III
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 (i.e., they cannot be nested). Figure III.1 shows a compact reference of the syntax of Céu, which helps to follow the examples in this chapter.

We start the chapter with the fundamental design decisions behind Céu's execution model, namely the uniqueness of external events and deterministic scheduler (Section III.1). Then, we discuss how they enable safe concurrency support for shared memory and native *C* function calls (Sections III.2 and III.3). We further introduce some new programming features that match Céu's synchronous and safety-oriented design: local scope finalization (Section III.4), first-class timers (Section III.5), and a stack-based communication mechanism (Section III.6). We finish with a discussion that summarizes the chapter by comparing Céu with Esterel (Section III.7).

III.1 The execution model of Céu

Céu is grounded on a precise definition of "logical time" as a discrete sequence of external input events: a sequence because only a single input event is handled at a logical time; discrete because reactions to events are guaranteed to execute in bounded time (here the "physical" notion of time, to be discussed further). The execution model for Céu programs 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.

```
// DECLARATIONS
input <type> <id>;       // external event
event <type> <id>;       // internal event
var   <type> <id>;       // variable

// EVENT HANDLING
await <id>;              // awaits event
emit  <id>;              // emits  event

// COMPOUND STATEMENTS
<...> ; <...> ;          // sequence
if <...> then <...>      // conditional
         else <...> end
loop do <...> end        // repetition
    break                //   (escape loop)
finalize <...>           // finalization
    with <...> end

// PARALLEL COMPOSITIONS
par/and do <...>         // rejoins on termination of both sides
      with <...> end
par/or do  <...>         // rejoins on termination of any side
      with <...> end
par do <...>             // never rejoins
  with <...> end

// C INTEGRATION
_f();                    // C call (prefix '_')
native do <...> end      // block of native code
pure <id>;               // pure annotation
safe <id> with <id>;     // safe annotation
```

Figure III.1: Syntax of Céu.

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 execution model of Céu is based on the hypothesis that internal reactions run *infinitely faster* in comparison to the rate of external events [36]. An internal reaction is the set of computations that execute when an external event occurs. 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 logical time (i.e. awaking on 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 deterministic and reproducible execution for programs,

which is important for simulation purposes. A reaction chain may also contain emissions and reactions to internal events, which are presented in Section III.6.

The synchronous model is applicable to typical WSN applications, which are most of the time waiting for events (e.g. timers and network packets) to perform a fast reaction (e.g. forwarding a packet or blinking a LED) before going idle again.

```
1  input void A, B, C;
2  par/and do
3      // trail 1
4      <...>          // <...> represents non-awaiting statements
5      await A;
6      <...>
7  with
8      // trail 2
9      <...>
10     await B;
11     <...>
12 with
13     // trail 3
14     <...>
15     await A;
16     <...>
17     await B;
18     par/and do
19         // trail 3
20         <...>
21     with
22         // trail 4
23         <...>
24     end
25 end
```

Figure III.2: A Céu program to illustrate the scheduler behavior.

To illustrate the behavior of the scheduler of Céu, the execution of the program in Figure III.2 is depicted in the diagram of Figure III.3. The program starts in the boot reaction and is split in three trails. Following the order of declaration, the scheduler first executes *trail 1* until it awaits A in line 5; then *trail 2* executes until it awaits B in line 10; then *trail 3* is scheduled and also awaits A, in line 15. As no other trails are pending, the reaction chain terminates and the scheduler remains idle until the occurrence of A: *trail 1* awakes, executes and terminates; and then *trail 3* executes and waits for B in line 17. *Trail 2* remains suspended, as it is not awaiting A. During this reaction, new instances of events A, B, and C occur and are enqueued to be handled in the reactions that follow. As A happened first, it is used in the next reaction. However, no trails are awaiting it, so an empty reaction chain takes place. The next reaction dequeues event B: *trail 2* awakes, executes and terminates; then

