)}$$

$$\langle S,\ emitting(|S|) \rangle \xrightarrow[n]{} \langle S,\ nop \rangle \qquad \textbf{(emitting)}$$

An *await* is simply transformed into an *awaiting* that remembers the current external sequence number n (rule **await**). An *awaiting* can only transit to a *nop* (rule **awaiting**) if its referred event *id* matches the top of the stack and its sequence number is smaller than the current one ($m < n$). An *emit* transits to an *emitting* holding the current stack level ($|S|$ stands for the stack length), and pushing the referred event on the stack (in rule **emit**). With the new stack level $|S|+1$, the *emitting*($|S|$) itself cannot transit, as rule **emitting** expects its parameter to match the current stack level. This trick provides the desired stack-based semantics for internal events.

Proceeding to compound expressions, the rules for conditionals and sequences are straightforward:

$$\frac{val(id,n) \neq 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, p\rangle} \quad \textbf{(if-true)}$$

$$\frac{val(id,n) = 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle} \quad \textbf{(if-false)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ ;\ q)\rangle} \quad \textbf{(seq-adv)}$$

$$\langle S, (mem(id)\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle \quad \textbf{(seq-nop)}$$

$$\langle S, (break\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, break\rangle \quad \textbf{(seq-brk)}$$

Given that our semantics focuses on control, rules **if-true** and **if-false** are the only to query *mem* expressions. The “magical” function *val* receives the memory identifier and current reaction sequence number, returning the current memory value. Although the value is arbitrary, it is unique in a reaction chain, because a given expression can execute only once within it (remember that *loops* must contain *awaits* which, from rule **await**, cannot awake in the same reaction they are reached).

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

$$\langle S, (loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, (p\ @\ loop\ p)\rangle \qquad \textbf{(loop-expd)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ @\ loop\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ @\ loop\ q)\rangle} \qquad \textbf{(loop-adv)}$$

$$\langle S, (mem(id)\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, loop\ p\rangle \qquad \textbf{(loop-nop)}$$

$$\langle S, (break\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, nop\rangle \qquad \textbf{(loop-brk)}$$

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

The rules for parallel *and* compositions force transitions on the left branch p to occur before transitions on the right branch q (rules **and-adv1** and **and-adv2**). Then, if one of the sides terminate, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**):

$$\begin{aligned}
isBlocked(n, a : S, awaiting(b, m)) &= (a \neq b \ \vee \ m = n) \\
isBlocked(n, S, emitting(s)) &= (|S| \neq s) \\
isBlocked(n, S, (p \ ; \ q)) &= isBlocked(n, S, p) \\
isBlocked(n, S, (p \ @ \ loop \ q)) &= isBlocked(n, S, p) \\
isBlocked(n, S, (p \ and \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q) \\
isBlocked(n, S, (p \ or \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q) \\
isBlocked(n, S, \_) &= false \ \ (mem, await, \\
&\qquad\qquad emit, break, if, loop)
\end{aligned}$$

Figure VI.2: The recursive predicate *isBlocked* is true only if all branches in parallel are hanged in *awaiting* or *emitting* expressions that cannot transit.

$$\frac{\langle S, p\rangle \ \xrightarrow[n]{} \ \langle S', p'\rangle}{\langle S, (p \ and \ q)\rangle \ \xrightarrow[n]{} \ \langle S', (p' \ and \ q)\rangle} \qquad \textbf{(and-adv1)}$$

$$\frac{isBlocked(n, S, p) \ , \ \ \langle S, q\rangle \ \xrightarrow[n]{} \ \langle S', q'\rangle}{\langle S, (p \ and \ q)\rangle \ \xrightarrow[n]{} \ \langle S', (p \ and \ q')\rangle} \qquad \textbf{(and-adv2)}$$

$$\langle S, (mem(id) \ and \ q)\rangle \ \xrightarrow[n]{} \ \langle S, q\rangle \qquad \textbf{(and-nop1)}$$

$$\langle S, (p \ and \ mem(id))\rangle \ \xrightarrow[n]{} \ \langle S, p\rangle \qquad \textbf{(and-nop2)}$$

$$\langle S, (break \ and \ q)\rangle \ \xrightarrow[n]{} \ \langle S, (clear(q) \ ; \ break)\rangle \qquad \textbf{(and-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p \ and \ break)\rangle \ \xrightarrow[n]{} \ \langle S, (clear(p) \ ; \ break)\rangle} \qquad \textbf{(and-brk2)}$$

The deterministic behavior of the semantics relies on the *isBlocked* predicate, defined in Figure VI.2 and used in rule **and-adv2**, requiring the left branch p to be blocked in order to allow the right transition from q to q'. An expression becomes blocked when all of its trails in parallel hang in *awaiting* and *emitting* expressions.

