Safe System-level Concurrency on Resource-Constrained Nodes

Accepted paper in SenSys'13 (preprint version)

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 concurrency are still left to developers, who must manage synchronization and shared memory explicitly. In this paper, we present a system language that ensures safe concurrency by handling threats at compile time, rather than at runtime. The synchronous and static foundation of our design allows for a simple reasoning about concurrency enabling compile-time analysis to ensure deterministic and memory-safe programs. As a trade-off, our design imposes limitations on the language expressiveness, such as doing computationally-intensive operations and meeting hard real-time responsiveness. We implement widespread network protocols and the CC2420 radio driver to show that the achieved expressiveness and responsiveness is sufficient for a wide range of WSN applications. The implementations show a reduction around 25% in code complexity, with a penalty of memory increase below 10% in comparison to *nesC*. Overall, we ensure safety properties for programs relying on high-level control abstractions that also lead to concise and readable code.

1 Introduction

System-level development for WSNs commonly follows three major programming models: *event-driven*, *multi-threaded*, and *synchronous* models. In event-driven programming [19, 11], each external event can be associated with a short-lived function callback to handle a reaction to the environment. This model is efficient, but is known to be difficult to program [1, 12]. Multi-threaded systems emerged as an alternative, providing traditional structured programming for WSNs [12, 7]. However, the development process still requires manual synchronization and bookkeeping of threads [24]. Synchronous languages [2] have also been adapted to WSNs and offer higher-level compositions of activities, considerably reducing programming efforts [21, 22].

Despite the increase in development productivity, WSN system languages still fail to ensure static safety properties for concurrent programs. However, given the difficulty in debugging WSN applications, it is paramount to push as many safety guarantees to compile time as possible [25]. As an example, shared memory is widely used as a low-level communication mechanism, but current languages do not go beyond runtime atomic access guarantees, either through synchronization primitives [7, 27], or by adopting cooperative scheduling [21, 17]. Requiring explicit synchronization primitives lead to potential safety hazards [24]. Enforcing cooperative scheduling 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.

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 WSN applications. CÉU is based on a small set of control primitives similar to Esterel's, leading to implementations that more closely reflect program specifications [8]. In addition, CÉU provides a disciplined policy for typical system-level functionality, such as low-level access to *C* and shared memory concurrency. It also introduces the following new safety mechanisms: *first-class timers* to ensure that timers in parallel remain synchronized (not depending on internal reaction timings); *finalization blocks* for local pointers going out of scope; and *stack-based communication* that avoids cyclic dependencies. We propose 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. Our work focus on *concurrency safety*,

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

/ nesC */*

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

/ Protothreads */*

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

/ CÉU */*

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

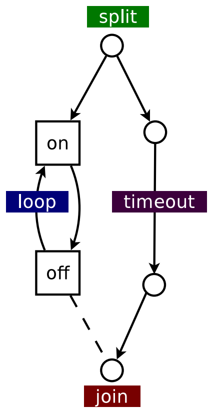

Figure 1. "Blinking LED" in nesC [17], Protothreads [12], and CÉU. The colors associate chunks of code with respective actions in the diagram.

rather than *type safety* [9].[2]

In order to enable the static analysis, CÉU programs must undergo some limitations. Computations that run in unbounded time (e.g., compression, image processing) cannot be elegantly implemented [29], and dynamic support, such as dynamic loading is forbidden. However, we show that CÉU is sufficiently expressive for the context of WSN applications. We successfully implemented the *CC2420* radio driver, and the *DRIP*, *SRP*, and *CTP* network protocols [32] in CÉU. The implementations reduced code complexity by 25%, with an increase in ROM and RAM below 10% in comparison to *nesC* [17].

The rest of the paper is organized as follows: Section 2 gives an overview on how different programming models used in WSNs can express typical control patterns. Section 3 details the design of CÉU, motivating and discussing the safety aspects of each relevant language feature. Section 4, evaluates the implementation of the network protocols in CÉU and compares some 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

WSN applications must handle a multitude of concurrent events, such as timers and packet transmissions. Although they may seem random and unrelated for an external observer, a program must logically keep track of them according to its control specification. From a control perspective, programs are composed of two main patterns: *sequential*, i.e., an activity with two or more states in sequence; and *parallel*, i.e., unrelated activities that eventually need to synchronize. As an example, an application that alternates between sampling a sensor and broadcasting its readings has a sequential pattern (with an enclosing loop); while including an 1-minute timeout to interrupt an activity denotes a parallel pattern.

