Safe Concurrent Abstractions for Wireless Sensor Networks

Abstract

ABSTRACT

1 Introduction

Applications developed for wireless sensor networks (WSNs) typically perform a succession of sensing, processing, actuating, and communicating. This process involves an external environment, which concurrently interacts with the application by issuing events that represent expiring timers, messages arrivals, sensor readings, etc.

The first languages and operating systems for WSNs provide an event-driven programming model [15, 10], in which each external event can be associated with a short-lived function callback to handle a reaction to the environment. This model is efficient for the severe resource constraints of WSNs, but is difficult to program, given the ineffectiveness of local variables [2] and the explicit management of state machines [11].

Multi-threaded systems emerged as an alternative, providing traditional structured programming for WSNs (e.g. [11, 6]). However, the programmer still has to manually synchronize and maintain threads (e.g. create, start, and destroy). Furthermore, preemptive scheduling of threads is a potential source of safety hazards [18], while cooperative scheduling is susceptible to unbounded execution (i.e. infinite loops), breaking responsiveness in programs [9].

Synchronous languages are a higher-level alternative that have been successfully adapted to WSNs without imposing a significant overhead [16, 17]. They provide high-level compositions of concurrent activities through hierarchies of processes [7] or state machines [14], reducing the programmer efforts with synchronization issues.

However, existing synchronous languages targetting WSNs do not consider safety aspects in a broad view, suffering from the same difficulties of multi-threaded designs. For instance, shared memory is usually the only mechanism for communication and synchronization, but current works do not go beyond atomic access guarantees relying on a run to completion semantics [16, 17].

Furthermore, WSN development typically involves low-level access to the platform through *C* system calls that pass pointers back and forth (e.g. message buffers to the radio driver). Current works do not discuss effective policies to handle such corner cases. For instance, the claimed gains in memory and expressiveness with support for locals [4, 17] are under safety threats if pointers that go out of scope are used.

In this work, we present the design of CÉU[1], a system-level programming language based on Esterel [7] that provides a reliable yet powerful programming environment for WSNs. We explore the precious control information that can be inferred from compositions at compile time in order to embrace safety aspects to a greater extend.

Our design is first compromised with the main principles that govern WSNs development: *resource minimization* and *bug prevention*, as defined by Levis [19]. That said, support for hierarchical compositions, together with a convenient syntax for timers and internal communication also lead to more compact programs (up to 70% reduction, in our evaluation).

Currently, our design focus only on *concurrency safety*, rather than on *type safety* [8]. We consider both aspects to be complimentary and orthogonal, i.e., type safety annotations and runtime checks could also be applied to CÉU.

Our overall contribution is the careful design of concurrent abstractions that considers safety aspects for every proposed functionality. As more specific contributions, we enumerate the following language features:

- A compile-time analysis that enforces reliable shared-memory concurrency.
- A finalization mechanism to safely release resources just before they go out of scope.
- First-class timers with a predictable behavior.
- A stack-based inter-thread communication mechanism that provides a restricted (but safer) form of subroutines.
- An object system that provides code reentrancy and enables the creation of higher-level abstractions.

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

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

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

```
/******** CEU ********/
par/or do
   loop do
      _Leds_led0On();
      await 2s;
      _Leds_led0Off();
      await 1s;
   end
with
   await 1min;
end
_Leds_led0Off(0);
<...> // CONTINUE
```

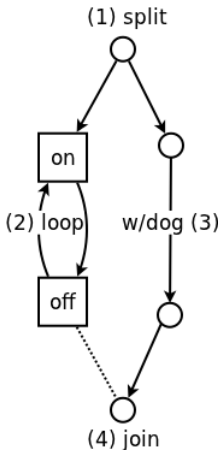

Figure 1. "Blinking LED" in nesC [12], Protothreads [11], and CÉU. (TODO: point (1..4) in the impls.)

As a limitation of the synchronous model, computations that run in unbounded time (e.g., cryptography, image processing) do not fit the zero-delay reaction hypothesis [21], and cannot be elegantly implemented in CÉU. Also, the static analysis precludes any dynamic support in the language, such as memory allocation and dynamic loading. However, this trade-off seems to be favorable in the context of WSNs, as dynamic features are discouraged due to resource constraints and safety requirements.