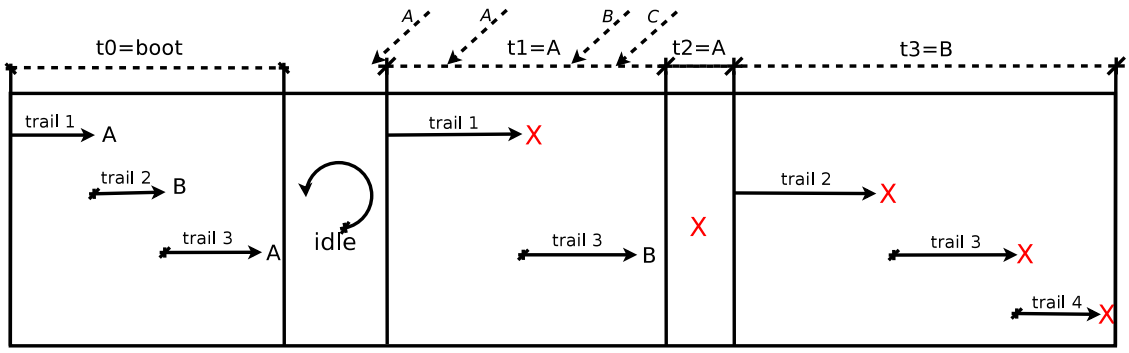

Figure III.3: A sequence of reaction chains for the program in Figure III.2.

trail 3 is split in two and both terminate. The program terminates and does not react to the pending event C. Note that each step in the logical time line ($t0$, $t1$, etc.) is identified by the event it handles. Inside a reaction, trails only react to that identifying event (or remain suspended).

(a) Bounded execution

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 [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. We evaluate the responsiveness of the radio driver when this restriction is (intentionally) relaxed and discuss alternatives adopted in other synchronous languages in Section V.3.

(b) Parallel compositions and abortion

The use of trails in parallel allows that programs wait for multiple events at the same time. Furthermore, trails await without loosing context information, such as locals and the program counter, what is a desired behavior in concurrent applications. [1]

Céu supports three kinds of parallel constructs regarding how they rejoin in the future: a `par/and` requires that all trails in parallel terminate before proceeding to the next statement; a `par/or` requires that any trail in parallel terminates before proceeding to the next statement, aborting all awaiting sibling trails; finally, a `par` never rejoins and should be used when trails in parallel are supposed to run forever (if all trails in `par` terminates, the scheduler forcedly halts them forever). To illustrate how trails rejoin, consider the two variations of the following archetype:

```
loop do
    par/and do
        <...>
    with
        await 100ms;
    end
end
```

```
loop do
    par/or do
        <...>
    with
        await 100ms;
    end
end
```

In the `par/and` variation, the block marked as `<...>` in the first trail (which may contain nested compositions with `await` statements) is repeated every 100 milliseconds at minimum, as both sides must terminate before re-executing the loop. In the `par/or` variation, if the block does not terminate within 100 milliseconds, it is restarted. These archetypes represent, respectively, the *sampling* and *timeout* patterns, which are very common in reactive applications.

The code in Figure III.4 is extracted from our implementation of the *CC2420* radio driver 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); once the client application makes a start 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, including other loops and parallel constructs. However, once the client requests to stop the driver, the trail in line 5 awakes and terminates, causing the `par/or` also terminate and abort the receiving loop. In this case, the top-level loop restarts and waits for the next

```
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 III.4: Start/stop behavior for the radio driver.
The occurrence of `CC2420_STOP` (line 5) seamlessly aborts the receiving loop (collapsed in line 9) and resets the driver to wait for the next `CC2420_START` (line 3).

request to start the radio (line 3, again).

The `par/or` construct of Céu is regarded as an *orthogonal preemption primitive* [7] because the two sides in the composition need not be tweaked with synchronization primitives or state variables in order to affect each other. In contrast, it is known that traditional (asynchronous) multi-threaded languages cannot express thread abortion safely [7, 35]. For instance, it is not safe to terminate a thread holding a lock.

(c) Reasoning about concurrency

The blinking LED of Figure II.2 in Céu illustrates how synchronous parallel constructs lead to a simpler reasoning about concurrency aspects in comparison to the other implementations. As reaction times are assumed to be instantaneous, the blinking loop takes exactly 3 seconds (i.e., $2s + 1s$). Hence, after 20 iterations, the accumulated time becomes 60 seconds and the loop terminates concurrently with the 1-minute timeout in parallel. Given that the loop appears first in the code, the scheduler will restart it and turn on the LED for the last time. Then, the 1-minute timeout is scheduled, aborts 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 detail in Section III.5. Note that this static control inference is impossible in asynchronous languages, given that internal reactions take an unpredictable time (as illustrated in Figure II.1). Even in the other implementations of Figure II.2, this control inference cannot be easily extracted, 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* [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 III.2).

III.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 followed by an `await` (or termination).

In the first code fragment that follows, the two 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` are 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: the final value of `x` is always 4 (i.e. $(1+1)*2$). Remember from Section III.1 that trails are scheduled in the order they appear and run to completion (i.e., until they await or terminate). However, programs with concurrent accesses to shared memory are suspicious, because an apparently innocuous reordering of trails modifies the semantics of the program; for instance, 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 in Figure III.5. 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. Note that the assignment in reaction to `B` (line 8) is safe given that reactions to different events cannot overlap (due to the single-event rule). The third code illustrates a false positive in our algorithm: 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`, while the first trail assigns and terminates in the first occurrence.

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. For this reason, we opted for the simpler algorithm. That being said, the simpler static analysis does not detect false positives in any of the implementations to be presented in Section V and executes in negligible time, suggesting that the algorithm is practical.

Conflicting weak and strong abortions, as introduced in Section III.1, are also detected with the proposed algorithm. Besides accesses to variables, the algorithm also keeps track of trail terminations inside a `par/or`, issuing a warning when they can occur concurrently. This way, the programmer can be aware about the conflict existence and choose 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.

