II
Overview of programming models

Concurrent languages can be generically classified in two major execution models. In the *asynchronous model*, the program activities (e.g. threads and processes) run independently of one another as result of non-deterministic preemptive scheduling. In order to coordinate at specific points, these activities require explicit use of synchronization primitives (e.g. mutual exclusion and message passing). In the *synchronous model*, the program activities (e.g. callbacks and coroutines) require explicit control/scheduling primitives (e.g. returning or yielding). For this reason, they are inherently synchronized, as the programmer himself specifies how they execute and transfer control.

In this chapter we give an overview of these models, focusing on the synchronous model, given that Céu and most related work targeting WSNs [19, 14, 28, 33, 27, 4, 5] (detailed in Chapter VIII) are synchronous.

II.1 Asynchronous model

The asynchronous model of computation can be classified according to how independent activities communicate and synchronize. In *shared memory* concurrency, communication is via global state, while synchronization is via mutual exclusion. In *message passing*, both communication and synchronization happen via exchanging messages.

The default behavior of activities being independent hinders the development of highly synchronized applications. As a practical evidence, Figure II.1 shows a simple application that blinks two LEDs in parallel with different frequencies[1]. We implemented it in two asynchronous styles and also in Céu. For *shared memory* concurrency, we used a multithreaded RTOS[2], while for message passing, we used an *occam* variation for Arduino [25].

The LEDs should blink together every 3 seconds (*least common denominator* between 600ms and 1s). As we expected, even for such a simple appli-

[1] The complete source code and a video demos for the application can be found at http://www.ceu-lang.org/TR/#blink.

[2] http://www.chibios.org/dokuwiki/doku.php?id=start

```
// OCCAM-PI
PROC main ()
  CHAN SIGNAL s1,s2:
  PAR
    PAR
      tick(600, s1!)
      toggle(11, s1?)
    PAR
      tick(1000, s2!)
      toggle(12, s2?)
:
```

```
// ChibiOS
void thread1 () {
  while (1) {
    sleep(600);
    toggle(11);
  }
}
void thread2 () {
  while (1) {
    sleep(1000);
    toggle(12);
  }
}

void setup () {
  create(thread1);
  create(thread2);
}
```

```
// Ceu
par do
  loop do
    await 600ms;
    _toggle(11);
  end
with
  loop do
    await 1s;
    _toggle(12);
  end
end
```

Figure II.1: Two blinking LEDs in OCCAM-PI, ChibiOS and Céu.
Each line of execution in parallel blinks a LED with a fixed (but different) frequency. (The LEDs are connected to I/O ports 11 and 12.) Every 3 seconds both LEDs should light on together. After a couple of minutes of execution, only the implementation in Céu remains synchronized.

cation, the LEDs in the two asynchronous implementations lost synchronism after some time of execution. The Céu implementation remained synchronized for all tests that we have performed.

The implementations are intentionally naive: they just spawn the activities to blink the LEDs in parallel. The behavior for the asynchronous implementations of the blinking application is perfectly valid, as the preemptive execution model does not ensure implicit synchronization among activities. In a synchronous language, however, the behavior must be predictable, and loosing synchronism is impossible by design. We used timers in this application, but any kind of high frequency input would also behave nondeterministically in asynchronous systems.

Note that even though the implementations are syntactically similar, with two endless loops in parallel, the underlying execution models between Céu and the two others are antagonistic, hence, the different execution behavior.

Although this application can be implemented correctly with an asynchronous execution model, it circumvents the language style, as timers need to be synchronized in a single thread. Furthermore, it is common to see similar naive blinking examples in reference examples of asynchronous systems[3],

[3] Example 1 in the RTOS *DuinOS v0.3*: http://code.google.com/p/duinos/.
Example 3 in the occam-based *Concurrency for Arduino v20110201.1855*: http://

suggesting that LEDs are really supposed to blink synchronized, a guarantee that the language cannot provide (as shown with the examples).

II.2 Synchronous model

In this section, we present a review of some synchronous languages and programming techniques that more closely relate to Céu. We refer back to them in detail in Chapter VIII to discuss specific features and differences that require a deeper knowledge about Céu.

Event-driven programming

Event-driven programming is usually employed as a technique in general-purpose languages with no specific support for reactivity. Because only a single line of execution and stack are available, programmers need to deal with the burden of manual stack management and inversion of control. [1]

In the context of WSNs, the programming language *nesC* [19] offers event-driven programming for the TinyOS operating system [23]. The concurrency model of *nesC* is very flexible, supporting serialization among callbacks (the default and recommended behavior), and also asynchronous callbacks that interrupt others. To deal with race conditions, *nesC* supports atomic sections with a semantics similar to mutual exclusion in asynchronous languages. We use *nesC* as the output of the Céu compiler to take advantage of the existing ecosystem for WSNs with TinyOS.

Cooperative multithreading

Cooperative multithreading is an alternative approach to preemptive multithreading where the programmer is responsible for scheduling activities in the program (known as *coroutines* [34]). With this approach, there are no possible race conditions on global variables, as the points that transfer control in coroutines are explicit (and, supposedly, are never inside critical sections).

In the context of WSNs, Protothreads [14] offer lightweight cooperative multithreading for embedded systems. Its stackless implementation reduces memory consumption but precludes support for local variables. Furthermore, Protothreads provide no static safety warranties: programs can loop indefinitely, and accesses to globals are unrestricted.