The last two rules **and-brk1** and **and-brk2** deal with a *break* in each of the sides in parallel. A *break* should terminate the whole composition in order

$$
\begin{aligned}
\mathit{clear}(\mathit{fin}\ p) &= p \\
\mathit{clear}(p\ ;\ q) &= \mathit{clear}(p) \\
\mathit{clear}(p\ @\ \mathit{loop}\ q)) &= \mathit{clear}(p) \\
\mathit{clear}(p\ \mathit{and}\ q) &= \mathit{clear}(p)\ ;\ \mathit{clear}(q) \\
\mathit{clear}(p\ \mathit{or}\ q) &= \mathit{clear}(p)\ ;\ \mathit{clear}(q) \\
\mathit{clear}(\_) &= \mathit{mem}(\mathit{id})
\end{aligned}
$$

Figure VI.3: The function *clear* extracts fin expressions in parallel and put their bodies in sequence.

to escape the innermost loop (*aborting* the other side). The *clear* function in the rules, defined in Figure VI.3, concatenates all active fin bodies of the side being aborted (to execute before the *and* rejoins). Note that there are no transition rules for fin expressions. This is because once reached, an fin expression only executes when it is aborted by a trail in parallel. In Section VI.3(c), we show how an fin is mapped to a finalization block in the concrete language. Note that there is a syntactic restriction that an fin body cannot *emit* or *await*—they are guaranteed to completely execute within a reaction chain.

Most rules for parallel *or* compositions are similar to *and* compositions:

$$
\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p\ \mathit{or}\ q)\rangle \xrightarrow[n]{} \langle S', (p'\ \mathit{or}\ q)\rangle} \qquad \textbf{(or-adv1)}
$$

$$
\frac{\mathit{isBlocked}(n, S, p)\ ,\ \ \langle S, q\rangle \xrightarrow[n]{} \langle S', q'\rangle}{\langle S(p\ \mathit{or}\ q)\rangle \xrightarrow[n]{} \langle S', (p\ \mathit{or}\ q')\rangle} \qquad \textbf{(or-adv2)}
$$

$$
\langle S, (\mathit{mem}(\mathit{id})\ \mathit{or}\ q)\rangle \xrightarrow[n]{} \langle S, \mathit{clear}(q)\rangle \qquad \textbf{(or-nop1)}
$$

$$
\frac{\mathit{isBlocked}(n, S, p)}{\langle S, (p\ \mathit{or}\ \mathit{mem}(\mathit{id}))\rangle \xrightarrow[n]{} \langle S, \mathit{clear}(p)\rangle} \qquad \textbf{(or-nop2)}
$$

$$
\langle S, (\mathit{break}\ \mathit{or}\ q) \xrightarrow[n]{} \langle S, (\mathit{clear}(q)\ ;\ \mathit{break})\rangle \qquad \textbf{(or-brk1)}
$$

$$
\frac{\mathit{isBlocked}(n, S, p)}{\langle S, (p\ \mathit{or}\ \mathit{break})\rangle \xrightarrow[n]{} \langle S, (\mathit{clear}(p)\ ;\ \mathit{break})\rangle} \qquad \textbf{(or-brk2)}
$$

For a parallel *or*, the rules **or-nop1** and **or-nop2** must terminate the composition, and also apply the function *clear* to the aborted side, in order to properly finalize it.

A reaction chain eventually blocks in *awaiting* and *emitting* expressions in parallel trails. If all trails hangs only in *awaiting* expressions, it means that the program cannot advance in the current reaction chain. However, *emitting* expressions should resume their continuations of previous *emit* in the ongoing reaction, they are just hanged in lower stack indexes (see rule **emit**). Therefore, we define another relation that behaves as the previous if the program is not blocked, and, otherwise, pops the stack:

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, p\rangle \underset{n}{\Longrightarrow} \langle S', p'\rangle} \qquad\qquad \frac{isBlocked(n,\ s : S,\ p)}{\langle s : S, p\rangle \underset{n}{\Longrightarrow} \langle S, p\rangle}$$

To describe a *reaction chain* in CÉU, i.e., how a program behaves in reaction to a single external event, we use the reflexive transitive closure of this relation:

$$\langle S, p\rangle \overset{*}{\underset{n}{\Longrightarrow}} \langle S', p'\rangle$$

Finally, to describe the complete execution of a program, we need multiple "invocations" of reaction chains, incrementing the sequence number:

$$\begin{gathered}\langle [e1], p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle \\ \langle [e2], p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle \\ \ldots\end{gathered}$$

Each invocation starts with an external event at the top of the stack and finishes with a modified program and an empty stack. After each invocation, the sequence number is incremented.

VI.3 Concrete language mapping

Although the reduced syntax presented in Figure VI.1 is similar to the concrete language in Figure III.1, there are some significant mismatches between CÉU and the formal semantics that require some clarification. In this section, we describe an informal mapping between the two.

