Safe System-level Concurrency for Resource-Constrained Motes

Abstract

Research in Wireless Sensor Networks conducts a continuous effort to reduce programming complexity without introducing significant overheads. Multi-threaded and synchronous languages offer proper support for sequential and structured programming, overcoming the difficulties of (still) prevailing event-driven development. However, safety aspects in concurrent and control-intensive applications is not taken into a greater extent in current designs, and most analysis and mitigation efforts are delegated to programmers.

We present a system language design that prioritizes safe concurrency, going beyond only *providing* mechanisms, but rather *enforcing* handling threats at compile time. CÉU supports concurrent lines of execution that run in time steps and are allowed to share variables. However, its synchronous and static nature enables a compile-time analysis that ensures deterministic and memory-safe programs, even in the presence of pointers and low-level system calls. Our proposed analysis pervades all language functionality, such as parallel compositions, first-class timers, and internal event communication. We successfully ported existing network protocols and a radio driver to CÉU, showing the applicability of our design in highly-constrained platforms.

1 Introduction

System-level development for WSNs basically consists of abstracting access to the hardware and designing specially tweaked network protocols [15, 3] to be further integrated as services in higher-level applications or macro-programming systems [22]. The external environment plays an important, as it interacts with applications permanently (and concurrently) by issuing events that represent expiring timers, messages arrivals, sensor readings, etc. If (hard) timeliness cannot be met by WSNs applications, given their networked nature, it is still paramount to have a reliable programming environment, pushing to compile-time as much safety guarantees as possible [19].

Three major programming models have been proposed for system-level development in WSNs: the *event-driven*, *multi-threaded*, and *synchronous* models. In event-driven programming [15, 11], 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 known to be difficult to program [2, 12]. Multi-threaded systems emerged as an alternative, providing traditional structured programming for WSNs (e.g. [12, 8]). However, the development process still requires manual synchronization and bookkeeping of threads (e.g. create, start, and rejoin) [18]. Synchronous languages [4] have also been successfully adapted to WSNs and offer higher-level compositions of activities, considerably reducing programming efforts [16, 17].

Despite the increase in development productivity, languages still lack effective safety guarantees to build complex concurrent applications. As an example, shared memory is widely used as a low-level communication mechanism, but current languages do not go beyond atomic access guarantees, either explicitly through synchronization primitives [8, 21], or implicitly with cooperative scheduling [16, 13]. Requiring manual synchronization leads to potential safety hazards [18], while implicit synchronization is no less questionable, as it assumes that *all* accesses are potentially dangerous. The bottom line is that existing languages cannot detect and enforce atomicity only when they are required. Furthermore, system-level development typically involves accesses to the underlying platform through *C* system calls that hold pointers for some time (e.g. transmission of a message buffer). Hence, programs can potentially have a *dangling pointer* if the referred memory is a local variable that goes out of scope.

In this work, we present the design of CÉU[1], a synchronous system-level programming language based on Esterel [9] that provides a reliable yet powerful programming environment for WSNs. We explore the precious control information that can be inferred from thread compositions at compile time in order to embrace safety aspects to a greater extent. Currently, we focus only on *concurrency safety*, rather than on *type safety* [10].[2]

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

[2] We consider both safety aspects to be complimentary and orthogonal,

```
/*********** 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(&timeout, 60000);
  PT_INIT(&blink);
  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_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();
<...> // CONTINUE
```

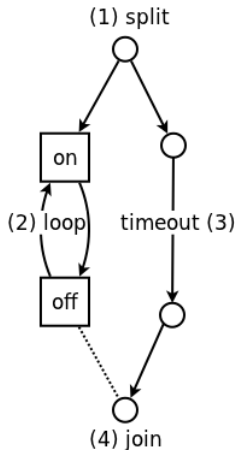

Figure 1. "Blinking LED" in nesC [13], Protothreads [12], and CÉU. (TODO: point (1..4) in the impls.)

Our overall contribution is the careful design of concurrent abstractions that can detect safety threats at compile time, enforcing the rewriting of suspicious code. As more specific contributions, we enumerate the following features:

- A compile-time analysis for 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.
- Stack-based inter-thread communication, which provides a restricted (but safer) form of subroutines.
- An object system that provides code reentrancy and enables the creation of higher-level abstractions.

