Safe System-level Concurrency for Resource-Constrained Motes

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

Noemi Rodriguez
noemi@inf.puc-rio.br

Roberto Ierusalimschy
roberto@inf.puc-rio.br

Departamento de Informática – PUC-Rio, Brazil

Olaf Landsiedel
olafl@chalmers.se

Philippas Tsigas
tsigas@chalmers.se

Computer Science and Engineering – Chalmers University of Technology, Sweden

Abstract

Despite the continuous research in facilitating programming WSNs, most safety analysis and mitigation efforts in control and concurrency are still left to programmers, who must explicitly manage synchronization and shared memory. We present a system language that ensures safe concurrency by enforcing the handling of threats at compile time, rather than at runtime. On the one hand, the synchronous and static foundation of our design allows for a simple reasoning about concurrency that enables a compile-time analysis to ensure deterministic and memory-safe programs. On the other hand, this design imposes limitations on the language expressiveness, such as for doing computation-intensive operations and meeting hard real-time responsiveness. Nevertheless, we implemented a number of standard network protocols and a radio driver in order to show that the achieved expressiveness and responsiveness is sufficient for a wide range of WSNs applications. Besides the ensured safety properties, the implementations show a reduction around 25% in code complexity, with a penalty of memory increase below 10% in terms of total ROM and RAM needs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Reliability

Keywords

Concurrency, Determinism, Safety, Static Analysis, Synchronous, Wireless Sensor Networks

1 Introduction

System-level development for WSNs basically consists of abstracting access to the hardware and designing specially tweaked network protocols [21, 3] to be further integrated as services in higher-level applications or macro-programming systems [30]. 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 [27].

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 [21, 13], 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, 14]. Multi-threaded systems emerged as an alternative, providing traditional structured programming for WSNs (e.g. [14, 9]). However, the development process still requires manual synchronization and bookkeeping of threads (e.g. create, start, and rejoin) [26]. Synchronous languages [4] have also been successfully adapted to WSNs and offer higher-level compositions of activities, considerably reducing programming efforts [23, 24].

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 [9, 29], or implicitly with cooperative scheduling [23, 19]. Requiring manual synchronization leads to potential safety hazards [26], while implicit synchronization is no less questionable, as it assumes that *all* accesses are 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, if a local variable goes out of scope and its reference has been exposed, a system-level activity may end

```
/*********** 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
```

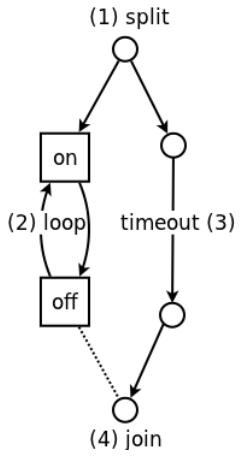

Figure 1. "Blinking LED" in nesC [19], Protothreads [14], and CÉU. (TODO: point (1..4) in the impls.)

up with a dangling pointer (i.e. a pointer to freed memory).

In this work, we present the design of CÉU[1], a synchronous system-level programming language that provides a reliable yet powerful set of abstractions for the development of control-intensive WSNs applications. CÉU is based on Esterel [10] and introduces the following new safety mechanisms: *first-class timers* to ensure that timers started in parallel remain synchronized (not depending on internal reaction timings); *finalization blocks* for local pointers going out of scope; and a *stack-based event-driven communication* that avoids cyclic dependencies. Our overall contribution is a static analysis that considers all language mechanisms and detects safety threats at compile time, such as concurrent accesses to shared memory, and concurrent termination of timers and threads. Currently, we focus only on *concurrency safety*, rather than on *type safety* [11].[2]

As a inherent limitation of CÉU's synchronous model, computations that run in unbounded time (e.g., compression, image processing) do not fit the zero-delay reaction hypothesis [32], and cannot be elegantly implemented. Also, the static analysis precludes any dynamic support in the language, such as memory allocation and dynamic loading. That said, we re-implemented in CÉU the *CC2420* radio driver, and the *DRIP*, *SRP*, and *CTP* network protocols [1]. Our evaluation shows that the expressiveness of CÉU is not only sufficient for the context of WSNs, but actually reduced the code complexity in 25%, with an increase in ROM and RAM consumption below 10% in comparison to *nesC* [19].