Most statements from CÉU map directly to the formal semantics, e.g., `if` $\mapsto$ if, `';'` $\mapsto$ $';'$, `loop` $\mapsto$ *loop*, `par/and` $\mapsto$ *and*, `par/or` $\mapsto$ *or*. (Again, we are not considering side-effects, which are all mapped to the *mem* semantic construct.)

(a) await and emit

The `await` and `emit` primitives of CÉU are slightly more complex in comparison to the formal semantics, as they support communication of values between emits and awaits. In the two-step translation below, we start with the program in CÉU, which communicates the value 1 between the `emit` and `await` in parallel (left-most code). In the intermediate translation, we include the shared variable `e_` to hold the value being communicated between the two trails in order to simplify the `emit`. Finally, we convert the program into the equivalent in the formal semantics, translating side-effect statements into *mem* expressions:

```
par/or do
  <...>
  emit e => 1;
with
  v = await e;
  _printf("%d\n",v);
end
```

```
par/or do
  <...>
  e_ = 1;
  emit e;
with
  await e;
  v = e_;
  _printf("%d\n",v);
end
```

```
<...> ; mem ; emit(e)
          or
await(e) ; mem ; mem
```

Note that a similar translation is required for external events, i.e., each external event has a corresponding variable that is explicitly set by the environment before each reaction chain.

(b) First-class timers

To encompass first-class timers, we need a special `TICK` event that should be intercalated with each other event occurrence in an application:

$$\begin{aligned}
\langle [TICK], p\rangle &\overset{*}{\underset{1}{\Longrightarrow}} \langle [], p'\rangle \\
\langle [e1], p'\rangle &\overset{*}{\underset{2}{\Longrightarrow}} \langle [], p''\rangle \\
\langle [TICK], p''\rangle &\overset{*}{\underset{3}{\Longrightarrow}} \langle [], p'''\rangle \\
\langle [e2], p'''\rangle &\overset{*}{\underset{4}{\Longrightarrow}} \langle [], p''''\rangle \\
&\ldots
\end{aligned}$$

The `TICK` event has an associated variable `TICK_` (as illustrated in the previous section) with the time elapsed between the two occurrences of external events.

The translation in two steps from a timer await to the semantics is as follows:

```
dt = await 10ms;
```

```
var int tot = 10000;
loop do
    await TICK;
    tot = tot - TICK_;
    if tot <= 0 then
        dt = tot;
        break;
    end
end
```

```
mem;
loop(
    await(TICK);
    mem;
    if mem then
        mem;
        break
    else
        nop
)
```

(c) Finalization blocks

The biggest mismatch between Céu and the formal semantics is regarding finalization blocks, which require more complex modifications in the program for a proper mapping using the *fin* semantic construct. The code that follows uses a `finalize` to safely `_release` the reference to `ptr` kept after the call to `_hold`:

```
do
    var int* ptr = <...>;
    await A;
    finalize
        _hold(ptr);
    with
        _release(ptr);
    end
    await B;
end
```

In the translation to the semantics, the first required modification is to catch the `do-end` termination to run the finalization code. For this, we translate the block into a `par/or` with the original body in parallel with a *fin* to run the finalization code:

```
par/or do
    var int* ptr = <...>;
    await A;
    _hold(ptr);
    await B;
with
    { fin
        _release(ptr); }
end
```

In this intermediate code (mixing the syntaxes), the *fin* body will execute whenever the `par/or` terminates, either normally (after the `await B`) or aborted from an outer composition (rules **and-brk1**, **and-brk2**, **or-nop1**, **or-nop2**, **or-brk1**, and **or-brk2** in the semantics). However, the *fin* will also (incorrectly) execute even if the call to `_hold` is not reached in the body due to an abort before awaking from the `await A`. To deal with this issue, for each *fin* we need a corresponding flag to keep track of code that needs to be finalized:

```
1  f_ = 0;
2  par/or do
3      var int* ptr = <...>;
4      await A;
5      _hold(ptr);
6      f_ = 1;
7      await B;
8  with
9      { fin
10         if f_ then
11             _release(ptr);
12         end }
13 end
```

The flag is initially set to false (line 1), avoiding the finalization code to execute (lines 9-12). Only after the call to `_hold` (line 5) that we set the flag to true (line 6) and enable the *fin* body to execute. The complete translation from the original example in Céu is as follows:

```
mem;    // f_ = 0
(
   mem;          // ptr = <...>
   await(A);
   mem;          // _hold(ptr)
   mem;          // f_ = 1
   await(B);
or
   fin
     if mem then     // if f_
        mem          // release _ptr
     else
        nop
)
```

VII
The implementation of Céu

The compilation process of a program in Céu is composed of three main phases, as illustrated in Figure VII.1:

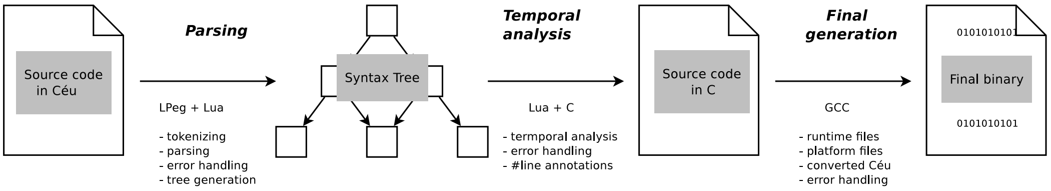
Source code in Céu
Parsing
LPeg + Lua
- tokenizing
- parsing
- error handling
- tree generation
Syntax Tree
Temporal analysis
Lua + C
- termporal analysis
- error handling
- #line annotations
Source code in C
Final generation
GCC
- runtime files
- platform files
- converted Céu
- error handling
0101010101
Final binary
0101010101

Figure VII.1: Compilation process: from the source code in Céu to the final binary.

Parsing The parser of Céu is written in *LPeg* [24], a pattern matching library that also recognize grammars, making it possible to write the tokenizer and grammar with the same tool. The source code is then converted to an *abstract syntax tree (AST)* to be used in further phases. This phase may be aborted due to syntax errors in the Céu source file.

Temporal analysis This phase detects inconsistencies in Céu programs, such as unbounded loops and the forms of non-determinism. It also makes some "classical" semantic analysis, such as building a symbol table for checking variable declarations. However, most of type checking is delayed to the last phase to take advantage of GCC's error handling. Therefore, this phase needs to annotate the *C* output with `#line` pragmas that match the original file in Céu. This phase must output code in *C*, given how tied Céu is to *C* by design.

Final generation The final phase packs the generated *C* file with the Céu runtime and platform-dependent functionality, compiling them with *gcc* and generating the final binary. The Céu runtime includes the scheduler, timer management, and the external *C* API. The platform files include libraries for I/O and bindings to invoke the Céu scheduler on external events.

In the sections that follow, we discuss the most sensible parts of the compiler considering our design, such as the temporal analysis, runtime scheduler, and the external API.

VII.1 Temporal analysis

As introduced, the *temporal analysis* phase detects inconsistencies in CÉU programs. Here, we focus on the algorithm that detects non-deterministic access to variables, as presented in Section III.2.

For each node representing a statement in the program AST, we keep the set of events I (for *incoming*) that can lead to the execution of the node, and also the set of events O (for *outgoing*) that can terminate the node.

A node inherits the set I from its direct parent and calculates O according to its type:

- Nodes that represent expressions, assignments, C calls, and declarations simply reproduce $O = I$, as they do not await;
- An `await e` statement has $O = \{e\}$.
- A `break` statement has $O = \{\}$ as it escapes the innermost `loop` and never terminate (see also `loop` below);
- A *sequence node (;)* modifies each of its children to have $I_n = O_{n-1}$. The first child inherits I from the sequence parent, and the set O for the sequence node is copied from its last child, i.e., $O = O_n$.
- A `loop` node includes its body's O on its own I ($I = I \cup O_{body}$), as the loop is also reached from its own body. The union of all `break` statements' O forms the set O for a `loop`.
- An `if` node has $O = O_{true} \cup O_{false}$.
- A parallel composition (`par/and` / `par/or`) may terminate from any of its branches, hence $O = O_1 \cup ... \cup O_n$.

With all sets calculated, any two nodes that perform side effects and are in parallel branches can have their I sets compared for intersections. If the intersection is not the empty set, they are marked as suspicious (see Section III.2).

Figure VII.2 reproduces the second code of Figure III.5 and shows the corresponding *AST* with the sets I and O for each node. The event `.` (dot) represents the "boot" reaction. The assignments to `y` in parallel (lines 5,8 in the code) have an empty intersection of I (lines 6,9 in the AST), hence, they do not conflict. Note that although the accesses in lines 5, 11 in the code (lines

6,11 in the AST) do have an intersection, they are not in parallel and are also safe.

```
input void A, B;
var int y;
par/or do
  await A;
  y = 1;
with
  await B;
  y = 2;
end
await A;
y = 3;
```

```
 1  Stmts I={.} O={A}
 2      Dcl_y I={.} O={.}
 3      ParOr I={.} O={A,B}
 4          Stmts I={.} O={A}
 5              Await_A I={.} O={A}
 6              Set_y I={A} O={A}
 7          Stmts I={.} O={B}
 8              Await_B I={.} O={B}
 9              Set_y I={B} O={B}
10      Await_A I={A,B} O={A}
11      Set_y I={A} O={A}
```

Figure VII.2: A program with a corresponding AST describing the sets I and O. The program is safe because accesses to `y` in parallel have no intersections for I.

VII.2 Memory layout

Céu favors a fine-grained use of trails, being common the use of trails that await a single event. For this reason, Céu does not allocate per-trail stacks; 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. Memory for trails in parallel must coexist, while statements in sequence can reuse it. Céu reserves a single static block of memory to hold all memory slots, whose size is the maximum the program uses at a given time. A given position in the memory may hold different data (with variable sizes) during runtime.

