Safe System-level Concurrency on Resource-Constrained Nodes

Francisco Sant'Anna
fsantanna@inf.puc-rio.br

Noemi Rodriguez
noemi@inf.puc-rio.br

Roberto Ierusalimschy
roberto@inf.puc-rio.br

Departamento de Informática – PUC-Rio, Brazil

Olaf Landsiedel
olafl@chalmers.se

Philippas Tsigas
tsigas@chalmers.se

Computer Science and Engineering – Chalmers University of Technology, Sweden

ABSTRACT

Despite the continuous research to facilitate WSNs development, most safety analysis and mitigation efforts in concurrency are still left to developers, who must manage synchronization and shared memory explicitly. In this paper, we present a system language that ensures safe concurrency by handling threats at compile time, rather than at runtime. Based on the synchronous programming model, 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.

1. INTRODUCTION

System-level development for Wireless Sensor Networks (WSNs) commonly follows one of three major programming models: *event-driven*, *multi-threaded*, or *synchronous*. In event-driven programming [19, 11], 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, 12]. Multi-threaded systems emerged as an alternative in WSNs, providing traditional structured programming in multiple lines of execution [12, 7]. However, the development process still requires manual synchronization and bookkeeping of threads [24]. Synchronous languages [2] have also been adapted to WSNs and offer higher-level compositions of activities with a step-by-step execution, considerably reducing programming efforts [21, 22].

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 [25]. 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 [7, 27], or by adopting cooperative scheduling which also requires explicit yield points in the code [21, 12]. In either case, there are 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. In this work, we present the design of 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 [8], 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 at compile time. This allows Céu to support safe shared-memory concurrency as well as system-level *C* calls. In addition, we introduce the following new safety mechanisms: *first-class timers* to ensure precise synchronization among timers in parallel (not depending on internal reaction timings); *finalization blocks* for local pointers going out of scope; and *stack-based communication* that avoids cyclic dependencies. In contrast with existing synchronous languages, Céu introduces a stack-based execution for internal events which enables a limited but safer form of subroutines. Our work focuses on *concurrency safety*, rather than *type safety* [9].[2]

SenSys'13, November 11-15, 2013, Rome, Italy.

ACM 978-1-4503-2027-6/13/11 $15.00.
http://dx.doi.org/10.1145/2517351.2517360.

[1] Céu is the Portuguese word for *sky*.
[2] We consider both safety aspects to be complimentary and

```
/* 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
```

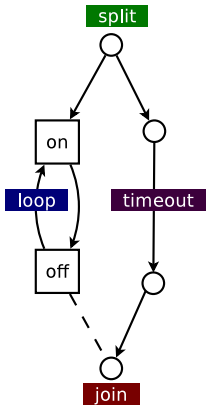

Figure 1: "Blinking LED" in nesC, Protothreads, and Céu.

In order to enable static analysis, programs in Céu must suffer some limitations. Computations that run in unbounded time (e.g., compression, image processing) cannot be elegantly implemented [29], 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 [32] in Céu. In comparison to *nesC* [17], the implementations reduced the number of source code tokens by 25%, with an increase in ROM and RAM below 10%.

The rest of the paper is organized as follows: Section 2 gives an overview on how different programming models used in WSNs can express typical control patterns. Section 3 details the design of Céu, motivating and discussing the safety aspects of each relevant language feature. Section 4 evaluates the implementation of the 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, showing extreme high-load conditions in which the disciplined synchronous execution of our model may not be suitable. Section 5 discusses related work to Céu. Section 6 concludes the paper and makes final remarks.

2. OVERVIEW OF PROGRAMMING MODELS

WSN applications must handle a multitude of concurrent events, such as timers and packet transmissions. Although they may seem random and unrelated for an external observer, a program must logically keep track of them according to its control 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 including an 1-minute timeout to interrupt an activity denotes a parallel pattern.

Figure 1 presents the three different programming models commonly used in WSNs. It shows the implementations in *nesC* [17], *Protothreads*[12], 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 right of Figure 1 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.

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, 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

orthogonal, i.e., type-safety techniques could also be applied to Céu.

```
// 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 both sides
      with <...> end
par/or do  <...>         // rejoins on 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 2: Syntax of Céu.

synchronous 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 lines of execution terminates. (Céu also supports `par/and` compositions, which rejoin when *all* spawned lines of execution terminate.) 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.

3. 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 2 shows a compact reference of Céu, which helps to follow the examples in this chapter.

As an introductory example, the code in Figure 3 is

```
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 3: 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).

extracted from our implementation of the *CC2420* radio driver [32] 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); upon 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 and contain other loops and parallel constructs. However, once the client requests to stop the driver, the trail in line 5 awakes and terminates, also terminating the `par/or` which aborts the receiving loop and proceeds to the statement in sequence. In this case, the top-level loop restarts, waiting for the next request to start (line 3, again).