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 [24], 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 and discussing safety aspects for each relevant language feature. In Section 4, we describe how we ported to CÉU some open standards of system-level protocols [1], comparing some quantitative aspects with *nesC* (e.g. memory usage and tokens count).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, a program must keep track of them in a logical fashion, in accordance with its specification. From a control perspective, the logical relations among events follow two main patterns: *sequential*, i.e., program 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 timeout to interrupt the activity consists of a parallel pattern.

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 overall 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 timeout.

The first implementation, which represents the *event-driven* model [13, 11], 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 has little structure, 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 timeout handler needs specific knowledge about how to stop the blinking activity manually (`T1.cancel()`).

The second implementation, which represents the *multi-threaded* model [12, 8], 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

i.e., type-safety annotations could also be applied to CÉU.

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

after the timeout.

The third implementation, in CÉU, which represents the *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. 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 structure of CÉU 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` (line 1) represent the external interface of the driver with a client application (e.g. a protocol). The driver enters the top-level loop (lines 2-11) 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, 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* [7] because the two sides in the composition need not to be tweaked with synchronization primitives or state variables in order to be affected by the other side in parallel. It is known that traditional multi-threaded languages cannot express thread termination safely [7, 23], thus being incompatible with a `par/or` construct.

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 [24]. 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 appears first, the loop restarts and turns on the LED for the last time. Then, the 1-minute timeout is scheduled, kills the whole `par/or`, and turns off the LED. This reasoning is reproducible in practice, and the LED will light on exactly 21 times for every execution of this program. First-class timers are discussed in more depth in Section 3.5. Note that this static inference cannot be easily extracted from the other implementations of Figure 1, specially considering the presence of timers.

The described behavior for the last iteration of the loop is known as *weak abortion*, because the blinking trail had the chance to execute for the last time. By inverting the two trails, the `par/or` would terminate immediately, and the blinking trail would not execute, qualifying a *strong abortion* [7]. CÉU not only provides means to choose between weak and strong abortion, but also detects the two conflicting possibilities and issues a warning at compile time (to be discussed in Section 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 [9], 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 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 trails segments execute during the same reaction chain. A trail segment is a sequence of statements separated by an `await`.

In the 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, 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. For each variable access, the analysis algorithm holds the set of all possible preceding `await` statements. Then, the sets for all accesses in parallel trails are compared

```
 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.

to assert that no `await` statements are shared. Otherwise the compiler warns 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 algorithm, as the assignments in parallel can only occur in different reactions to *A* (lines 5 and 9).

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 nested trail in parallel, issuing a warning otherwise.

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. Such information cannot be inferred when the program control relies on state variables.

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

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 gen-

```
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.

eration 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 presented in 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 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 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 safe
    _Leds_led0Toggle with _Leds_led1Toggle;
int*  buf1, buf2;  // point to different buffers
deterministic      // 'buf1' vs 'buf2' is safe
    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 provide means to circumvent the rigor of CÉU in a well-marked way (the special underscore syntax). Evidently, programs should only recur to *C* for I/O operations that are assumed to be instantaneous, 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, given that they are never active at the same time.

The syntactic compositions of trails allows the CÉU compiler to statically allocate and optimize memory usage [17]: memory for trails in parallel must coexist; trails that follow rejoin points reuse all memory.

However, the unrestricted use of locals 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, leading to a dangling pointer.

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 (protocol/radio stop, and local/network resend 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), but its reference is manipulated by two *TinyOS* global 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. As the radio driver runs asynchronously and holds a reference to the message until it is completely transmitted, the sending trail may be killed in the meantime, resulting in a dangling pointer in the program. A possible solution is to include a call to `AMSend_cancel` in all trails that kill the sending trail. However, this would require to expand the scope of the message buffer and add a state variable to keep track of the sending status, increasing considerably the complexity of the application.

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

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

First, the `nohold` annotation informs the compiler that the referred *C* function does not require finalization code because it does not hold references. Second, the `finalize` construct automatically executes the `with` clause when 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 politely requests the OS to cancel the ongoing send operation.