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

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

The second implementation in *Protothreads*, which represents the *multi-threaded* model [12, 7], uses a dedicated thread to make the LED blink in a loop. This brings more structure to the solution. The main thread also helps a reader to identify the overall sequence of the program, which is not easily identifiable in the event-driven implementation without tracking the dependencies among callbacks. However, it still requires much bookkeeping for initializing, scheduling

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

and rejoining the blinking thread after the timeout (inside the `while` condition).

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. The `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.). We argue that the hierarchical structure of CÉU more closely reflects the control diagram and ties the two activities together, implying that (a) they can only exist together; (b) they always start together (c) they always terminate together. Besides the arguably cleaner syntax, the additional control-flow information that can be inferred from the program is the base for all features and safety guarantees introduced by CÉU.

3 The Design of Céu

CÉU is a concurrent language in which multiple lines of execution—known as *trails*—continuously react to input events from the environment. Waiting for an event halts the running trail until that event occurs. The environment broadcasts an occurring event to all active trails, which share a single global time reference (the event itself). The fundamental distinction between CÉU and prevailing multi-threaded designs is the way threads are combined in programs. CÉU provides Esterel-like syntactic hierarchical compositions, while most multi-threaded systems typically only support top-level definitions for threads. CÉU distinguishes itself from Esterel by its tight integration with *C* and support for shared memory, which also demands an effective safety analysis permeating (and affecting) all language features.

As an introductory example, the code in Figure 2 is extracted from our implementation of the *CC2420* radio driver [32] and uses a `par/or` to control the start/stop behavior of the radio. The input events `CC2420_START` and `CC2420_STOP` (line 1) represent the external interface of the driver with a client application (e.g. a protocol). The driver enters the top-level loop and awaits the starting event (line 3); upon request, the driver spawns two other trails: one to await the stopping event (line 5), and another to actually receive radio messages in a loop (collapsed in line 9). As compositions can be nested, the receiving loop can be as complex as needed and contain other loops and parallel constructs. However, once the client requests to stop the driver, the trail in line 5 awakes and terminates, making the `par/or` to also terminate and kill the receiving loop, proceeding to the statement in sequence. In this case, the top-level loop restarts, waiting for the next request to start.

The `par/or` construct is regarded as an *orthogonal preemption primitive* [5] because the two sides in the composition need not to be tweaked with synchronization primitives or state variables in order to affect each other. It is known that traditional multi-threaded languages cannot express thread termination safely [5, 28], 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.
The occurrence of `CC2420_STOP` (line 5) seamlessly kills the receiving loop (collapsed in line 9) and resets the driver to wait for the next `CC2420_START` (line 3).

3.1 Deterministic and Bounded Execution

CÉU is grounded on a precise definition of time as a discrete sequence of external input events: a sequence because only a single input event is handled at a time; discrete because reactions to events are guaranteed to execute in bounded time (to be discussed further). The execution model for a 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 [29]. Conceptually, a program takes no time on step 2 and is always idle on step 3. In practice, if a new external input event occurs while a reaction chain is running (step 2), it is enqueued to run in the next reaction. When multiple trails are active at a time (i.e. awaking 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 in CÉU 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 loop appears first, it will restart and turn 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 actually reproducible in practice, and the LED will light on exactly 21 times for every single execution of this program. First-class timers are discussed in more depth in Section 3.5. Note that this static control inference cannot be easily extracted from the other implementations of Figure 1, specially considering the presence of two different timers.

The behavior for the LED timeout just described denotes a *weak abortion*, because the blinking trail had the chance to execute for the last time. By inverting the two trails, the `par/or` would terminate immediately, and the blinking trail would not execute, denoting a *strong abortion* [5]. CÉU not only provides means to choose between weak and strong abortion, but also detects the two conflicting possibilities and issues a warning at compile time (to be discussed in Section 3.2).

