Safe Concurrent Abstractions for Wireless Sensor Networks

Abstract

ABSTRACT

1 Introduction

2 columns - ABSTRACT

2 Overview

2 columns

```
// Ocram implementation
THREAD void blink1() {
   int now = tc_time();
   while(1) {
      now += 250;
      tc_sleep(now);
      tc_toggle_led_0();
   }
}
THREAD void blink2() {
   int now = tc_time();
   while(1) {
      now += 500;
      tc_sleep(now);
      tc_toggle_led_1();
   }
}
```

```
// Ceu implementation
par do
   // blink 1
   loop do
      await 250ms;
      _Leds_led0Toggle();
   end
with
   // blink 2
   loop do
      await 500ms;
      _Leds_led1Toggle();
   end
end
```

Figure 1. "Two blinking LEDs" in Ocram and CÉU.

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

Figure 2. Start/stop behavior for the CC2420 radio driver.

3 The design of Céu

9 columns

3.1 Parallel syntactic compositions

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.

Figure 1 illustrates this difference, showing the implementations for two blinking LEDs with both approaches.

In the Ocram [2] implementation, the two threads are defined as global functions and must be controlled manually, i.e., the decision to start or terminate each is independent of one another and up to the programmer.

In the CÉU implementation, the two trails are syntactically tied together inside a `par` composition, implying that (a) they can only exist together; (b) they always start together (c) if they terminate, they do it 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.

The `par` composition is used when the trails are intended to run forever. CÉU also provides `par/and` and `par/or` compositions in order to logically combine trails: a `par/or` terminates (rejoins) when any of its trails terminates; a `par/and`, only when all terminate;

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

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

The `par/or` construct is regarded as an *orthogonal preemption primitive* [3] because the two sides in the composition do not know when and why they get killed. Furthermore, they need not to be tweaked with synchronization primitives or state variables in order to be affected by related trails in parallel.

The same start/stop control pattern for the radio driver appears in all ported network protocols presented in Section 4. In practical terms, parallel compositions eliminated all state variables in our ports, not only those related to split-phase operations [5].

3.2 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. If there are no remaining awaiting trails, the program terminates. Otherwise, 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.

If a new external input event occurs while a reaction chain is running (step 2), the environment enqueues it to run in the next reaction, because reaction chains must run to completion.

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 arbitrary but ensures a deterministic and reproducible execution for programs.

Reactions to the environment should run in bounded time to guarantee that programs are responsive and can handle upcoming input events. Similarly to Esterel [4], 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
```

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

```
    end                                   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.

3.3 Shared-memory concurrency

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

Concurrency in CÉU is characterized when two or more trails segments execute during the same reaction chain. A trail segment is a sequence of statements separated by an `await`. In the following example, the assignments run concurrently:

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

while in

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

the assignments are never concurrent, as *A* and *B* represent different external events and the segments never execute at the same reaction chain.

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

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

We propose a compile-time *temporal analysis* in order to detect concurrent accesses to shared variables: If a variable is written in a trail segment, then a concurrent trail segment cannot read or write to that variable, nor dereference a pointer of that variable type. An analogous policy is applied for pointers vs variables and pointers vs pointers.

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

Consider the three examples of Figure 3. The first code is detected as suspicious, given that both assignments may be concurrent in a reaction to *A*; In the second code, although two of the assignments occur in reactions to *A*, 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*.

```
input void A;
var int v;
var int* p;
<...>
par/or do
  loop do
    await A;
    if <cond> then
      break;
    end
  end
  v = 1;
with
  await A;
  *p = 2;
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 do
  await A;
  v = 1;
with
  await A;
  await A;
  v = 2;
end
```