The `par/or` construct is regarded as an *orthogonal preemption primitive* [5] 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 [5, 28].

3.1 Deterministic and Bounded Execution

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 a program in Céu 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.
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 [29]. 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 source code. 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 3.6.

The blinking LED of Figure 1 in Céu illustrates how the synchronous model leads 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 $2 + 1$ seconds. Hence, after 20 iterations, the accumulated time becomes 1 minute and the loop terminates concurrently with the 1-minute timeout in parallel. Given that the loop appears first, it will restart 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 depth in Section 3.5. Note that this static control inference cannot be easily extracted from the other implementations of Figure 1, 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 one last time. By inverting the two trails, the `par/or` would terminate immediately, and the blinking trail would not execute, denoting a *strong abortion* [5]. 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 3.2).

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 [8], 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
        await A;
    end
end
```

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

3.2 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 example that follows, the two assignments to `x` run concurrently, because both trail segments are spawned during the same reaction chain. However, in the second example, 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[3]: the final value of `x` is always 4 (i.e. $(1 + 1) * 2)$). 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 4. The first code is detected 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. 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 3.1, 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

[3]Remember from Section 3.1 that trails are scheduled in the order they appear and run to completion (i.e., until they await or terminate).

```
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 4: 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.

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 false positives in the static analysis. 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 4 and executes in negligible time, suggesting that the algorithm is practical.

3.3 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 5 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 3.2 and cannot appear in concurrent trails segments, because Céu has no knowledge about their side

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

```
1  pure  _abs();           // side-effect free
2  safe _Leds_led0Toggle with
3       _Leds_led1Toggle; // led0 vs led1 is safe
4  var int* buf1, buf2;   // point to dif. bufs
5  safe buf1 with buf2;   // buf1 vs buf2 is safe
```

Figure 6: 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).

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 6, and are described as follows:

- The `pure` modifier declares a *C* function that does not cause side effects, allowing it to be called concurrently with any other function in the program.
- The `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.

3.4 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

```
1  <...>
2  par/or do
3    <...>          // stops the protocol or radio
4  with
5    <...>          // neighbor 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 7: 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.

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 7 was extracted from our implementation of the CTP collection protocol [32]. The protocol contains a complex control hierarchy in which the trail that sends beacon frames (lines 11-16) may be 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). Then, 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 rewritten as shown in Figure 8.

First, the `nohold` annotation informs the compiler that

```
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   <...>
```

Figure 8: Safe use of local references.
The call to `_AMSend_send` is finalized with the call to `_AMSend_cancel`. The call to `_AMSend_getPayload` does not require finalization because it does not hold pointers.

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), releasing the reference held by the radio driver.

All network protocols that we implemented in Céu use this finalization mechanism when sending messages. We looked through the TinyOS codebase 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.

3.5 First-class Timers

Activities that involve reactions to *wall-clock time*[4] 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 blocking calls with "sleep". 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.

The `await` statement of Céu supports wall-clock time and handles deltas automatically, resulting in more robust

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

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 1 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
```

3.6 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-based 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*[5].

```
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_RECV;
13     <...>
14     emit send => 0;     // broadcasts data
15 with
16   loop do
17     var int isMeta = await send;
18     <...>            // sends data or metadata
19   end                //   (contains awaits)
20 end
```

Figure 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).

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 9 shows how the dissemination trail from our implementation of the DRIP protocol simulates a subroutine (lines 16-19) and can be invoked from different parts of the program. 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).

Internal events also provide means for describing more elaborate control structures, such as *exceptions*. The code in Figure 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