3.5 Wall-clock time

Activities that involve reactions to *wall-clock time*[3] appear in typical patterns of WSNs, such as sensor sampling and activity timeouts. 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 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 recur-

[3] 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 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 5. Unsafe use of local references.

rent 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.

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 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`. 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 the scheduler 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
```

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.

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

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

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. 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*. In order to ensure bounded execution, an `await` statement cannot awake on the same reaction chain it is invoked.

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 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 on external requests. For instance, when the trickle timer expires, the program invokes `emit send=1`, 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 provide 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 to read the bytes from the hardware buffer (lines 8-17). If any anomaly is found on the received data, the program invokes `emit next` to discard the current packet (lines 10,14).

```
 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`.

Given the execution semantics of internal events, the `emit` continuation is stacked and awakes the trail in line 6, which terminates and kills the whole `par/or` in which the emitting trail is paused. 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 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 as objects.

In CÉU, we designed an hybrid approach, which combines threads and classes in the so called *organisms*. An organism class is composed of an *interface* and a single *body*. The interface exposes public variables and internal events that other organisms (and the top-level body) can refer. The body of a class is equivalent to a trail, having 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.

The first column of Figure 8 re-implements the example of Figure 1 to blink two LEDs with different frequencies. The `Blink` class exposes the `led` and `freq` variables to be configured by the application, which then creates two instances and initializes them.

The second column shows how the organisms bodies are expanded to run in a `par/or` together with the enclosing block

```
C _on(), _off();

class Blink with
   var int led;
   var int freq;
do
   loop do
      _on(this.led);
      await (this.freq)s;
      _off(this.led);
      await (this.freq/2)s;
   end
end

var Blink b1;
b1.led  = 0;
b1.freq = 2;

var Blink b2;
b2.led  = 1;
b2.freq = 4;

await 1min;
```

```
C _on(), _off();
var Blink b1, b2;
par/or do
   b1.led  = 0;
   b1.freq = 2;
   b2.led  = 1;
   b2.freq = 4;
   await 1min;
with
   loop do
      _on(b1.led);
      await (b1.freq)s;
      _off(b1.led);
      await (b1.freq/2)s;
   end
   await FOREVER;
with
   loop do
      _on(b2.led);
      await (b2.freq)s;
      _off(b2.led);
      await (b2.freq/2)s;
   end
   await FOREVER;
end
```

Figure 8. "Two blinking LEDs" using organisms.

body. The expansion is illustrative, i.e., the code is not duplicated. Note that in the expansion, the bodies of the organisms are followed by `await FOREVER`[4], meaning that only the enclosing block can terminate the `par/or`. Note also that the block body runs first and properly initializes the organisms before they are spawned.

Once the enclosing block terminates, declared organisms are killed and all memory can be reused, just as happens in standard parallel compositions. The allocation and deallocation of organisms is static, with no runtime overhead such as garbage collection.

Figure 9 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 9 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 `Forwarder` instances have 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 9 shows the `Forwarder` class. Initially, all forwarders are in the same state, waiting for the global event `go`. Once the receiving loop emits the event in the top-level body, the forwarders awake in the order they were declared. The first forwarder atomically sets the `gotcha` variable, indicating that the message will be handled and that other forwarders should ignore it: all other forwarders will