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, motivating and discussing the safety aspects of each relevant language feature. Section 4, evaluates the implementation of the network protocols in CÉU and compares some aspects with *nesC* (e.g. memory usage and tokens count). We also evaluate the responsiveness of the radio driver written in CÉU. Section 5 discusses related work 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 [19, 13], 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

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

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

needs specific knowledge about how to stop the blinking activity manually (`T1.cancel()`).

The second implementation, which represents the *multi-threaded* model [14, 9], 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 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, 31], thus being incompatible with a `par/or` construct.

```
 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.

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 [32]. 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 [10], CÉU requires that each possible path in a loop body contains at least one `await` or `break` statement, thus ensuring that loops never run in unbounded time. Consider the examples that follow:

```
loop do
    if <cond> then
        break;
    end
end
```

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

The first example is refused at compile time, because the `if` true branch may never execute, resulting in a *tight loop* (i.e., an infinite loop that does not await). The second variation is accepted, because for every iteration, the loop either breaks or awaits.

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

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

```
 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.

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 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, the simpler static analysis executes in negligible time for all implementations to be presented in Section 4 and does not detect any false positives, suggesting that the algorithm is practical.

```
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.

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 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.

```
 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.

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 [24]: 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 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.

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.

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

```
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`

```
 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`.

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). 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.

4 Evaluation

In order to evaluate the applicability of CÉU in the context of WSNs, we re-implemented a number of protocols and system utilities originally written in *nesC* [19] from the TinyOS operating system [21]. We chose *nesC* in our evaluation given its resource efficiency, code base maturity, and because it is used as benchmark in many related works to CÉU (e.g. [14, 23, 6, 5]).

Our evaluation consists of the following implementations: the *Trickle* timer; the receiving component of the *CC2420* radio driver; the *DRIP* dissemination protocol; the *SRP* routing protocol; and the routing component of the *CTP* collection protocol. They represent well the realm of system-level development for WSNs, which mostly consists of network protocols and lower level utilities to be used as services in applications. They are also intensive in control and concurrency aspects, being perfect targets to CÉU. Finally, they are open standards [1], with open implementations[4], and are academically recognized [28, 20].

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 swapping a single file. This way, we could use existing test applications available in the TinyOS repository (e.g. *RadioCountToLeds* and *TestNetwork* [4]).

Figure 8 shows the comparison for *resource usage* and *code complexity* between *nesC* and CÉU, which are discussed in Sections 4.1 and 4.2, respectively. The figure also details how many times each relevant feature of CÉU was used in the implementations (i.e., local variables, internal events, first-class timers, parallel compositions, and maximum number of concurrent trails).

We also evaluate the performance of CÉU with respect to responsiveness, i.e., its capacity to promptly acknowledge requests from the environment, such as radio packets arrivals. Section 4.3 presents two experiments that measure the responsiveness of the implemented radio driver under high loads.

4.1 Memory usage

Memory is a scarce resource in WSNs motes and it is important that CÉU does not pose significant overheads in comparison to *nesC*. We evaluate ROM and RAM consumption by using the simplest available test applications for the ported implementation. We tweaked each application to remove extra functionality so that the generated binary is dominated by the component of interest. Them, we compiled each application twice: one with the original component in *nesC*, and another with the ported component in CÉU. Figure 4 shows in the column *Memory usage* the consumption of ROM and RAM for the generated applications. With the exception of the Trickle timer, the results in CÉU remains below 10% in ROM and 5% in RAM in comparison with the implementations in *nesC*. Our method and results are similar to those for Protothreads [14], which is used extensively in Contiki, and has a simple and lightweight implementation based on a set of C macros.

The Trickle timer illustrates the footprint of the runtime of CÉU. The RAM overhead of 20% actually corresponds to only 16 bytes: 1 byte for each of the maximum 6 concurrent trails, and 10 bytes to handle synchronization among timers. As the complexity of the application grows, this overhead tends to discharge. The SRP implementation could eliminate a queue by signaling events internally, hence the decrease in RAM. Note that both TinyOS and CÉU define functions to manipulate queues for tasks (or trails) and timers. Hence, given that our implementations mix components in the two systems, we pay a considerable overhead in ROM.

We focused most of our implementation efforts on RAM optimization, as it has been historically considered more scarce then ROM [27]. Although, we have achieved competitive results, we expected more gains with memory reuse for blocks in sequence, because it is something that cannot be done automatically by the *nesC* compiler. However, we carefully analyzed each ported application and it turned out that we had no gains *at all* from blocks in sequence. Our conclusion is that sequential patterns in WSNs applications either come from split-phase operations, which always require to preserve memory; or from loops, which do reuse all memory, but in the same way event-driven systems do.