Translating this idea to C is straightforward [28, 5]: memory for blocks in sequence are packed in a `struct`, while blocks in parallel, in a `union`. As an example, Figure VII.3 shows a program with corresponding memory layout. Each variable is assigned a unique *id* (e.g. `a_1`) so that variables with the same name can be distinguished. The `do-end` blocks in sequence are packed in a `union`, given that their variables cannot be in scope at the same time, e.g., `MEM.a_1` and `MEM.b_2` can safely share the same memory address. The example also illustrates the presence of runtime flags related to the parallel composition, which also reside in reusable slots in the static memory.

VII.3 Trail allocation

The compiler extracts the maximum number of trails a program can have at the same time and creates a static vector to hold runtime information about

```
input int A, B, C;
do
    var int a = await A;
end
do
    var int b = await B;
end
par/and do
    await B;
with
    await C;
end
```

```
union {             // sequence
    int a_1;        //   do_1
    int b_2;        //   do_2
    struct {        //   par/and
        u8 _and_3: 1;
        u8 _and_4: 1;
    };
} MEM ;
```

Figure VII.3: A program with blocks in sequence and in parallel, with corresponding memory layout.

them. Again, trails that cannot be active at the same time can share memory slots in the static vector.

At any given moment, a trail can be awaiting in one of the following states: `INACTIVE`, `STACKED`, `FIN`, or in any event defined in the program:

```
enum {
    INACTIVE = 0,
    STACKED,
    FIN,
    EVT_A,      // input void A;
    EVT_e,      // event int e;
    <...>       // other events
}
```

All terminated or not-yet-started trails stay in the `INACTIVE` state and are ignored by the scheduler. A `STACKED` trail holds its associated stack level and is delayed until the scheduler runtime level reaches that value again. A `FIN` trail represents a hanged finalization block which is only scheduled when its corresponding block goes out of scope. A trail waiting for an event stays in the state of the corresponding event, also holding the sequence number (*seqno*) in which it started awaiting. A trail is represented by the following `struct`:

```
struct trail_t {
    state_t evt;
    label_t lbl;
    union {
        unsigned char seqno;
        stack_t       stk;
    };
};
```

The field `evt` holds the state of the trail (or the event it is awaiting); the field `lbl` holds the entry point in the code to execute when the trail is

```
1   input void A;
2   event void e;
3   // TRAIL 0 — lbl Main
4   par/and do
5     // TRAIL 0 — lbl Main
6     await e;
7     // TRAIL 0 — lbl Awake_e
8     // TRAIL 0 — lbl ParAnd_chk
9   with
10    // TRAIL 1 — lbl ParAnd_sub_2
11    await A;
12    // TRAIL 1 — lbl Awake_A_1
13    emit e;
14    // TRAIL 1 — lbl Emit_e_cont
15    // TRAIL 1 — lbl ParAnd_chk
16  end
17  // TRAIL 0 — lbl ParAnd_out
18  await A;
19  // TRAIL 0 — lbl Awake_A_2
```

```
enum {
  Main = 1,       // ln  3
  Awake_e,        // ln  7
  ParAnd_chk,     // ln  8, 15
  ParAnd_sub_2,   // ln 10
  Awake_A_1,      // ln 12
  Emit_e_cont,    // ln 14
  ParAnd_out,     // ln 17
  Awake_A_2       // ln 19
};
```

Figure VII.4: Static allocation of trails and entry-point labels.

scheduled; the third field depends on the `evt` field and may hold the `seqno` for an event, or the stack level `stk` for a `STACKED` state.

The size of `state_t` depends on the number of events in the application; for an application with less than 253 events (plus the 3 states), one byte is enough. The size of `label_t` depends primarily on the number of `await` statements in the application—each `await` splits the code in two and requires a unique entry point in the code for its continuation. Additionally, split & join points for parallel compositions, `emit` continuations, and finalization blocks also require labels. The `seqno` will eventually overflow during execution (every 256 reactions). However, given that the scheduler traverses all trails in each reaction, it can adjust them to properly handle overflows (actually 2 bits to hold the `seqno` would be already enough). The stack size depends on the maximum depth of nested emissions and is bounded to the maximum number of trails, e.g., a trail emits an event that awakes another trail, which emits an event that awakes another trail, and so on—the last trail cannot awake any trail, because they will be all hanged in a `STACKED` state. In WSNs applications, the size of `trail_t` is typically only 3 bytes (1 byte for each field).