Reaction chains should run in bounded time to guarantee that programs are responsive and can handle upcoming input events from the environment. Similarly to Esterel [8], CÉU requires that each possible path in a loop body contains at least one `await` or `break` statement, thus ensuring that loops never run in unbounded time. Consider the examples that follow:

```
loop do
   if <cond> then
      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

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

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

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

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

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

Note that although the variable `x` is accessed concurrently in the first example, the assignments are both atomic and deterministic (due to the scheduling policy and run to completion semantics): the final value of `x` is always 4 (i.e. $(1+1)*2)$). However, programs with concurrent accesses to shared memory are suspicious, because an apparently innocuous reordering of trails modifies the semantics of the program (e.g. the previous example would yield 3 with the trails reordered, i.e., $(1*2+1)$).

We developed a compile-time temporal analysis for CÉU in order to detect concurrent accesses to shared variables, as follows: *if a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable, nor dereference a pointer of that variable type.* An analogous policy is applied for pointers *vs* variables and pointers *vs* pointers. The algorithm for the analysis holds the set of all events in preceding `await` statements for each variable access. Then, the sets for all accesses in parallel trails are compared to assert that no events are shared among them. Otherwise the compiler warns about the suspicious accesses.

Consider the three examples of Figure 3. The first code is detected as suspicious, because the assignments to `x` and `p` (lines 11 and 14) may be concurrent in a reaction to `A` (lines 6 and 13); In the second code, although two of the assignments to `y` occur in reactions to `A` (lines 4-5 and 10-11), they are not in parallel trails and, hence, are safe. The third code illustrates a false positive in our algorithm, as the assignments to `z` in parallel can only occur in different reactions to `A` (lines 5 and 9), as the second assignment awaits two occurrences of `A`.

Conflicting weak and strong abortions, as introduced in Section 3.1, are also detected with the proposed algorithm. Besides accesses to variables, the algorithm also keeps track of trails terminations inside a `par/or`, issuing a warning when they can occur concurrently. This way, the programmer can be aware about the existence and its decision between weak or strong abortion.

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

We also implemented an alternative algorithm that converts a CÉU program into a deterministic finite automata. The resulting DFA represents all possible points a program

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

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

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

Figure 3. Automatic detection for concurrent accesses to shared memory.
The first example is suspicious because x and p can be accessed concurrently (lines 11 and 14). The second example is safe because accesses to y can only occur in sequence. The third example illustrates a false positive in our algorithm.

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

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

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.

3.3 Integration with *C*

Most existing operating systems and libraries for WSNs are based on *C*, given its omnipresence and level of portability across embedded platforms. 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. Code inside "`C do ... end`" is also repassed to the *C* compiler for the final generation phase. As CÉU mimics the type system of *C*, values can be easily passed back and forth between the languages.

C calls are fully integrated with the static analysis presented in Section 3.1 and cannot appear in concurrent trails segments, because CÉU has no knowledge about their side effects. Also, passing variables as parameters counts as read accesses to them, while passing pointers counts as write accesses to those types (because functions may dereference and assign to them). This policy increases considerably the number of false positives in the analysis, given that many functions can actually be safely called concurrently. Therefore, CÉU supports syntactic annotations to relax the policy explicitly. *C* annotations, which are illustrated in Figure 5, are described as follows:

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

Figure 5. Annotations for *C* functions.
Function abs is side-effect free and can be concurrent with any other function. The functions _Leds_led0Toggle and _Leds_led1Toggle can execute concurrently. The variables buf1 and buf2 can be accessed concurrently (annotations are also applied to variables).

- 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.

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, as they can never be active at the same time. The syntactic compositions of trails allows the CÉU compiler to statically allocate and optimize memory usage [22]: memory for trails in parallel must coexist; trails that follow rejoin points reuse all memory.

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

The code snippet in Figure 6 was extracted from our implementation of the CTP collection protocol [32]. The protocol contains a complex control hierarchy in which the trail