Finite state machines

The use of finite state machines (FSMs) is a classic technique to implement reactive applications, such as network protocols and graphical user interfaces. A contemporary work [28] targets WSNs and is based on the Statecharts formalism [22].

`concurrency.cc/`.

FSMs have some known limitations. For instance, writing purely sequential flow is tedious [28], requiring programmers to break programs up in multiple states with a single transition connecting each of them. Another inherent problem of FSMs is the state explosion phenomenon, which can be alleviated in some designs that support hierarchical FSMs running in parallel [28].

Synchronous languages

The family of reactive synchronous languages[4] is an established alternative to *C* in the field of safety-critical embedded systems [3]. Two major styles of synchronous languages have evolved: in the *control–imperative* style (e.g. Esterel [10]), programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style (e.g. Lustre [21]), programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming.

Céu is strongly influenced by Esterel in its support for hierarchical compositions of activities and reactivity to events. However, some fundamental differences exist, and we discuss them in detail in Section III.7.

Esterel designers usually advocate that Esterel programs are similar to their specification [8, 10]. Such claim is exemplified with the specification and implementation that follow:

> Emit the output `O` as soon as both the inputs `A` and `B` have been received. Reset the behavior whenever the input `R` is received.

```
module ABRO:
    input A, B, R;
    output O;
    loop
        [ await A  await B];
        emit O
    each R
end
```

The program first defines its input and output signals. Then, it enters in a loop that is restarted on each `R` received. The loop body first awaits both `A` and `B`, and then emits `O`.

In Esterel, `||` is the parallel operator, while `;` is the sequencing operator. This way, `await A` and `await B` run in parallel, and `emit O` is only executed after both awaits return. The `await` primitive suspends the running activity until the given signal is emitted somewhere.

[4] The term *synchronous* is convoluted here: *Synchronous languages* evidently follow the *synchronous programming model*, but multi-purpose languages (e.g., *Java* and *C*) can also behave synchronously by applying techniques such as event-driven programming and state machines.

The communication units in Esterel are the *signals*. A signal is equivalent to an *event* of event-driven programming, and can be instantly broadcast to the entire application, waking up its listeners. Signals are emitted with the `emit` primitive and caught with `await` and other temporal constructs like `loop-each`. The `emit` command may pass a value along with the signal, as in `emit X(1)`. The value of a signal may be accessed prefixing it with `?`, as in `v := v + ?X`.

Esterel supports a rich set of preemption constructs, used to structure activities in hierarchies. The following example [8], uses the `every-do-end`, `loop-each`, and `abort-when` constructs:

```
module Runner:
    input Morning, Step, Second, Meter, Lap;
    every Morning do
        abort
            loop
                abort <RunSlowly> when 15 Second;
                abort
                    every Step do
                        <Jump>  <Breathe>
                    end
                when 100 Meter;
                <FullSpeed>
            each Lap
        when 2 Lap
    end
end
```

Conventional variables are also supported in Esterel, however they cannot be freely shared between concurrent statements. In a statement like `[ v := 1 || v := 2 ]`, the value of `v` would become non-deterministic, a situation that is not acceptable in Esterel's semantics. If a variable is written in any parallel activity, it cannot be read or written elsewhere.

II.3 Programming models in WSNs

A WSN application has to handle a multitude of concurrent events, such as timers and packet transmissions, keeping track of them according to its specification. From a control perspective, programs are composed of two main patterns: *sequential*, i.e., an activity with two or more sub-activities 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 using a 1-minute timeout to interrupt an activity denotes a parallel pattern.

/ nesC */*

```
event void Boot.booted() {
  call T1.start(0)
  call T2.start(60000)
}
event void T1.fired() {
  static int on = 0;
  if (on) {
    call Leds.led0Off();
    call T1.start(1000);
  } else {
    call Leds.led0On();
    call T1.start(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(expired(&timer));
    leds_off(LEDS_RED);
    timer_set(&timer,1000);
    PT_WAIT(expired(&timer));
  }
}
```

/ Céu */*

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

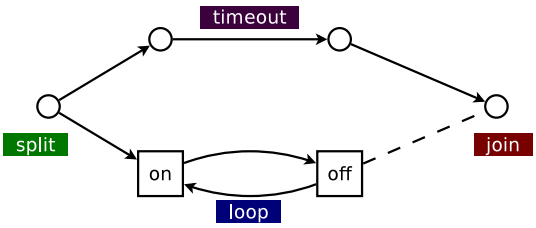

Figure II.2: "Blinking LED" in nesC, Protothreads, and Céu.

Figure II.2 presents the three synchronous programming (sub-)models commonly used in WSNs through a simple concurrent application. It shows the implementations in *nesC* [19], *Protothreads* [14], and Céu for an application that continuously turns 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 bottom-right of Figure II.2 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 "in parallel" 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, and the programmer must manually terminate it (`T1.cancel()`).

The second implementation, in *Protothreads*, which represents the *multi-threaded* model [14, 9], 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 bookkeeping for initializing, scheduling and rejoining the blinking thread after the timeout (inside the `while` condition).

The third implementation, in Céu, which represents the *synchronous-language 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. 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.

The use of timers in the example of Figure II.2 illustrates *split-phase* control operations typically found in WSN applications [23], which require a pair of request and answer to/from the underlying system (e.g. `T1.start` and `T1.fired`). As other examples, requesting sensor readings and forwarding radio packets also require split-phase operations and would be programmed similarly to timers in each of the three models of Figure II.2.