(a) Code generation

The example in Figure VII.4 illustrates how trails and labels are statically allocated in a program. The program has a maximum of 2 trails, because the `par/and` (line 4) can reuse *TRAIL 0*, and the join point (line 16) can reuse both *TRAIL 0* and *TRAIL 1*. Each label is associated with a unique identifier

```
1  while (<...>) {             // scheduler main loop
2     trail_t* trail = <...>   // choose next trail
3     switch (trail->lbl) {
4        case Main:
5           // activate TRAIL 1 to run next
6           TRLS[1].evt = STACKED;
7           TRLS[1].lbl = ParAnd_sub_2;  // 2nd trail of par/and
8           TRLS[1].stk = current_stack;
9
10          // code in the 1st trail of par/and
11          // await e;
12          TRLS[0].evt = EVT_e;
13          TRLS[0].lbl = Awake_e;
14          TRLS[0].seq = current_seqno;
15          break;
16
17       case ParAnd_sub_2:
18          // await A;
19          TRLS[1].evt = EVT_A;
20          TRLS[1].lbl = Awake_A_1;
21          TRLS[1].seq = current_seqno;
22             break;
23
24         <...>   // other labels
25    }
26 }
```

Figure VII.5: Generated code for the program of Figure VII.4.

in the `enum`. The static vector to hold the two trails in the example is defined as

```
trail_t TRLS[2];
```

In the final generated *C* code, each label becomes a *switch case* working as the entry point to execute its associated code. Figure VII.5 shows the corresponding code for the program of Figure VII.4. The program is initialized with all trails set to `INACTIVE`. Then, the scheduler executes the *Main* label in the first trail. When the *Main* label reaches the `par/and`, it "stacks" the 2nd trail of the `par/and` to run on *TRAIL 1* (line 5-8) and proceeds to the code in the 1st trail (lines 10-15), respecting the deterministic execution order. The code sets the running *TRAIL 0* to await `EVT_e` on label `Awake_e`, and then halts with a `break`. The next iteration of the scheduler takes *TRAIL 1* and executes its registered label `ParAnd_sub_2` (lines 17-22), which sets *TRAIL 1* to await `EVT_A` and also halts.

Regarding cancellation, trails in parallel are always allocated in subsequent slots in the static vector `TRLS`. Therefore, when a `par/or` terminates, the scheduler sequentially searches and executes `FIN` trails within the range of the `par/or`, and then clears all of them to `INACTIVE` at once. Given that finalization blocks cannot contain `await` statements, the whole process is guaranteed to terminate in bounded time. Escaping a `loop` that contains parallel compositions also trigger the same process.

VII.4 The external *C* API

As a reactive language, the execution of a program in CÉU is guided entirely by the occurrence of external events. From the implementation perspective, there are three external sources of input into programs, which are all exposed as functions in a *C* API:

ceu_go_init(): initializes the program (e.g. trails) and executes the "boot" reaction (i.e., the `Main` label).

ceu_go_event(id,param): executes the reaction for the received event id and associated parameter.

ceu_go_wclock(us): increments the current time in microseconds and runs a reaction if any timer expires.

Given the semantics of CÉU, the functions are guaranteed to take a bounded time to execute. They also return a status code that says if the CÉU program has terminated after the reactions. Further calls to the API have no effect on terminated programs.

The bindings for the specific platforms are responsible for calling the functions in the API in the order that better suit their requirements. As an example, it is possible to set different priorities for events that occur concurrently (i.e. while a reaction chain is running). However, a binding must never interleave or run multiple functions in parallel. This would break the CÉU sequential/discrete semantics of time.

As an example, Figure VII.6 shows our binding for *TinyOS* which maps *nesC* callbacks to input events in CÉU. The file `ceu.h` (included in line 3) contains all definitions for the compiled CÉU program, which are further queried through `#ifdef`'s. The file `ceu.c` (included in line 4) contains the main loop of CÉU pointing to the labels defined in the program. The callback `Boot.booted` (lines 6-11) is called by TinyOS on mote startup, so we initialize CÉU inside it (line 7). If the CÉU program uses timers, we also start a periodic timer (lines 8-10) that triggers callback `Timer.fired` (lines 13-17) every 10 milliseconds and advances the wall-clock time of CÉU (line 15)[1]. The remaining lines map pre-defined TinyOS events that can be used in CÉU programs, such as the light sensor (lines 19-23) and the radio transceiver (lines 25-36).

---

[1] We also offer a mechanism to start the underlying timer on demand to avoid the "battery unfriendly" 10ms polling.

```
implementation
{
    #include "ceu.h"
    #include "ceu.c"

    event void Boot.booted () {
        ceu_go_init();
#ifdef CEU_WCLOCKS
        call Timer.startPeriodic(10);
#endif
    }

#ifdef CEU_WCLOCKS
    event void Timer.fired () {
        ceu_go_wclock(10000);
    }
#endif

#ifdef _EVT_PHOTO_READDONE
    event void Photo.readDone (uint16_t val) {
        ceu_go_event(EVT_PHOTO_READDONE, (void*)val);
    }
#endif

#ifdef _EVT_RADIO_SENDDONE
    event void RadioSend.sendDone (message_t* msg) {
        ceu_go_event(EVT_RADIO_SENDDONE, msg);
    }
#endif

#ifdef _EVT_RADIO_RECEIVE
    event message_t* RadioReceive.receive (message_t* msg) {
        ceu_go_event(EVT_RADIO_RECEIVE, msg);
        return msg;
    }
#endif

    <...>   // other events
}
```

Figure VII.6: The *TinyOS* binding for CÉU.