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

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

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

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

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

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

The finalization mechanism is fundamental to preserve the orthogonality of the `par/or` construct, thus avoiding to handle trail termination by tweaking other trails in parallel.

3.5 First-class Timers

Activities that involve reactions to *wall-clock time*[3] appear in typical patterns of WSNs, such as timeouts and sensor sampling. However, support for wall-clock time is somewhat low-level in existing languages, usually through timer callbacks or sleep blocking calls. In any concrete system implementation, a requested timeout does not expire precisely with zero-delay, a fact that is usually ignored in the development process. We define the difference between the requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in sequence (or in a loop) may accumulate a considerable amount of deltas that can lead to incorrect behavior in programs.

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

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

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

CÉU also takes into account the fact that time is a physical quantity that can be added and compared. For instance, for the program that follows, although the scheduler cannot

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

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.

3.6 Internal Events

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

In contrast with external events, which are handled in a queue, internal events follow a stack policy. In practical terms, this means that a trail that emits an internal event pauses until all trails awaiting that event completely react to it, continuing to execute afterwards. Another difference to external events is that internal events occur in the same reaction chain they are emitted, i.e., an `emit` instantaneously matches and awakes all corresponding `await` statements that were invoked in *previous reaction chains*[4].

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 7 shows how the dissemination trail from our implementation of the DRIP protocol can be invoked from different parts of the program, just like a subroutine. The DRIP protocol distinguishes from data and metadata packets and disseminates one or the other based on a request parameter. For instance, when the trickle timer expires (line 8), the program invokes `emit send=1` (line 9), which awakes the dissemination trail (line 17) and starts sending a metadata packet (collapsed in line 18). Note that if the trail is already sending a packet, than the `emit` will not match the `await` and will have no effect (just like the *nesC* implementation does, but using a explicit state variable).

Internal events also provide means for describing more elaborate control structures, such as *exceptions*. The code in Figure 8 handles incoming packets for the CC2420 radio driver in a loop. After awaking from a new packet notification (line 4), the program enters in a sequence to read the bytes from the hardware buffer (lines 8-16). If any anomaly is found on the received data, the program invokes `emit next` to discard the current packet (lines 10 and 14). Given the execution semantics of internal events, the `emit` continuation is

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

Figure 7. A loop that awaits an internal event can emulate a subroutine.
The `send` "subroutine" (lines 16-19) is invoked from three different parts of the program (lines 5, 9, and 14).

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 from the TinyOS operating system [19], which are written in *nesC* [17]. We chose TinyOS and *nesC* in our evaluation given its resource efficiency, code base maturity, and because it is used as benchmark in many systems related to CÉU [12, 21, 4, 3].

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 are representative of the realm of system-level development for WSNs [26, 18], which mostly consists of network protocols and lower level utilities to be used as services in applications. They are also rich in control and concurrency aspects, being perfect targets for implementations in CÉU. Finally, they are open standards [32], with open implementations.

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 consists in swapping a single file. This way, we can also use existing test applications available in the TinyOS repository (e.g. *RadioCountToLeds* or *TestNetwork*)

Figure 9 shows the comparison for *Code complexity* and *Memory usage* between the implementations in *nesC* and CÉU, which are discussed in Sections 4.1 and 4.2. The fig-

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

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

Figure 9. Comparison between CÉU and *nesC* for the re-implemented applications.
The column group *Code complexity* compares the number of language tokens and global variables in the sources; the group *Céu features* shows the number of times each functionality is used in each application; the group *Memory usage* compares ROM and RAM consumption.

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

Figure 8. Exception handling in CÉU.
The `emit`'s in lines 10 and 14 raise an exception to be caught by the `await` in line 6. The `emit` continuations are discarded given that the surrounding `par/or` is killed.

ure also details how many times each relevant feature of CÉU was used in the implementations. In Section 4.3, we 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.