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

			Resource usage				Code complexity				Céu features				
Component	Application	Language	ROM	Céu vs nesC	RAM	Céu vs nesC	tokens	Céu vs nesC	globals		locals	int evts	timers	pars	trails
									state	data					
CTP	*TestNetwork*	nesC	18896	9%	1295	2%	383	-21%	4	5	2;5;6	2	3	5	12
		Céu	20660		1323		303		-	2					
SRP	*TestSrp*	nesC	12266	5%	1252	-3%	418	-30%	2	8	2;2;2;-	1	-	1	3
		Céu	12834		1215		291		-	4					
DRIP	*TestDissemination*	nesC	12708	8%	393	4%	342	-25%	2	1	4	1	-	1	5
		Céu	13734		407		258		-	-					
CC2420	*RadioCountToLeds*	nesC	10546	2%	283	2%	519	-27%	1	2	3;3	1	-	2	4
		Céu	10776		289		380		-	-					
Trickle	*TestTrickle*	nesC	3504	22%	72	22%	477	-69%	2	2	2;5	-	2	3	6
		Céu	4284		88		149		-	-					

Figure 8. Comparison between CÉU and *nesC* for the ported applications.

4.2 Code complexity

We used two metrics to compare code complexity between the implementations in CÉU and *nesC*: the number of tokens and the number of global variables used in the source code. Similarly to comparisons from related works [6, 14], we did not consider code shared among the implementations, as they do not represent control functionality and pose no challenges regarding concurrency aspects (i.e. they are predicates, `struct` accessors, etc.).

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 (e.g. `signal`, `call`, `command`, etc.) for a fair comparison. Figure 8 shows a considerable decrease in the number of tokens for all implementations (from 20% up to 70%).

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 [14]. The implementations in CÉU completely eliminated state variables, and 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[5]. 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 8 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 8 point out 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 uses 2 timers and 3 parallel compositions, resulting in at most 6 trails active at the same time (a `par` may spawn more than 2 trails). Six trails for such a small application is justified by its highly control-intensive nature, and the almost 70% code reduction illustrates the huge gains with CÉU in this context.

4.3 Responsiveness

A known limitation of languages with synchronous and cooperative execution is that they cannot meet hard real-time deadlines [12, 25]. For instance, the rigorous synchronous semantics of CÉU forbids non-deterministic preemption to serve high priority trails. This way, the implementation of a radio driver purely in CÉU raises questions regarding its responsiveness and we conducted two experiments in order two evaluate it. Even though WSNs protocols are usually tolerant to faults (given the unreliable link), packet losses affect the network performance and cause motes to spend more battery.

In the first experiment[6], we performed a stress test on the radio driver to compare its performance in the CÉU and *nesC* implementations. We use 10 motes that broadcast 100 consecutive messages of 20 bytes to a mote that runs a periodic time-consuming activity. The receiving handler simply adds the value of each received byte to a global counter. The sending rate of each mote is 200ms (leading to an average of 50 messages per second considering the 10 motes), and the activity in the receiving mote runs every 140ms. We run the experiment varying the duration of the lengthy activity from 0 to 128 milliseconds. We assume that the lengthy operation is implemented directly in C and cannot be easily split in smaller operations (e.g. recursive algorithms [12, 25]). This way, we simulated it with a simple busy wait that will keep

[5] 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).

[6] The experiments use the *COOJA* simulator [15] running images compiled to *TelosB* motes.

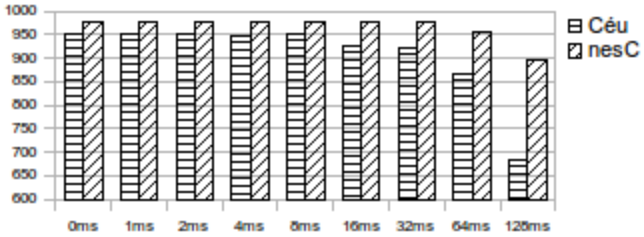

Figure 9. Packet receives under high-loads for CÉU and *nesC*.

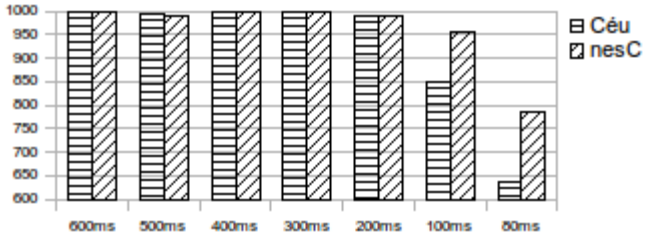

Figure 10. Packet receives under high-loads for CÉU and *nesC*.