The rest of the paper is organized as follows: Section 2 gives an overview of how different programming models used in WSNs can express typical control patterns. Section 3 details the design of CÉU by motivating, describing, exemplifying, and discussing safety aspects of each relevant language feature. In Section 4, we describe how we ported to CÉU the open-source implementations of some open standards [1] (e.g. network protocols and a radio driver), comparing some quantitative aspects with *nesC* (e.g. memory usage and tokens count) backed by a thoughtful discussion. Section 5 presents a XXX of related works to CÉU. Section 6 concludes the paper and makes final remarks.

2 Overview of programming models

As briefly introduced, WSNs applications must handle a multitude of concurrent events, such as from timers and packet transmissions. Although they may seem random and unrelated for an external observer, the programmer keeps track of them in a logical fashion, according to the application specification.

From a control perspective, the logical relations among events represent activities in an application that follow two main patterns: *sequential*, i.e., activities composed of two or more states in sequence; or *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 clear sequential pattern (with an enclosing loop); while including an 1-minute watchdog timer to interrupt the activity comprises a parallel pattern.

An effective language for WSNs should provide control abstractions that better express the programmer intetions, desirably ensuring that the high degree of concurrency does not impose safety threats.

Figure 1 exposes the different programming models used in WSNs, showing three implementations for an application that continuously lights on a LED for 2 seconds and off for 1 second. After 1 minute, the application turns off the LED and proceeds to the code marked as `<...>`. The diagram on the right describes the control behavior for the application. The sequential pattern is represented with the LED alternating between the two states, while the parallel pattern is represented by the 1-minute watchdog that interrupts the blinking LED.

The first implementation, which represents the *event-driven* model [12, 10], spawns two timers at boot time (`Boot.booted`), one to blink the LED 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 once, cancelling the blinking timer and proceeding to `<...>`. This implementation is the least structured, given that the blinking loop is not explicit, but instead, relies on a static state variable and multiple invocations of the same callback. Furthermore, the 1-minute watchdog needs specific knowledge on how to stop the blinking activity manually (`T1.cancel()`).

The second implementation, which represents the *multi-threaded* model [11, 6], uses a dedicated thread to blink the LED in a loop, bringing more structure to the solution. The main thread also helps identifying 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 a lot of bookkeeping for initializing, scheduling and rejoining the blinking thread after the watchdog expires.

The third implementation, in CÉU, which represents the *synchronous model* [17, 16], 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

```
input void CC2420_START, CC2420_STOP;
loop do
    await CC2420_START;
    par/or do
        await CC2420_STOP;
    with
        // loop with other nested trails
        // to receive radio packets
        <...>
    end
end
```

Figure 2. Start/stop behavior for the radio driver.

terminating. A `par/or` stands for *parallel or* and rejoins automatically when any of its trails terminates. (CÉU also supports `par/and` compositions, which rejoin when *all* spawned trails terminate.)

The hierarchical syntactic structure more closely reflects the 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 provided in 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 occurring events to all active trails, which share a single global time reference (an 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 conventional multi-threaded systems typically only support top-level definitions for threads.

The example in Figure 2 is extracted from our port of the *CC2420* radio driver [1] to CÉU and uses a `par/or` to control the start/stop behavior of the radio. The input events `CC2420_START` and `CC2420_STOP` 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; upon request, the driver spawns two other trails: one that awaits the stopping event, and another to actually receive radio messages in a loop.

As compositions can be nested, the receive loop can be as complex as needed, but once an external request to stop the driver is triggered, the `par/or` composition kills all nested trails and proceeds to the statement in sequence. In this case, the top-level loop restarts, waiting again for the start event.

The `par/or` construct is regarded as an *orthogonal preemption primitive* [5] because the two sides in the composition do not know when and why they get killed. Furthermore, they need not to be tweaked with synchronization primitives or state variables in order to be affected by related trails in parallel.

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 a complete reaction always executes in bounded time (to be discussed further). The execution model for a CÉU program 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 model is based on the hypothesis that internal reactions run *infinitely faster* than the rate of events from the environment [21]. 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 from 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 a deterministic and reproducible execution for programs.