[5]In order to ensure bounded reactions, an `await` statement cannot awake in the same reaction chain it is invoked.

```
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 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.

paused. Therefore, the continuation for the `emit` never resumes, and the loop restarts to await a next packet.

3.7 Differences to Esterel

Esterel first introduced the imperative synchronous programming model and influenced other synchronous WSNs languages [21, 22]. Based on the discussion about our design in the previous sections, we next summarize the main differences between Céu and Esterel (and derived languages).

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 3.1, 3.2 and 3.3). 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 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 3.2.

Esterel makes no semantic distinctions between internal and external signals, both having only the notion of presence or absence during the entire reaction [5]. 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 for internal events in Céu enables a limited but safe form of subroutines and an exception-handling mechanism, as discussed in Section 3.6.

Apart from these fundamental differences to Esterel, Céu introduces first-class timers with a convenient syntax and predictable behavior (Section 3.5), and also finalization blocks to safely handle memory going out of scope (Section 3.4).

4. EVALUATION

In this section we present a quantitative evaluation of Céu. Our assumption is that when considering Céu for system-level development, programmers would face a trade-off 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 [32]. 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[6]. In addition, *nesC* is used as benchmark in many systems related to Céu [12, 21, 4, 3]. In particular, the work on *Protothreads* [12] 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 [32]: 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 [26, 18]; routing, dissemination, and collection are the most common classes of protocols in WSNs.

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

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

Figure 11: 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.

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 11 shows the comparison for *Code size* and *Memory usage* between the implementations in *nesC* and Céu. For memory usage, detailed in Section 4.2, we compare the binary code size and required RAM. For code size, detailed in Section 4.1, we compare the number of tokens used in the source code. For responsiveness, detailed in Section 4.3, we evaluate the capacity to promptly acknowledge radio packet arrivals in the *CC*2420 driver.

4.1 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 [4, 12], 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 11 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 [12]. 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[7]. 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 11 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 11 point out how many times each functionality has been used in the implementations in Céu, helping to identify where the reduction in source code 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.

[7]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.

4.2 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 11 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 [12], which is an actively supported programming system for the Contiki OS [11].

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 [25]. 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.

4.3 Responsiveness

A known limitation of languages with synchronous and cooperative execution is that they cannot guarantee hard real-time deadlines [10, 23]. 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 codebase). 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 [13] running images compiled to *TelosB* motes.

In the first experiment, we "stress-test" the radio driver to compare its performance in the Céu and *nesC* implementations. We use 10 motes that broadcast 100 consecutive packets of 20 bytes to a mote that runs a periodic time-consuming activity. The receiving handler simply adds the value of each received byte to a global counter. The sending rate of each mote is 200ms (leading to a receiving average of 50 packets per second considering the 10 motes), and the time-consuming activity in the receiving mote runs every 140ms. Note that these numbers are much above typical WSN applications: 10 neighbors 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 1). We assume that the lengthy operation is implemented directly in *C* and cannot be easily split in smaller operations (e.g., recursive algorithms [10, 23]). So, we simulated them with simple busy waits that would keep the driver in Céu unresponsive during that period.

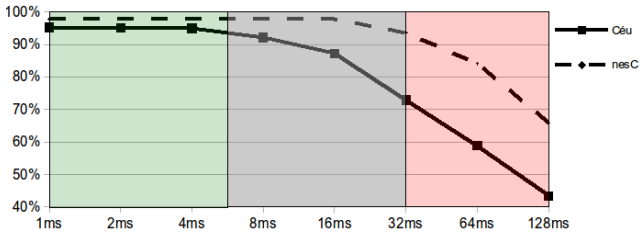

Figure 12: Percentage of received packets depending on the duration of the lengthy operation.
Note the logarithmic scale on the *x*-axis. The packet arrival frequency is 20ms. The operation frequency is 140ms. In the (left) green area, Céu performs similarly to *nesC*. The (middle) gray area represents the region in which *nesC* is still responsive. In the (right) red area, both implementations become unresponsive (i.e. over 5% packet losses).

Operation	Duration
Block cipher [20, 16]	1ms
MD5 hash [16]	3ms
Wavelet decomposition [34]	6ms
SHA-1 hash [16]	8ms
RLE compression [31]	70ms
BWT compression [31]	300ms
Image processing [30]	50–1000ms

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

Figure 12 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 [23]. Table 1 shows the duration of some lengthy operations specifically designed for WSNs found in the literature. The operations in the group with timings up to 6*ms* could be used with real-time responsiveness in Céu (considering the proposed high-load parameters).

Although we did not perform specific tests to evaluate

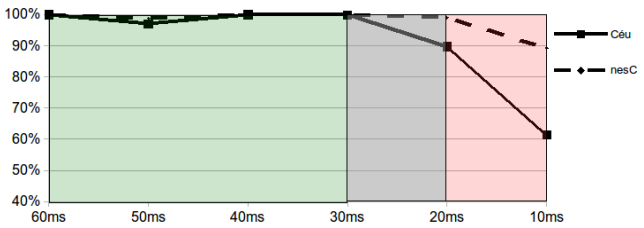

Figure 13: Percentage of received packets depending on the sending frequency.
Each received packet is tied to a 8-ms operation. Céu is 100% responsive up to a frequency of 30ms per packet.

CPU usage and battery consumption, the experiment suggests that the overhead of Céu over *nesC* is very low. When the radio driver is the only running activity (column $1ms$, which is the same result for an addition test we did for $0ms$), both implementations loose packets with a difference under 3 percentage points. This difference remains the same up to 4-ms activities, hence, the observed degradation for longer operations is only due to the lack of preemption, not execution speed. Note that for lengthy operations implemented in *C*, there is no runtime 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 13 shows the percentage of handled packets in Céu and *nesC* for each rate of message arrival. The results show that Céu is 100% responsive up to frequency of 33 packets per second, while *nesC* up to 50 packets.