```
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 III.5: 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.

III.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. Therefore, 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. Therefore, access to *C* is seamless and, more importantly, easily trackable. Céu also supports *native blocks* to define new symbols in *C*, as Figure III.6 illustrates. Code inside "`native 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 III.2 and cannot appear in concurrent trails segments, because Céu has no knowledge about their side effects (e.g. calls that access the same LED). Also, passing variables as parameters is regarded as read accesses to them, while passing pointers 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 explicit syntactic annotations to relax the policy. They are illustrated in Figure III.7, and are described as follows:

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

Figure III.6: A Céu program with embedded *C* definitions. The globals `I` and `inc` are defined in the `native` 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).

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

Figure III.7: 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 `safe` 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 resort to *C* for simple operations that can be assumed to be instantaneous, such as non-blocking I/O and `struct` accessors, but never for control purposes.

III.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 allow the Céu compiler to statically allocate and optimize memory usage: 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 the execution of system software outlives the scope of any local variable, a pointer passed as parameter to a system call may be held by a device driver for longer than the scope of the referred variable, leading to a dangling pointer.

The code snippet in Figure III.8 was extracted from our implementation of the CTP collection protocol [39]. The protocol contains a complex control hierarchy in which the trail that sends beacon frames (lines 11-16) may be aborted 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 passed 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 aborts the sending trail and makes the local buffer 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 abort the sending trail (through a call to `AMSend_cancel`). However, this would require expanding the scope of the message buffer, adding a state variable to keep track of the sending status, and duplicating 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 rewritten as

```
1  native 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

```
1  <...>
2  par/or do
3     <...>          // stops the protocol or radio
4  with
5     <...>          // neighbour request
6  with
7     loop do
8        par/or do
9           <...>   // resends 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 III.8: 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.

(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 (i.e., the block the variable is defined terminates). Therefore, regardless of how the sending trail is aborted, 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 among 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 (i.e., at any time the protocol can receive a request to stop immediately). In *nesC*, because message buffers are usually globals, a send that is not properly canceled typically results in an extra packet transmission that wastes battery. However, in the presence of dynamic 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, i.e., an aborted trail does not require clean up code outside it.

III.5 First-class timers

Activities that involve reactions to *wall-clock time*[1] appear in typical code 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, however, a requested timeout does not expire precisely with zero-delay, a fact that is usually ignored in the software 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 guarantee that the first trail terminates exactly in 11ms, it can at least ensure that the program always terminates with `v=1`:

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

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

The time reference for `await` statements is always the beginning of the reaction chain: timers starting in parallel share the same reference. Also, 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. The importance of synchronized timers becomes more evident in the presence of loops, like in the introductory example of Figure II.2 in which the first trail is guaranteed to execute exactly 21 times before being aborted by the timer in the second trail.

Note that in extreme scenarios, small timers in sequence (or in a loop) may never "catch up" with the external clock, resulting in a *delta* that increases indefinitely. To deal with such cases, the current *delta* is always returned from an `await` and can be used in programs:

```
loop do
    var int late = await 1ms;
    if late < 1000 then
        <...>   // normal behavior
    else
        <...>   // abnormal behavior
    end
end
```

III.6 Internal events

Céu provides internal events as a signaling mechanism among parallel trails: 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 order to provide a limited but safe form of subroutines. 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*[2].