The blinking LED example of Figure 1 illustrates how the synchronous model leads to a simpler reasoning about concurrency aspects. As reaction times are assumed to be instantaneous, the blinking loop takes exactly 3 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 blinking trail is defined first, the loop restarts and turns on the LED for the last time. Then, the blinking trail awaits again, and the 1-minute timeout is scheduled and kills the whole `par/or`. This reasoning is reproducible in practice, and the LED will light on exactly 21 times for every execution of this program.

The described behavior is known as *weak abortion*, because the blinking trail has the chance to execute for the last time. By inverting the two trails, the `par/or` terminates immediately, and the blinking trail does not execute, qualifying a *strong abortion*. [5]

CÉU not only provides means to choose between weak and strong abortion, but also detects the two possibilities and issues a warning at compile time (to be discussed in Section 3.2).

Reactions to the environment should run in bounded time to guarantee that programs are responsive and can handle upcoming input events. Similarly to Esterel [7], 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.

3.2 Shared-memory concurrency

WSNs applications make extensive use of shared memory, such as memory pools, message queues, routing tables, etc. Hence, an important goal of CÉU is to ensure a reliable execution for concurrent programs that use shared memory.

Concurrency in CÉU is characterized when two or more trails segments execute during the same reaction chain. A trail segment is a sequence of statements separated by an `await`.

In first example that follows, the assignments run concurrently, because both trail segments are spawned during the same reaction chain. However, in the second example, the assignments are never concurrent, because *A* and *B* represent different external events and the respective segments can never execute during the same reaction chain:

```
var int v=0;
par/and do
    v = v + 1;
with
    v = v * 2;
end
```

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

Note that although variable *v* is accessed concurrently in the first example, the assignments are both atomic and deterministic (given the run to completion semantics and the scheduling policy): the final value of *v* is always 2.

However, programs with concurrent accesses to shared memory are suspicious, because an apparently innocuous reordering of trails modifies the semantics of the program.

We developed a compile-time temporal analysis for CÉU in order to detect concurrent accesses to shared variables: 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.

For each variable access, the algorithm holds the set of all possible preceding `await` statements. Then, the sets for all accesses in parallel trails are compared to assert that no `await` statements are shared. Otherwise the compiler warns the programmer about the suspicious accesses.

Consider the three examples of Figure 3. The first code is detected as suspicious, given that both assignments may be concurrent in a reaction to *A* (lines 11 and 14); In the second code, although two of the assignments occur in reactions to *A* (lines 5 and 11), they are not in parallel trails and, hence, are safe. The third code illustrates a false positive in our

```
 1: input void A;
 2: var int v;
 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:   v = 1;
12: with
13:   await A;
14:   *p = 2;
15: end
```

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

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

Figure 3. The first and third programs are suspicious.

algorithm, as the assignments in parallel can only occur in different reactions to *A* (lines 5 and 9).

The proposed static analysis is only possible due to 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.

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. However, the algorithm is exponential and may be impractical in many situations.

That said, we run the simpler static analysis for all ports presented in Section 4 and no false positives were detected, suggesting that the algorithm is practical.

Weak and strong abortions, as presented in Section 3.1, are also detected with the proposed algorithm. Instead of accesses to variables, the algorithm asserts that no join points of a `par/or` execute concurrently with a trail in parallel, issuing a warning otherwise.

3.3 Integration with *C*

Most existing operating systems, programming languages, and libraries for WSNs rely on *C*, given its omnipresence and level of portability across embedded platforms. This way, 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. This way, access to *C* is seamless and, more importantly, easily trackable.

CÉU also supports *C blocks* to define new symbols, as Figure 4 illustrates. All code inside "`C do ... end`" is also repassed to the *C* compiler for the final generation phase. Note that CÉU mimics the type system of *C*, so that values can be seamlessly passed back and forth between the languages.