4.1 Code Complexity

We use 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 [4, 12], 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 basically predicates, `struct` accessors, etc.).

We chose to use tokens instead of lines of code because the code density is considerably lower in CÉU, as 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 for a fair comparison (e.g. `signal`, `call`, `command`, etc.). The column *Code complexity* in Figure 9 shows a considerable decrease in the number of tokens for all implementations (from 23% up to 69%).

Regarding the metric of number of globals, we categorized them in *state* and *data* variables.

State variables are used as a mechanism to control the application flow (on the lack of a better primitive). Keeping track of them is often regarded as a difficult task, hence, reduction of state variables has already been proposed as a metric of code complexity in a related work [12]. The implementations in CÉU completely eliminated state variables, given that all control patterns could be expressed with hierarchical compositions of activities assisted by internal events communication.

Data variables in WSN programs usually hold message buffers and protocol parameters (e.g. sequence numbers, timer intervals, etc.). In event-driven systems, given that stacks are not retained across reactions to the environment, all data variables must be global[5]. Although the use of local variables does not imply in reduction of lines of code (or tokens), the smallest the scope of a variable, the more

[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).

readable and less susceptible to bugs the program becomes. In the CÉU implementations, most variables could be nested to a deeper scope. The column *local data variables* in Figure 9 shows the new depth of each global that became a local variable in CÉU (e.g. "2;5;6" represents globals that became locals inside blocks in the 2nd, 5th, and 6th depth level).

The columns below *Céu features* in Figure 9 point out how many times each functionality has been used in the implementations in CÉU, helping on identifying where the reduction in complexity comes from. As an example, Trickle uses 2 timers and 3 parallel compositions, resulting in at most 6 trails active at the same time. Six coexisting trails for such a small application is justified by its highly control-intensive nature, and the almost 70% code reduction illustrates the huge gains with CÉU in this context.

4.2 Memory Usage

Memory is a scarce resource in WSN 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, which were carefully tweaked to remove extra functionality, so that the generated binary is dominated by the component of interest. Then, we compiled each application twice: first with the original component in *nesC*, and then with the ported component in CÉU. The column *Memory usage* in Figure 9 shows the consumption of ROM and RAM for the generated applications. With the exception of the Trickle timer, the results in CÉU remain below 10% in ROM and 5% in RAM in comparison with the implementations in *nesC*. Our method and results are similar to those for Protothreads [12], which is an actively supported system for Contiki [11], with a simple and lightweight implementation based on a set of C macros.

The results for Trickle illustrate the footprint of the runtime of CÉU. The RAM overhead of 22% actually corresponds to only 16 bytes: 1 byte for each of the maximum 6 concurrent trails, and 10 bytes to handle synchronization among timers. As the complexity of the application grows, this basic overhead tends to become irrelevant. The SRP implementation shows a decrease in RAM, which comes from the internal communication mechanism of CÉU that could eliminate a queue. Note that both TinyOS and CÉU define functions to manipulate queues for timers and tasks (or trails). Hence, as our implementations mix components in the two systems, we pay an extra overhead in ROM for all applications.

We focused most of the language implementation efforts on RAM optimization, as it has been historically considered more scarce then ROM [25]. Although we have achieved competitive results, we expected more gains with memory reuse for blocks with locals in sequence, because it is something that cannot be done automatically by the *nesC* compiler. However, we analyzed each ported application and it turned out that we had no gains *at all* from blocks in sequence. Our conclusion is that sequential patterns in WSN applications come either from split-phase operations, which always require memory to be preserved; or from loops, which do reuse all memory, but in the same way event-driven systems do.

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

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

4.3 Responsiveness

A known limitation of languages with synchronous and cooperative execution is that they cannot guarantee to meet hard real-time deadlines [10, 23]. For instance, the rigorous synchronous semantics of CÉU forbids non-deterministic preemption to serve high priority trails. Even though CÉU ensures bounded execution for reactions, this guarantee is not extended to *C* function calls, which are usually preferred for executing long computations (due to performance and existing code base). This way, the implementation of a radio driver purely in CÉU raises questions regarding its responsiveness and we conducted two experiments in order to evaluate it.