The stacked execution for internal events introduces support for a restricted form of subroutines that cannot express recursive definitions (either

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

```
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 III.9: 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).

directly or indirectly), resulting in bounded memory and execution time. Figure III.9 shows how the dissemination trail from our implementation of the DRIP protocol simulates a function and can be invoked from different parts of the program (lines 16-19), just like a subroutine. The `await send` (line 17) represents the function entry point, which is surrounded by a `loop` so that it can be invoked repeatedly. The DRIP protocol distinguishes 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, then the `emit` will not match the `await` and will have no effect (the *nesC* implementation uses an explicit state variable to attain this same behavior).

This form of subroutines has some significant limitations:

***Single instance*:** Calls to a running subroutine have no effect. As noted in the example of Figure III.9, a subroutine that awaits on its body may miss further calls to it (in some cases this behavior is actually desired).

***Single calling*:** Further calls to a subroutine in a reaction chain also have no effect. Even if a subroutine terminates within a reaction chain (i.e.

reaches the `await` again), other `emit` invocations are ignored until the next reaction chain. Remember that *await* statements must be awaiting before the reaction chain starts to be awoken and that `emit` statements are immediately broadcast (i.e., they are not buffered).

***No recursion*:** Recursive calls to a subroutine also have no effect. For the same reason of the *single instance* property, a trail cannot be awaiting itself while running and the recursive call is ignored.

***No concurrency*:** If two trails in parallel try to call the same subroutine, the static analysis warns about non-determinism. Even if the warning is ignored, the call from the first trail takes effect (based on deterministic scheduling), while the second call fails on the *single call* property.

Céu provides no support for standard functions for a number of reasons:

- The interaction with other Céu control primitives is not obvious (e.g., executing an *await* or a *par/or* inside a function).
- They would still be restricted in some ways given the embedded context (e.g. no recursion or closures).
- Programs can always recur to *C* when absolutely necessary.

Regardless of the limitations, this form of subroutines is widely adopted in Céu programs, given that they were designed to work with the other control mechanisms. One should keep in mind that the typical reactive organization of programs (awaiting an external stimulus, reacting to it, and going back to awaiting) does not demand recursive subroutines.

Internal events also provide means for describing more elaborate control structures, such as *exceptions*. The code in Figure III.10 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 stacked and awakes the trail in line 6, which terminates and aborts the whole `par/or` in which the emitting trail is paused. Therefore, the continuation for the `emit` never resumes, and the loop restarts to await a next packet.

```
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 III.10: 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 aborted.

III.7 Differences to Esterel

A primary goal of Céu is to support reliable shared-memory on top of a deterministic scheduler and effective safety analysis (Sections III.1, III.2 and III.3). Esterel, however, does not support shared-memory concurrency because *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [6]. Furthermore, Esterel is deterministic only with respect to reactive control, i.e., *"the same sequence of inputs always produces the same sequence of outputs"* [6]. However, the order of execution for side-effect operations within a reaction is non-deterministic: *"if there is no control dependency and no signal dependency, as in "`call f1() || call f2()`", the order is unspecified and it would be an error to rely on it"* [6].

In Esterel, an external reaction can carry simultaneous signals, while in Céu, a single event defines a reaction. The notion of logical time in Esterel is similar to that of digital circuits, in which multiple wires (signals) can be queried for their status (*present* or *absent*) on each clock tick. Céu more closely reflects event-driven programming, in which occurring events are sequentially and uninterruptedly handled by the program. This design decision is fundamental for the temporal analysis of Section III.2.

Esterel makes no semantic distinctions between internal and external

signals, both having only the notion of presence or absence during the entire reaction [7]. In Céu, however, internal and external events behave differently:

- External events can be emitted only by the environment, while internal events only by the program.
- A single external event can be active at a logical time, while multiple internal events can coexist within a reaction.
- External events are handled in a queue, while internal events follow a stacked execution policy.

In particular, the stack-based execution policy for internal events in Céu enables a limited but safe form of subroutines and an exception-handling mechanism, as discussed in Section III.6.

Apart from these fundamental differences to Esterel, Céu introduces first-class timers with a convenient syntax and predictable behavior (Section III.5), and also finalization blocks to safely handle memory going out of scope (Section III.4).