C calls are fully integrated with the static analysis of Section 3.1 and cannot appear in concurrent trails segments, given that 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

```
C do
    #include <assert.h>
    int I = 0;
    int inc (int i) {
        return I+i;
    }
end
C _assert(), _inc(), _I;
return _assert(_inc(_I));
```

Figure 4. A CÉU program with embedded *C* definitions.

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 syntactic annotations that the programmer can use to relax the policy explicitly:

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

The following code illustrates CÉU annotations:

```
pure  _abs();      // 'abs' is side-effect free
deterministic      // 'led0Toggle' vs 'led1Toggle' is ok
    _Leds_led0Toggle with _Leds_led1Toggle;
int*  buf1, buf2;  // point to different buffers
deterministic      // 'buf1' vs 'buf2' is ok
    buf1 with buf2;
```

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 give the programmer means to circumvent the rigor of CÉU in a well-marked way (the special underscore syntax).

Evidently, the programmer should only recur to *C* for I/O operations that are assumed to be instantaneous, but never for control activities.

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, given that they are never active at the same time.

Once again, due to syntactic compositions of trails, the CÉU compiler can statically allocate and optimize memory usage [17]: memory for trails in parallel must coexist; trails that follow rejoin points may reuse all memory.

However, the use of locals may introduce subtle bugs when dealing with pointers and *C* functions. Given that global *C* functions outlive the scope of locals, a pointer passed as parameter may be used after the referred variable goes out of scope.

The code snippet in Figure 5 was extracted from our port of the CTP collection protocol [1] to CÉU. The protocol contains a complex control hierarchy in which the trail that sends beacon frames may be killed or restarted from multiple sources: stop the protocol or radio, explicit resend request, or a neighbour request (all collapsed in lines 3, 5, and 9).

The sending loop (lines 7-18) awakes when the beacon timer expires (line 11). The message buffer is declared only where it is required (line 12, in the 6th depth-level of the program) and its reference is manipulated by two *TinyOS* functions: `AMSend_getPayload` (line 13), which gets the data region of the message to be prepared (collapsed in line 14); and `AMSend_send` (line 15), which requests the operating system to actually send the message.

However, the radio driver runs asynchronously with the protocol and holds the reference to the message until it is completely transmitted, signaling back the `AMSEND_SENDDONE` event (line 16). In the meantime, the sending trail may be killed, resulting in a dangling pointer in the program.

A possible solution is to change each trail that kills the sending trail to call `AMSend_cancel`. This would require to expand the scope of the message buffer and a state variable to keep track of the sending status, increase considerably the complexity of the application.

CÉU provides a safer and simpler solution with the following rule: *pointers passed to C functions require finalization code to safely handle the variable going out of scope*.

This rule prevents the previous example to compile, forcing the relevant parts to be be rewritten as

```
C nohold _AMSend_getPayload();
    <...>
        var _message_t msg;
        <...>
        finalize
            _AMSend_send(..., &msg, ...);
        with
            _AMSend_cancel(&msg);
        end
    <...>
```

The `nohold` annotation informs the compiler that the referred *C* function does not requires finalization code because it does not hold references. The `finalize` construct executes the `with` clause when the block containing the variable passed as parameter in the `finalize` clause goes out of scope. This way, regardless of how the sending loop is killed, the finalization code always executes and politely informs the OS to cancel the ongoing send operation.

The send/cancel pattern occurs in all ported applications that use the radio for communication evaluated in Section 4.

3.5 Wall-clock time

Activities that involve reactions to *wall-clock time*[2] appear in typical patterns of WSNs, such as sensor sampling and watchdogs. 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, a requested timeout does not expire precisely with zero-delay, a fact that is usually ignored by programmers. 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) might accumulate a considerable amount of deltas that could lead to incorrect behavior in programs.

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

```
 1:  <...>
 2:  par/or do
 3:      <...>           // stop the protocol or radio
 4:  with
 5:      <...>           // neighbour request
 6:  with
 7:      loop do
 8:          par/or do
 9:              <...>   // resend
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 5. Unsafe use of local references.

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:

```
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 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`. However, the current delta is higher than the requested timeout (i.e. $5ms > 1ms$), so the trail is immediately 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 CÉU cannot guarantee that the first trail terminates exactly in 11ms, it can at least ensure that the program always returns 1:

```
par do
    await 10ms;
    <...>         // any non-awaiting sequence
    await  1ms;
    return 1;
with
    await 12ms;
    return 2;
end
```

A similar program in a language without first-class support for timers, would depend on the execution timings for the code marked as `<...>` (which is assumed to be instantaneous in the synchronous model).