In the first experiment[6], we "stress-test" the radio driver to compare its performance in the CÉU and *nesC* implementations. We use 10 motes that broadcast 100 consecutive packets of 20 bytes to a mote that runs a periodic time-consuming activity. The receiving handler simply adds the value of each received byte to a global counter. The sending rate of each mote is 200ms (leading to an average of 50 packets 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 1 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 [10, 23]). This way, we simulated it with a simple busy wait that will keep the driver in CÉU unresponsive during that period.

Figure 10 shows the percentage of handled packets in CÉU and *nesC* for each duration. Starting from the column 8ms, the performance of CÉU is 5% worse than the performance of *nesC*. Likewise, the *nesC* driver starts to become unresponsive with operations that take 32ms, which is a similar conclusion taken from TOSThreads experiments (25-bytes packets every 50ms, while running a 50-ms operation [23]). Table 1 shows the duration of some lengthy operations specifically designed for WSNs found in the literature. The operations in the group with timings below 6ms could be used with real-time responsiveness in CÉU (considering the proposed 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

[6]The experiments use the *COOJA* simulator [13] running images compiled to *TelosB* motes.

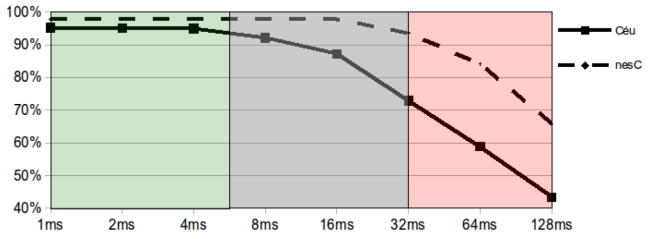

Figure 10. Percentage of received packets depending on the duration of the lengthy operation.

Note the logarithmic scale on the *x*-axis. The packet arrival frequency is 20ms. The operation frequency is 140ms. In the green area, CÉU performs similarly to *nesC*. The gray area represents the region in which *nesC* is still responsive. In the red area, both implementations become unresponsive (i.e. over 5% packet losses).

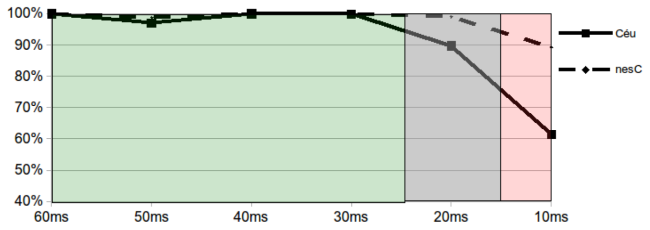

Figure 11. Percentage of received packets depending on the sending frequency.

Each received packet is tied to a 8-ms operation. CÉU is 95% responsive up to a frequency of 25ms per packet.

losses for the CPU awake full time. (We performed an additional test omitting the lengthy operation, yielding the same result of column 1ms in Figure 10.) Note that 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 tied in sequence with every packet arrival to simulate an activity such as encryption. We now run the experiment varying the rate in the 10 sending motes from 600ms to 100ms. Figure 11 shows the percentage of handled packets in CÉU and *nesC* for each rate of message arrival. The results show that CÉU is 95% responsive up to frequency of 40 packets per second.

The overall conclusion is that the radio driver in CÉU performs as well as the original driver in *nesC* under high loads for programs with lengthy operations of up to 4ms, which is a reasonable time for control execution and simple processing. The range between 6ms and 16ms is a "gray area" that offers opportunities for performing more complex operations, but that also requires careful analysis and testing. For instance, the second experiment shows that the CÉU driver can process in real time messages arriving every 35ms in sequence with a 8-ms operation.

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

4.4 Discussion

CÉU targets control-intensive applications and provides abstractions that can express program flow specifications concisely. Our evaluation shows a considerable decrease in code complexity that comes from logical compositions of trails through the `par/or` and `par/and` constructs. They handle startup and termination for trails seamlessly without extra programming efforts. We believe that the small overhead in memory qualifies CÉU as a realistic option for constrained devices. Furthermore, a broad safety analysis, encompassing all proposed concurrency mechanisms, ensures that the high degree of concurrency in WSNs does not pose safety threats to applications.

As a summary, the following safety properties hold for all programs that successfully compile in CÉU:

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

These properties are desirable in any application and are guaranteed as preconditions in CÉU by design. Ensuring or even extracting these properties from less restricted languages requires significant manual analysis.

Even though the achieved expressiveness and overhead of CÉU meet the requirements of WSNs, its design imposes two inherent limitations: the lack of dynamic primitives that would forbid the static analysis, and the lack of hard real-time guarantees. Regarding the first limitation, dynamic features are already discouraged due to resource constraints. For instance, even object-oriented languages targeting WSNs forbid dynamic allocation [3, 33].

To deal with the second limitation, which can be critical in the presence of lengthy computations, we can consider the following approaches: (1) manually placing `pause` statements in tight loops; (2) integrating CÉU with a preemptive system. The first option requires to rewrite the lengthy operations in CÉU with `pause` statements so that other trails can be interleaved with them. This option is the one recommended in many related works that provide a similar cooperative primitive (e.g. `pause` [6], `PT_YIELD` [12], `yield` [21], `post` [17]). Fortunately, CÉU and preemptive threads are not mutually exclusive. For instance, TOSThreads [23] proposes a message-based integration with *nesC* that is safe and matches the semantics of CÉU external events.

5 Related Work

CÉU is strongly influenced by Esterel [8] in its support for compositions and reactivity to events. However, Esterel is focused only on control and delegates to programmers most efforts to deal with data and low-level access to the un-

Language		Complexity				Safety			