Operation	Duration
Block cypher [22, 18]	1ms
MD5 hash [18]	3ms
Wavelet decomposition [36]	6ms
SHA-1 hash [18]	8ms
RLE compression [34]	70ms
BWT compression [34]	300ms
Image processing [33]	50–1000ms

Table 1. Durations for lengthy operations is WSNs.

the driver in CÉU unresponsive, which is our objective.

Figure 9 shows the total number of handled messages in CÉU and *nesC* for each configuration. Starting from operations that take 16ms, the performance of CÉU drops below 5% in comparison to *nesC*. Our direct interpretation is that the CÉU driver starts to become unresponsive when receiving messages every 20ms while executing a periodic operation of 16ms. The same interpretation can be done for *nesC*, but considering a 64-ms operation, which is a similar conclusion taken from TOSThreads experiments (25-bytes messages every 50ms, while running a 50-ms operation [25]). Table 1 shows the durations for some lengthy operations found in the literature specifically designed for WSNs. The operations in the group with timings below 8ms could be used with real-time responsiveness with CÉU, considering the proposed configuration parameters.

Although we did not perform specific tests to evaluate CPU usage, the first experiment suggests that the overhead of CÉU over *nesC* is lower than 3%, based on the packet losses for the CPU awake full time. For lengthy operations implemented in C, there is no overhead, as the generated code is the same for CÉU and *nesC*.

In the second experiment, instead of running an activity in parallel, we use a 8-ms operation in sequence with message arrivals to simulate a process that is tied with message handling, such as encrypting and forwarding. We now run the experiment varying the rate in the 10 sending motes from 600ms to 100ms. Figure 10 shows the total number of handled messages in CÉU and *nesC* for each configuration. The last column shows the theoretical limit of 80ms, i.e., receiving a message every 8ms and handling it in 8ms. The results show that CÉU performs equally to *nesC* for a frequency up to 50 messages per second.

The overall conclusion is that CÉU can be used to write system-level drivers if other parts of the application (also in CÉU) do not execute algorithmic-intensive operations. CÉU performs equally in comparison to *nesC* for sub-millisecond reactions. The range between 1ms and 8ms is a "gray area" that offers opportunities for simple lengthy operations, but that also requires careful analysis and testing. For operations over 8ms and under high loads, CÉU will perform considerably worse than *nesC*.

4.4 Discussion

CÉU targets control-intensive applications and provides abstractions that can better express program flow specifications. Our evaluation shows a considerable decrease in complexity that comes from logical compositions of trails through `par/or` and `par/and` constructs. They handle startup and termination for trails seamlessly without extra programming efforts. The small overhead in memory qualifies CÉU as a realistic option for constrained devices.

The main aspect of CÉU is to ensure that the high degree of concurrency in WSNs does not pose safety threats to applications. Restricting the expressiveness of the language enabled a broad safety analysis in programs at compile time. The analysis encompasses all proposed concurrency mechanisms, such as parallel compositions, first-class timers, and communication via events. As summary, the following safety properties hold for all programs that successfully compile in CÉU:

- Time-bounded reactions to the environment (Sections 3.1 and 3.6).
- Reliable weak and strong trail abortion (Sections 3.1 and 3.2).
- No concurrency in variable accesses (Section 3.2).
- No concurrency in system calls sharing a resource (Section 3.3).
- Finalization for blocks going out of scope (Section 3.4).
- Auto-adjustment for timers in sequence (Section 3.5).
- Synchronization for timers in parallel (Section 3.5).

These properties are desirable in any application and are guaranteed as preconditions in CÉU by design. Ensuring or even extracting these properties from unrestricted languages is not possible without significant manual analysis.

Even though the achieved expressiveness and overhead of CÉU meet the requirements of WSNs, its design imposes two inherent limitations: the lack of dynamic primitives that would forbid the static analysis, and the lack of hard real-time guarantees. Regarding the first limitation, dynamic features are already discouraged in the context of WSNs due to resource constraints. For instance, even object-oriented languages targeting WSNs forbid dynamic allocation [5, 35].