3.6 Internal events

CÉU provides internal events as a signaling mechanism among trails in parallel: a trail that invokes `await e` can be awaken 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.

```
event int send;
par do
    <...>
        await DRIP_KEY;
        emit send=1;        // broadcast data
with
    <...>
        await DRIP_TRICKLE;
        emit send=0;        // broadcast meta
with
    <...>
        var _message_t* msg = await DRIP_DATA_RECEIVE;
        <...>
        emit send=1;        // broadcast data
with
    loop do
        var int data? = await send;
        <...>   // send data or metadata
    end
end
```

Figure 6. The `send` "subroutine" is invoked from different parts of the program.

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 correspondent `await` statements that were invoked in previous reaction chains.

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 memory-bounded programs that preclude stack overflows.

Figure 6 shows how the dissemination trail from our port of the DRIP protocol to CÉU can be invoked from different parts of the program, just like subroutines. The DRIP protocol distinguishes from data and metadata packets and disseminates one or the other based depending on external events. For instance, when the trickle timer expires, the program invokes `emit send=0`, which awakes the dissemination trail and starts sending a metadata packet. If the trail is already sending a packet, than the `emit` does not match the `await` and will have no effect (just like the *nesC* implementation, which uses a explicit state variable to achieve this behavior).

Internal events also provides means for describing more elaborate control structures, such as exceptions. The code in Figure 7 handles incoming packets for the CC2420 radio driver in a loop (lines 3-17). After awaking from a new packet notification (line 4), the program enters in a sequence of (lines 8-17) to read the bytes from the hardware buffer. If any anomaly is found on the received data, the program invokes `emit next` to discard the current packet (lines 10,14). Given the stacked execution for internal events, the `emit` invocation is stacked and the trail in line 6 awakes, terminates, and kills the whole `par/or` in which the emitting trail is blocked. This way, the continuation for the `emit` never resumes, and the loop restarts to await the next packet.

3.7 Code reentrancy

Applications frequently require multiple instances of an abstraction to coexist during runtime. As an example, to

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

Figure 7. The `emit` raises an exception caught by the `await`.

```
event _fwd_t go;
loop do
  await SRP_START;
  par/or do
    await SRP_STOP;
  with
    var Forwarder[COUNT] fwds;
    <initialization>
    loop do
      await SRP_RECEIVE;
      <receive or forward>
      if hops_left > 0 then
        <...>
        emit go=&fwd;
      end
    end
  end
end
```

```
class Forwarder with
  <...>
do
  loop do
    fwd = await global:go;
    if fwd:gotcha then
      continue;
    end
    fwd:gotcha = 1;
    <send message>
    <...>
  end
end
```

Figure 8. SRP forwarders as organisms.

keep track of multiple dissemination values, an application may create multiple instances of the DRIP protocol which, in turn, each requires a local instance of a trickle timer.

Code reentrancy is a technique to avoid duplicating code to save ROM: the same code is shared among instances, which only differ on their data and point of execution.

In traditional multi-threaded systems, code reentrancy is achieved with function declarations that are executed with different stacks and instruction pointers. In object oriented languages, a *class* encapsulates methods and properties, and can be instantiated with objects.

In CÉU, we designed an hybrid approach, which combines both ideas in the so called *organisms*. An organism class is composed of an *interface* and a *body*. The interface exposes public variables and internal events that other organisms (and the top-level body) can refer to. The body of a class has access to all presented functionality provided by CÉU, such as parallel compositions, *C* calls, timers, etc. An organism is instantiated by declaring a variable of the desired class, and its body is automatically spawned in a `par/or` with the enclosing block. A method can be simulated by exposing an internal event in the interface of the class and using the same technique of Figure 6.