VIII
Related work

Figure VIII.1 presents an overview of work related to CÉU, pointing out supported features which are grouped by those that reduce complexity and those that increase safety. The line *Preemptive* represents asynchronous languages with preemptive scheduling [9, 29], which are summarized further. The remaining lines enumerate languages with goals similar to those of CÉU that follow a synchronous execution semantics.

Many related approaches allow events to be handled in sequence through a blocking primitive, overcoming the main limitation of event-driven systems (column 1 [14, 5, 33, 4, 27]). As a natural extension, most of them also keep the state of local variables between reactions to the environment (column 2). In addition, CÉU introduces a reliable mechanism to interface local pointers with the system through finalization blocks (column 8). Given that these approaches use cooperative scheduling, they can provide deterministic and reproducible execution (column 5). However, as far as we know, CÉU is the first system to extend this guarantee for timers in parallel.

Synchronous languages first appeared in the context of WSNs through OSM [28] and Sol [27], which provide parallel compositions (column 3) and distinguish themselves from multi-threaded languages by handling thread destruction seamlessly [35, 7]. Compositions are fundamental for the simpler reasoning about control that made possible the safety analysis of CÉU. Sol detects infinite loops at compile time to ensure that programs are responsive (column 6). CÉU adopts the same policy, which first appeared in Esterel. Internal events (column 4) can be used as a reactive alternative to shared-memory communication in synchronous languages, as supported in OSM [28]. CÉU introduces a stack-based execution that also provides a restricted but safer form of subroutines.

nesC provides a data-race detector for interrupt handlers (column 7), ensuring that *"if a variable x is accessed by asynchronous code, then any access of x outside of an atomic statement is a compile-time error"* [19]. The analysis of CÉU is, instead, targeted at synchronous code and points more precisely when accesses can be concurrent, which is only possible because of its restricted

Figure VIII.1: Table of features found in work related to Céu.
The languages are sorted by the date they first appeared in a publication. A gray background indicates where the feature first appeared (or a contribution if it appears in a Céu cell).

semantics. Furthermore, Céu extends the analysis for system calls (*commands* in *nesC*) and control conflicts in trail termination. Although *nesC* does not enforce bounded reactions, it promotes a cooperative style among tasks, and provides asynchronous events that can preempt tasks (column 6), something that cannot be done in Céu.

On the opposite side of concurrency models, languages with preemptive scheduling assume time independence among processes and are more appropriate for applications involving algorithmic-intensive problems. Preemptive scheduling is also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [9, 16, 17, 29]. The choice between the two models should take into account the nature of the application and consider the trade-off between safe synchronization and predictable responsiveness.

IX
Conclusion

We presented Céu, a system-level programming language targeting control-intensive WSN applications. Céu is based on a synchronous core that combines parallel compositions with standard imperative primitives, such as sequences, loops and assignments. Our work has three main contributions:

- A resource-efficient synchronous language that can express control specifications concisely.
- The stack-based execution policy for internal events as a powerful broadcast communication mechanism.
- A wide set of compile-time safety guarantees for concurrent programs that are still allowed to share memory and access the underlying platform in "raw *C*".

We argue that the dictated safety mindset of our design does not lead to a tedious and bureaucratic programming experience. In fact, the proposed safety analysis actually depends on control information that can only be inferred based on high-level control-flow mechanisms (which results in more compact implementations). Furthermore, Céu embraces practical aspects for the context of WSNs, providing seamless integration with *C* and a convenient syntax for timers.

As far as we know, Céu is the first language with stack-based internal events, which allows to build rich control mechanisms on top of it, such as a limited form of subroutines and exception handling. In particular, Céu's subroutines compose well with the other control primitives and are safe, with guaranteed bounded execution and memory consumption.

We presented two complete demos to show how typical patterns in WSNs such as sampling, timeout and concurrency can be easily implemented. They also explore parallel compositions for specifying complementary activities in separate. Communication among activities can either use internal events or safe access to global variables.

Our evaluation compares several implementations of widely adopted WSN protocols in Céu to *nesC*, showing a considerable reduction in code size with a small increase in resource usage. On the way to a more in-depth qualitative approach, we have been teaching CÉU as an alternative to *nesC* in hands-on WSN courses in a high school for the past two years (and also in two universities in short courses). Our experience shows that students are capable of implementing a simple multi-hop communication protocol in CÉU in a couple of weeks.

We presented a formal semantics for the control aspects of CÉU and discuss how they are implemented in *C*. The resource-efficient implementation of CÉU is suitable for constrained sensor nodes and imposes a small memory overhead in comparison to handcrafted event-driven code.