To deal with the second limitation, which can be critical in the presence of lengthy computations, we can consider the following approaches: (1) carefully measure reaction times to see if deadlines are met; (2) manually place `pause` statements in tight loops; (3) integrate CÉU with a preemptive system. The second option requires rewriting the lengthy computation in CÉU with `pause` statements to let other trails to be scheduled. This option is the one recommended in many related works that provide a similar primitive (e.g. `pause` [8], `PT_YIELD` [14], `yield` [23], `post` [19]). Fortunately, CÉU and preemptive threads are not mutually exclusive, and the third option is appealing to be explored in a future work. For instance, TOSThreads [25] is integrated with *nesC* via message passing, a mechanism that is safe and matches the semantics of external events in CÉU.

5 Related work

CÉU is strongly influenced by Esterel [10] on its support for compositions and reactivity to events. However, Esterel is focused on control and delegates to programmers most efforts to deal with data and low-level access to the underlying platform. For instance, read/write to shared memory among threads is forbidden and avoiding conflicts between concurrent *C* calls is left to programmers [8]. CÉU deals with shared memory and *C* integration at its very core, with additional support for finalization, conflict annotations, and a static analysis that permeates all languages aspects. This way, CÉU could not be designed easily as pure extensions to Esterel.

Figure 11 presents an overview of related work to CÉU, pointing out support for a number of features (grouped by those that reduce complexity and those that increase safety). The line *Preemptive* represents languages with preemptive scheduling [9, 25], which are summarized further. The remaining lines enumerate languages with similar goals of CÉU that follow a synchronous or cooperative execution semantics (they are sorted by the date they first appear in a publication).

Many related works allow events to be handled in sequence through a blocking primitive (column *seq*), overcoming the main limitation of event-driven systems [14, 6, 29, 5, 23]. Most of them also keep the state of local variables between reactions to the environment (column *locals*). CÉU introduces a reliable mechanism to interface local pointers with the system through annotations or finalization blocks (column *final*), as discussed in Section 3.4. TODO Regarding compile-time safety guarantees, many related works can provide deterministic and reproducible execution relying on cooperative scheduling (column *det*). However, as far as we know, CÉU is the first system to extend determinism for timers in parallel (as discussed in Section 3.5).

Synchronous languages first appeared in the context of WSNs through OSM [24] and Sol [23], which provide parallel compositions (column *par*) and distinguish from multi-threaded languages by handling thread destruction seamlessly [31, 7]. Compositions are fundamental for the simpler reasoning about control that enables the safety analysis of CÉU. Sol detects infinite loops at compile time to ensure that programs are responsive [23] (column *bounded*). CÉU adopts the same policy, which is based on Esterel. Internal events (column *int*) can be used as a reactive alternative to shared-memory communication in synchronous languages. OSM supports a queue-based mechanism based on Esterel [24], but CÉU introduces a stack-based execution that also provides a restricted but safer form of subroutines (as discussed in Section 3.6).

nesC provides a data-race detector for interrupt handlers (column *shared*): *"if a variable x is accessed by asynchronous code, then any access of x outside of an atomic statement is a compile-time error"* [19]. The analysis of CÉU is targeted at synchronous code and point more precisely when accesses can be concurrent, what is only possible given its restricted semantics. Furthermore, CÉU extends the analysis for system calls (*commands* in *nesC*) and control conflicts (as discussed in Sections 3.1 and 3.3).

On the opposite side of concurrency models when compared to CÉU, languages with preemptive scheduling assume time independence among processes and are more appropriate for applications involving algorithmic-intensive problems. Preemptive scheduling is also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [9, 16, 17, 25]. The choice between the two models should take into account the nature of the application and is a trade-off between safe synchronization and predictable responsiveness.

6 Conclusion

7 Acknowledgments

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

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

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

[11] 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.

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

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

Language		Complexity				Safety			
name	year	seq	locals	par	int	det	bounded	shared	final
Preemptive	many	✔	✔		✔		*rt*		
nesC [ref]	2003					✔	*rt*	✔	
OSM [ref]	2005		✔	✔	✔				
Protothreads [ref]	2006	✔				✔			
TinyThreads [ref]	2006	✔	✔			✔			
Sol [ref]	2007	✔	✔	✔		✔	✔		
FlowTalk [ref]	2011	✔	✔						
Ocram [ref]	2013	✔	✔			✔			
Céu		✔	✔	✔	✔	✔	✔	✔	✔

Figure 11. Table of features found in related work to CÉU. (Gray background indicates where they first appeared.) TODO CEU

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

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

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

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

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

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

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

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

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

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

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] 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.

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

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

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

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

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

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