Figure 8 shows part of our port of the SRP routing protocol [1] to CÉU. The protocol specifies a fixed number of *forwarders* responsible for routing received messages to neighbours based on a static table. Given that a forwarder holds internal state (i.e. a message buffer and the forwarding activity), we define a `Forwarder` class and create multiple instances to serve requests.

The first column of Figure 8 shows the receiving loop of the protocol, which invokes `emit go` when a message needs to be forwarded. The event is declared as global, so that the `Forwarder` class has access to it. The forwarders are declared in a vector, creating `COUNT` different instances. As the vector is local, all instances are automatically killed when the protocol is stopped. Note the use of the start/stop pattern of Figure 2 again.

The second column of Figure 8 shows the `Forwarder` class. Initially, all forwarders are in the same state, wiating for the global event `go`. Once the receiving loop emits the event, the forwarders awake in the order they were declared. The first forwarder atomically sets the `gotcha` variable, indicating that the message will be handled. All other forwarders will await again for the next `go` emission. With this technique, we eliminated the need of an explicit queue. In the case that all forwarders become busy, the `go` emission will be missed (with `gotcha=0`), acting just like a full queue.

Note that CÉU organisms are not global entities and do not use the heap for memory. Instead, they are bounded to the scope they are declared, and all memory is statically allocated, just like CÉU does for standard local variables. Also, when an organism goes out of scope, the same automatic bookkeeping of `par/or` compositions holds, all internal trails are killed and finalization blocks execute (if any). Hence, the "garbage collection" for both the memory and code in organisms is efficient and static. Although CÉU does not support dynamic creation (which could lead do unbounded memory), scoped organisms offer some degree of flexibility when compared to global objects only [22, 3].

3.8 Implementation

4 Evaluation

In order to evaluate the applicability of CÉU in the context of WSNs, we ported a number of pre-existing protocols and system utilities in the TinyOS operating system [15], which are all written in *nesC* [12]. We chose *nesC* in our evaluation given the availability and maturity of open-source implementations, and because it is also used as the base of comparison in many related works to CÉU ([11, 16, 4, 3]).

We ported the following applications to CÉU[3]: the *Trickle* timer [20]; the receiving component of the *CC2420* radio driver [1]; the *DRIP* dissemination protocol [1]; the *SRP* routing protocol [1]; the routing component of the *CTP* collection protocol [13].

We took advantage of the component-based model of TinyOS and all of our ports use the same interface provided by the *nesC* counterpart— changing from one implementation to the other is done by changing a single file. This

[3]The source code for both CÉU and *nesC* can be found at `github.com/xxx/`.

		Resource usage				Code complexity				Céu features					
Application	Language	ROM	*Céu vs nesC*	RAM	*Céu vs nesC*	tokens	*Céu vs nesC*	globals		loc	cls	par	evt	tmr	trl
								state	data						
Trickle	nesC	3894	28%	114	39%	477	-68%	2	2	2;5	1	3	-	2	6
	Céu	4974		158		155		-	-						
DRIP	nesC	13296	10%	415	17%	342	-23%	2	1	4	1	1	1	-	5
	Céu	14572		487		264		-	-						
SRP	nesC	12266	18%	1252	-3%	418	-30%	2	8	2;2;2;-	2	1	1	-	3
	Céu	14532		1213		291		-	4						
CTP	nesC	27712	6%	3281	1%	383	-21%	4	5	2;5;6	-	5	2	3	12
	Céu	29502		3311		303		-	2						
CC2420	nesC	12062	3%	379	1%	590	-24%	1	2	3;3	-	2	1	-	4
	Céu	12390		381		447		-	-						

Figure 9. Comparison between CÉU and *nesC* for the ported applications.

way, we could also use existing test applications available in the TinyOS repository (e.g. *TestDissemination*, *TestNetwork*, etc.). Furthermore, retaining the same architecture made easier to mimic the functionality between the implementations.

Figure 9 shows the comparison of *resource usage* and *code complexity*, which are discussed in detail in Sections 4.1 and 4.2, respectively. We also detail which features of CÉU have been used in the ports, for a more qualitative discussion. Note that all proposed features appear in at least two of the ported applications.