[4] `FOREVER` is a reserved keyword in CÉU, and represents an external input event that never occurs.

```
event _fwd_t go;
loop do
  await SRP_START;
  par/or do
    await SRP_STOP;
  with
    var Forwarder[COUNT] fwds;
    <initialize fwds>
    loop do
      await SRP_RECEIVE;
      <receive or forward>
      if hops_left > 0 then
        <prepare fwd>
        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 9. SRP forwarders as organisms.

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.

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* [13]. 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 ([12, 16, 6, 5]).

We ported the following applications to CÉU[5]: 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 [14].

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 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 10 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 [6, 12], 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 10 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 [12]. In 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[6]. 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 CÉU implementations, most variables could be nested to a deeper scope. The column *locals* in Figure 10 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 10 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

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

* 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

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

[6]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).

		Resource usage				Code complexity				Céu features					
Application	Language	ROM	*Céu vs nesC*	RAM	*Céu vs nesC*	tokens	*Céu vs nesC*	globals		locals	classes	int evts	timers	pars	trails
								state	data						
Trickle	nesC	3894	28%	114	39%	477	-68%	2	2	2;5	1	-	2	3	6
	Céu	4974		158		155		-	-						
CC2420	nesC	12062	3%	379	1%	590	-24%	1	2	3;3	-	1	-	2	4
	Céu	12390		381		447		-	-						
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	-	2	3	5	12
	Céu	29502		3311		303		-	2						

Figure 10. Comparison between CÉU and *nesC* for the ported applications.

* cancel
* timers
* locals

- automatically - timer, loops, recursion - C is easily trackable - manually - shared memory - calls - abortion - finalization

5 Related work

Figure 11 presents an overview of related works to CÉU, pointing support for specific features XXX, which are discussed individually. Although Esterel [9] is not targeted at WSNs, we included it given its strong influence on CÉU and many related works. The second line, *Preemptive*, represents all multi-threaded languages with preemptive scheduling [8, **?**, **?**]. The remaining lines enumerate languages with similar goals of CÉU that also follow a synchronous or cooperative execution semantics (they are sorted by the date they first appear in a publication).

The three columns under *Composition* indicate which languages provide abstractions to deal with *sequential* and *parallel* compositions of activities, as well as *internal communication* among them. Esterel first introduced hierarchical compositions of processes, which first appeared in the context of WSNs in Protothreads [12] and Sol [16] (indicated by the gray background). CÉU differs from these on the stack-based semantics for internal events, which is essential to bring other control mechanisms, such as exceptions, as discussed in Section 3.6.

The two columns under *Memory* list language support for *code reentrancy* and *local scopes* for variables, which account for saving ROM and RAM, respectively. Both properties are easily achieved in traditional multi-threaded systems, in which threads share global functions as bodies, with each holding a different stack. However, this approach is not resource-efficient, given that the compiler cannot infer when each thread is active, enforcing all to be statically allocated [8, 6] (even if some of them are never active at the same time). *nesC* provides compile-time instantiable components, which resemble classes from object-oriented programming. As discussed in Section 3.7, CÉU organisms introduce an hybrid approach, providing TODO. CÉU support for local scopes is very similar to how OSM [17] does.

TODO Esterel derived cannot offer the analysis

TODO Although CÉU does not support dynamic creation (which could lead do unbounded memory), scoped organisms offer some degree of flexibility when compared to systems providing global objects only [25, 5].

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

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

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

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

[7] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993.

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

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

[10] 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.

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

Language		Composition			Memory		Safety				
name	year	seq	par	int	reent	locals	fin	shared	det	bounded	rt
Esterel	1983	yes	yes	yes[1]	?	?	no	no	no	yes[2]	?
Preemptive	many	yes	no	yes	yes	yes	no	no	no	no	yes
nesC [ref]	2003	no	no	no	yes	no	no	no	yes[3]	no	yes
OSM [ref]	2005	no	yes	yes[1]	yes	yes	no	no	no	?	no
Virgil [ref]	2006	no	no	no	yes	no	no	no	?	no	no
Protothreads [ref]	2006	yes	no	no	no	no	no	no	yes[3]	no	no
Sol [ref]	2007	yes	yes	no	?	?	no	no	yes[3]	yes[2]	no
FlowTalk [ref]	2011	yes	no	no	no	yes	no	no	no	no	no
Ocram [ref]	2013	yes	no	no	yes	yes	no	no	yes[3]	no	no
Céu		yes	yes	yes	yes	yes	yes	yes	yes	yes[2]	no

([1]) Esterel and OSM provide internal events, but with a similar semantics of external events.
([2]) The bounded-execution garantees are not extended for calls to the host language (e.g. C functions).
([3]) Timers started in parallel depend on non-deterministic timings from internal reactions.
(?) We consider the feature to be feasible (given the language design), but support was not clear in the reference.

Figure 11. Table of features found in related works to CÉU.

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

[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] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 167–180, New York, NY, USA, 2006. ACM.

[22] L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43:19:1–19:51, April 2011.

[23] ORACLE. Java thread primitive deprecation. `http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`, 2011.

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

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