name	year	1: sequential execution	2: local variables	3: parallel compositions	4: internal events	5: deterministic execution	6: bounded execution	7: safe shared memory	8: finalization blocks
Preemptive	many	✔	✔		✔		*rt*		
nesC [18]	2003					✔	*async*	✔	
OSM [23]	2005		✔	✔	✔				
Protothreads [13]	2006	✔				✔			
TinyThreads [28]	2006	✔	✔			✔			
Sol [22]	2007	✔	✔	✔		✔	✔		
FlowTalk [4]	2011	✔	✔						
Ocram [5]	2013	✔	✔			✔			
Céu		✔	✔	✔	✔	✔	✔	✔	✔

Figure 12. Table of features found in related work to CÉU.
The languages are sorted by the date they first appeared in a publication. A gray background indicates where the feature first appeared (or a contribution if it appears in a CÉU cell).

derlying platform. For instance, read/write to shared memory among threads is forbidden, and avoiding conflicts between concurrent *C* calls is left to programmers [6]. CÉU deals with shared memory and *C* integration at its very core, with additional support for finalization, conflict annotations, and a static analysis that permeates all languages aspects. This way, CÉU could not be designed easily as pure extensions to Esterel.

Figure 12 presents an overview of work related to CÉU, pointing out supported features which are grouped by those that reduce complexity and those that increase safety. The line *Preemptive* represents languages with preemptive scheduling [7, 23], which are summarized further. The remaining lines enumerate languages with goals similar to those of CÉU that follow a synchronous or cooperative execution semantics.

Many related approaches allow events to be handled in sequence through a blocking primitive, overcoming the main limitation of event-driven systems (column 1 [12, 4, 27, 3, 21]). As a natural extension, most of them also keep the state of local variables between reactions to the environment (column 2). In addition, CÉU introduces a reliable mechanism to interface local pointers with the system through finalization blocks (column 8). Given that these approaches use cooperative scheduling, they can provide deterministic and reproducible execution (column 5). However, as far as we know, CÉU is the first system to extend this guarantee for timers in parallel.

Synchronous languages first appeared in the context of WSNs through OSM [22] and Sol [21], which provide parallel compositions (column 3) and distinguish themselves from multi-threaded languages by handling thread destruction seamlessly [28, 5]. Compositions are fundamental for the simpler reasoning about control that made possible the safety analysis of CÉU. Sol detects infinite loops at compile time to ensure that programs are responsive (column 6). CÉU adopts the same policy, which first appeared in Esterel. Internal events (column 4) can be used as a reactive alternative to shared-memory communication in synchronous languages, as supported in OSM [22]. CÉU introduces a stack-based execution that also provides a restricted but safer form of subroutines.

nesC provides a data-race detector for interrupt handlers (column 7), ensuring that *"if a variable x is accessed by asynchronous code, then any access of x outside of an atomic statement is a compile-time error"* [17]. The analysis of CÉU is, instead, targeted at synchronous code and points more precisely when accesses can be concurrent, which is only possible given its restricted semantics. Furthermore, CÉU extends the analysis for system calls (*commands* in *nesC*) and control conflicts in trail termination. Although *nesC* does not enforce bounded reactions, it promotes a cooperative style among tasks, and provides asynchronous events that can preempt tasks (column 6), something that cannot be done in CÉU.

On the opposite side of concurrency models, languages with preemptive scheduling assume time independence among processes and are more appropriate for applications involving algorithmic-intensive problems. Preemptive scheduling is also employed in real-time operating systems to provide response predictability, typically through prioritized schedulers [7, 14, 15, 23]. The choice between the two models should take into account the nature of the application and consider the trade-off between safe synchronization and predictable responsiveness.

6 Conclusion

We present CÉU, a system-level programming language targeting control-intensive WSN applications. CÉU is based on a synchronous core that combines parallel compositions with standard imperative primitives, such as sequences, loops and assignments. Our work has two main contributions: (1) a resource-efficient synchronous language that can express control specifications concisely; (2) a wide set of compile-time safety guarantees for programs with shared-memory concurrency and access to the underlying platform through *C*.

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

The resource-efficient implementation of CÉU is suitable for constrained sensor nodes and imposes a small memory overhead in comparison to handcrafted event-driven code.

7 References

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

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

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

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

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

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

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

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

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

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

[11] Dunkels et al. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of LCN'04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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