4.1 Resource usage

4.2 Code complexity

To measure the code complexity between the implementations in CÉU and *nesC*, we used the number of tokens present in the source code as the main metric. We chose to use tokens instead of lines of code because the code density is considerably lower in CÉU, given that most lines are composed of a single block delimiter from a structural composition. 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, which are not present in CÉU (e.g. `signal`, `call`, `command`, etc.) for a fair comparison.

Similarly to comparisons from related works [4, 11], we did not consider code shared among the implementations, as they do not represent control functionality and pose no challenges regarding concurrency aspects (e.g. predicates, packet accessors, etc.).

Figure 9 shows a considerable decrease in the number of tokens for all ported applications (from 20% up to 70%).

As a second metric, we counted the number of global variables used in the implementations. The globals were categorized 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 regarded as a difficult task, hence, reduction of state variables has already been proposed as a metric of code complexity in a related work [11]. In the CÉU implementations, state variables were reduced to zero, as all control patterns could be expressed with hierarchical compositions of activities.

Data variables in WSNs are usually used to 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[4]. Although the use of local variables does not incur in reduction of lines of code (or tokens), we believe that 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 *locals* in Figure 9 shows the depth of each global that became a local variable in CÉU (where depth=0 is a global variable).

The columns below *Céu features* in Figure 9 point how many times each functionality has been used in the ports, helping on identifying where the reduction in complexity comes from. As an example, the *Trickle* timer is abstracted as a class that uses 2 timers and 3 parallel compositions, with at most 6 trails active at the same time (a `par` may spawn more than 2 trails).

4.3 Safety

* static analysis

given that the porting process was straghtforward, we believe that the *nesC* implementation is free of such ... However, . another endorsement

- exception srp queue we know exactly the place where it occurs nesC no

* cancel
* timers
* locals

[4] 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 the component).

Language		composition			memory		safety			
name	year	seq	par	int	reent	locals	fin	shared	bounded	det
Esterel	1983	yes	yes	no[1]	?	?	no	no	yes[2]	no
Preemptive	many	yes	no	?	yes	yes	no	no	no	no
nesC [ref]	2003	no	no	no	yes	no	no	no	no	yes[3]
OSM [ref]	2005	no	yes[4]	no[1]	yes	yes	no	no	?	no
Virgil [ref]	2006	no	no	no	yes	no	no	no	no	?
Protothreads [ref]	2006	yes	no	no	no	no	no	no	no	yes[3]
Sol [ref]	2007	yes	yes	no	?	yes	no	?	yes[2]	yes[3]
Ocram [ref]	2013	yes	no	no	yes	yes	no	no	no	yes[3]
Céu[5]	2012	yes	yes	yes	yes	yes	yes	yes	yes[2]	yes

Languages are sorted by the year they first appeared in a paper.
Esterel is not targeted at WSNs but is included for its inflluence on many related works.
"Preemptive" represents all languages with non-deterministic preemptive schedulers.
Works with cells containing a "?" were not clear about this aspect.
Cells with gray background indicate where a Céu feature first appeared with a similar semantics (in the context of WSNs).
When they appear in the Céu row, they indicate that the feature is one of our contributions.

seq	sequential composition
par	parallel composition
int	special internal events
reent	memory reentrancy for threads
locals	optimal static allocation for locals
fin	finalization for local scopes
shared	reliable lock-free shared-memory concurrency
bounded	bounded execution for reactions
det	deterministic scheduling

[1] Esterel and OSM provide internal events with similar semantics of external events.
[2] The bounded-execution garantees are not extended to calls for the host language (e.g. C).
[3] Timers started in parallel depend on non-deterministic timings from internal reactions.
[4] OSM provides parallel compositions derived from hierachical state machines (not from Esterel).
[5] Open source release of Céu v0.1 (www.ceu-lang.org)

Figure 10. Table of features found in related works to CÉU.

5 Related work

(See Figure 10.)

6 Conclusion

7 References

[1] TinyOS TEPs. http://docs.tinyos.net/tinywiki/index.php/TEPs.

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

[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 *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993.

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

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

[8] 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.

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

[10] 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.

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

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

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

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

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

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

[17] 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.

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

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

[20] 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.

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

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