We believe that the strong position in favor of shared-memory concurrency is also a contribution of the thesis: first because although synchronous languages emerged in the early eighties, we are not aware of derived work allegedly addressing this issue; second because the current trend in the programming-languages community is towards the adoption of more pure functional languages and message-passing concurrency to get rid of shared memory, which is in the opposite direction of CÉU.

Bibliography

[1] A. Adya et al. Cooperative task management without manual stack management. In *ATEC'02*, pages 289–302. USENIX Association, 2002. I, II.2, III.1(b)

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

[3] Albert Benveniste et al. The synchronous languages twelve years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, Jan 2003. I, II.2

[4] Bergel et al. Flowtalk: language support for long-latency operations in embedded devices. *IEEE Transactions on Software Engineering*, 37(4):526–543, 2011. II, V, V.4, VIII

[5] Alexander Bernauer and Kay Römer. A comprehensive compiler-assisted thread abstraction for resource-constrained systems. In *Proceedings of IPSN'13*, Philadelphia, USA, April 2013. II, V, V.1, VII.2, VIII

[6] G. Berry. *The Esterel-V5 Language Primer.* CMA and Inria, Sophia-Antipolis, France, June 2000. Version 5.10, Release 2.0. III.7, V.4

[7] Gérard Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993. III.1(b), III.1(c), III.7, VIII

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

[9] Shah Bhatti et al. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10:563–579, August 2005. I, II.3, VIII

[10] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991. I, II.2, III.1(a)

[11] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proceedings of SenSys'07*, pages 205–218. ACM, 2007. I

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

[13] Dunkels et al. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of LCN'04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. I, V.2

[14] Dunkels et al. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*, pages 29–42. ACM, 2006. I, II, II.2, II.3, V, V.1, V.2, V.4, VIII

[15] Joakim Eriksson et al. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of SIMUTools'09*, page 27. ICST, 2009. V.3

[16] Muhammad Farooq and Thomas Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011. VIII

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

[18] Prasanth Ganesan et al. Analyzing and modeling encryption overhead for sensor network nodes. In *Proceedings of WSNA'03*, pages 151–159. ACM, 2003. V.3

[19] David Gay et al. The nesC language: A holistic approach to networked embedded systems. In *PLDI'03*, pages 1–11, 2003. I, II, II.2, II.3, V.4, VIII

[20] Omprakash Gnawali et al. Collection tree protocol. In *Proceedings of SenSys'09*, pages 1–14. ACM, 2009. V, V.3

[21] 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. II.2

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. II.2

[23] Hill et al. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, November 2000. I, II.2, IV.1

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

[25] Christian L. Jacobsen et al. Concurrent event-driven programming in occam-pi for the Arduino. In *CPA '11*, volume 68, pages 177–193, June 2011. II.1

[26] Chris Karlof et al. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of SenSys'04*, pages 162–175. ACM, 2004. V.3

[27] Marcin Karpinski and Vinny Cahill. High-level application development is realistic for wireless sensor networks. In *Proceedings of SECON'07*, pages 610–619, 2007. I, II, V, V.4, VIII

[28] Oliver Kasten and Kay Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of IPSN '05*, pages 45–52, April 2005. I, II, II.2, VII.2, VIII

[29] Kevin Klues et al. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proceedings of SenSys'09*, pages 127–140, New York, NY, USA, 2009. ACM. V.3, V.4, VIII

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

[31] Phil Levis et al. Trickle: A self-regulating mechanism for code propagation and maintenance in wireless networks. In *Proceedings of NSDI'04*, volume 4, page 2, 2004. V, V.3

[32] Philip Levis. Experiences from a decade of TinyOS development. In *Proceedings of OSDI'12*, pages 207–220, Berkeley, CA, USA, 2012. USENIX Association. I, V.2, V.3

[33] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of SenSys'06*, pages 167–180, New York, NY, USA, 2006. ACM. I, II, VIII

[34] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM TOPLAS*, 31(2):6:1–6:31, February 2009. II.2

[35] ORACLE. Java thread primitive deprecation. `http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`, 2011. III.1(b), VIII

[36] Dumitru Potop-Butucaru et al. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. 2005. I, III.1

[37] Mohammad Rahimi et al. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In *Proceedings of SenSys'05*, pages 192–204. ACM, 2005. V.3

[38] Christopher M Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of SenSys'06*, pages 265–278. ACM, 2006. V.3

[39] TinyOS TEPs. `http://docs.tinyos.net/tinywiki/index.php/TEPs`, 2013. I, III.4, V

[40] Ben L Titzer. Virgil: Objects on the head of a pin. In *ACM SIGPLAN Notices*, volume 41, pages 191–208. ACM, 2006. V.4

[41] Ning Xu et al. A wireless sensor network for structural monitoring. In *Proceedings of SenSys'04*, pages 13–24. ACM, 2004. V.3