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

Note that our experiments represent a "stress-test" scenario that is atypical to WSNs. Protocols commonly use longer intervals between message transmissions together with mechanisms to avoid contention, such as randomized timers [26, 18]. Furthermore, WSNs are not subject to strict deadlines, being not classified as hard real-time systems [25].

4.4 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 start-up 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 3.1 and 3.6).
- Reliable weak and strong abortion among activities (Sections 3.1 and 3.2).
- No concurrency in accesses to shared variables (Section 3.2).
- No concurrency in system calls sharing a resource (Section 3.3).
- Finalization for blocks going out of scope (Section 3.4).
- Auto-adjustment for timers in sequence (Section 3.5).
- Synchronization for timers in parallel (Section 3.5).

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 [3, 33]. Given that we focus on system-level development which does not require rich dynamic functionality, we leave for a future work an in-depth discussion about this issue. Note that queues, stacks, and other simple dynamic data structures for handling message packets can be made available to Céu through its safe integration with *C*.

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` [12], `yield` [21], `post` [17]). Considering the second option, Céu and preemptive threads are not mutually exclusive. For instance, TOSThreads [23] proposes a message-based integration with *nesC* that is safe and matches the semantics of Céu external events.

5. RELATED WORK

Céu is strongly influenced by Esterel [8] in its support for compositions and reactivity to events. However, Esterel is focused only on control and delegates to programmers most efforts to deal with data and low-level access to the underlying platform. For instance, read/write to shared memory among threads is forbidden, and avoiding conflicts between concurrent *C* calls is left to programmers [6]. Céu deals with shared memory and *C* integration at its very core, with additional support for finalization, conflict annotations, and a static analysis that permeates all languages aspects. This way, Céu could not be designed easily as pure extensions to Esterel.

Language		Complexity				Safety			
name	year	1: sequential execution	2: local variables	3: parallel compositions	4: internal events	5: deterministic execution	6: bounded execution	7: safe shared memory	8: finalization blocks
Preemptive	many	✔	✔		✔		*rt*		
nesC [17]	2003					✔	*async*	✔	
OSM [22]	2005		✔	✔	✔				
Protothreads [12]	2006	✔				✔			
TinyThreads [27]	2006	✔	✔			✔			
Sol [21]	2007	✔	✔	✔		✔	✔		
FlowTalk [3]	2011	✔	✔						
Ocram [4]	2013	✔	✔			✔			
Céu		✔	✔	✔	✔	✔	✔	✔	✔

Figure 14: Table of features found 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).

Figure 14 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 languages with preemptive scheduling [7, 23], which are summarized further. The remaining lines enumerate languages with goals similar to those of Céu that follow a synchronous or cooperative 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 [12, 4, 27, 3, 21]). 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 [22] and Sol [21], which provide parallel compositions (column 3) and distinguish themselves from multi-threaded languages by handling thread destruction seamlessly [28, 5]. 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. 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"* [17]. 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 semantics. Furthermore, Céu extends the analysis for system calls (*commands* in *nesC*), as well as 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 [7, 14, 15, 23]. 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.

6. 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.

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.

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.

7. ACKNOWLEDGMENTS

This work was partially supported by grants from CNPq (Brazil), SAAB (Sweden), and the European Union Seventh Framework Programme (FP7/2007-2013) under agreement No. 257007.

8. REFERENCES

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

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

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

[4] A. Bernauer and K. Römer. A comprehensive compiler-assisted thread abstraction for resource-constrained systems. In *Proceedings of IPSN'13*, Philadelphia, USA, Apr. 2013.

[5] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *LNCS*, pages 72–93. Springer, 1993.

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

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

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

[9] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of SenSys'07*, pages 205–218. ACM, 2007.

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

[11] 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.

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

[13] J. Eriksson et al. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *Proceedings of SIMUTools'09*, page 27. ICST, 2009.

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

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

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

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

[18] O. Gnawali et al. Collection tree protocol. In *Proceedings of SenSys'09*, pages 1–14. ACM, 2009.

[19] Hill et al. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, November 2000.

[20] C. Karlof et al. TinySec: a link layer security architecture for wireless sensor networks. In *Proceedings of SenSys'04*, pages 162–175. ACM, 2004.

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

[22] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of IPSN '05*, pages 45–52, April 2005.

[23] K. 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.

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

[25] P. Levis. Experiences from a decade of TinyOS development. In *Proceedings of OSDI'12*, pages 207–220, Berkeley, CA, USA, 2012. USENIX Association.

[26] P. 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.

[27] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded

systems. In *Proceedings of SenSys'06*, pages 167–180, New York, NY, USA, 2006. ACM.

[28] ORACLE. Java thread primitive deprecation. `http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html` (accessed in Aug-2014), 2011.

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

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

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

[32] TinyOS TEPs. `http://docs.tinyos.net/tinywiki/index.php/TEPs`, 2013.

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

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