Figure 3. The first and third programs are suspicious.

```
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.

The proposed static analysis is only possible due to hierarchical compositions, which provide precise information about the flow of trails, i.e., which may occur in parallel and which are guaranteed to be in sequence.

We also implemented an alternative algorithm that converts a CÉU program into a deterministic finite automata. The resulting DFA represents all possible points a program can reach during runtime and, hence, eliminates all false positives. However, the algorithm is exponential and may be impractical in many situations.

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

3.4 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 functions, types, constants, and globals that the underlying platform already provides.

In order to programs in CÉU access global *C* symbols offered by the target platform, any identifier prefixed with an underscore is repassed *as is* to the *C* compiler that generates the final binary.

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

C calls are fully integrated with the static analysis of CÉU. Given that CÉU has no knowledge about their side

effects, *C* calls cannot appear in concurrent trails segments. 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 be safely called concurrently. Therefore, CÉU supports syntactic annotations that the programmer can use to relax the policy explicitly:

- The `pure` modifier declares a *C* function that does not cause side effects, allowing it to be called concurrently with any other function in the program.
- The `det` modifier declares a pair of variables or functions that do not affect each other, allowing them to be used concurrently.

The following code illustrates CÉU annotations:

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

Annotations are typically write-once declarations (when integrating a *C* service for the first time) to be included in actual applications. Note that the example in Figure 1 should include the annotation for `_Leds_led0Toggle` and `_Leds_led1Toggle` above to be compiled correctly.

CÉU does not extend the bounded execution analysis to *C* function calls. On the one hand, *C* calls must be carefully analyzed in order to keep programs responsive. On the other hand, they also give the programmer means to circumvent the rigor of CÉU in a well-marked way (i.e., the special underscore syntax).

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

3.5 Local scopes and finalization

Local declarations for variables bring definitions closer to their use in programs, increasing the readability and containment of code. Another benefit, specially in the context of WSNs, is that blocks in sequence can share the same memory space, given that they are never active at the same time.

Once again, due to syntactic compositions of trails, the CÉU compiler can statically allocate and optimize memory usage [6]: memory for trails in parallel must coexist; trails that follow rejoin points may reuse all memory.

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

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

The sending loop (lines 7-18) awakes when the beacon timer expires (line 11). The message buffer is declared only

```
 1:  <...>
 2:  par/or do
 3:      <...>           // stop the protocol or radio
 4:  with
 5:      <...>           // neighbour request
 6:  with
 7:      loop do
 8:          par/or do
 9:              <...>   // resend
10:          with
11:              await (dt) ms;  // beacon timer expired
12:              var _message_t msg;
13:              payload = _AMSend_getPayload(&msg, ...);
14:              <prepare the message>
15:              _AMSend_send(..., &msg, ...);
16:              await CTP_ROUTE_RADIO_SENDDONE;
17:          end
18:      end
19:  end
```

Figure 5. Unsafe use of local references.

where it is required (line 12, in the 6th depth-level of the program) and its reference is manipulated by two *TinyOS* functions: `AMSend_getPayload` (line 13), which gets the data region of the message to be prepared (collapsed in line 14); and `AMSend_send` (line 15), which requests the operating system to actually send the message.

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

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

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

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

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

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

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

3.6 Wall-clock time

Activities that involve reactions to *wall-clock time*[1] appear in typical patterns of WSNs, such as sensor sampling and watchdogs. However, support for wall-clock time is somewhat low-level in existing languages, usually through timer callbacks or sleep blocking calls.

In any concrete system implementation, a requested timeout does not expire precisely with zero-delay, a fact that is usually neglected by programmers. We define the difference between the requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in sequence (or in a loop) might accumulate a considerable amount of deltas that could lead to incorrect behavior in programs.

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

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

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

CÉU also takes into account the fact that time is a physical quantity that can be added and compared. For instance, for the program that follows, although CÉU cannot guarantee that the first trail terminates exactly in 11ms, it can at least ensure that the program always returns 1:

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

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

3.7 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.

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

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

The stacked execution for internal events introduces support for a restricted form of subroutines that cannot express recursive definitions (either directly or indirectly), resulting in memory-bounded programs that preclude stack overflows.

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

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

3.8 Code reentrancy

```
 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 handled by `await`.

4 Evaluation

8 columns

4.1 Resource usage

4.2 Expressiveness

(Ease of programming)

4.3 Safety

5 Related work

(See Figure 8.)

2 columns

6 Conclusion

1 columns

7 References

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

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

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

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

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

[6] 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.

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

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

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

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

Figure 8. Table of features found in related works of CÉU.