Programs in Céu are structured with standard imperative primitives, such as sequences, loops, and assignments. The extra support for parallelism allows programs to wait for multiple events at the same time. Note that trails await events without loosing context information, such as locals and the program counter, what is a desired behavior in concurrent applications. [1]

The conjunction of parallelism with standard control-flow enable hierarchical compositions, in which self-contained blocks of code can be deployed independently. To illustrate the expressiveness of compositions in Céu, consider the two variations of the structure that follows:

```
loop do                      loop do
    par/and do                   par/or do
        ...                          ...
    with                         with
        await 1s;                    await 1s;
    end                          end
end                          end
```

In the `par/and` loop variation, the code in the first trail (represented as ...) is repeated every second at minimum, as the second trail must also terminate to rejoin the `par/and` primitive and restart the loop.

In the `par/or` loop variation, if the code does not terminate within a second, the second trail rejoins the composition (cancelling the first trail) and restarts the loop.

These structures represent, respectively, the *sampling* and *watchdog* patterns, which are common in embedded applications.

Note that the body of ... may contain arbitrary code with nested compositions and awaits, but the described patterns will work as expected. Even if the code share globals, the program as a whole is analysed at compile time in order to detect inconsistencies (to be discussed in Section 2.2).

Structured programming with local scopes and parallel compositions is not a privilege of synchronous languages. For instance, threads in *C* have stacks to hold local variables, while processes in message-passing systems are easily composable with a single operator. However, synchronous languages provide a more accurate control of the life cycle of concurrent activities. For instance, the code snippet that follows specifies that code `<P>` should be aborted exactly when event `E` occurs, and then code `<Q>` should execute:

```
par/or do
    <P>          // a self-contained composition
with
    await E;     // the preempting event
end
<Q>              // subsequent composition
```

The `par/or` statement is regarded as an *orthogonal preemption primitive* [4], because composition `<P>` does not know when and why it gets aborted by event `E`.

In a similar (hypothetical) construct for an asynchronous language, the occurrence of event `E` would not imply the immediate termination of `<P>`, which could continue to execute for an arbitrary amount of time. Asynchronous formalisms assume time independence among processes and require explicit synchronization mechanisms. Therefore, to achieve the desired specification, processes `<P>` and `<Q>` must be tweaked with synchronization commands, breaking the orthogonality assumption. [4]

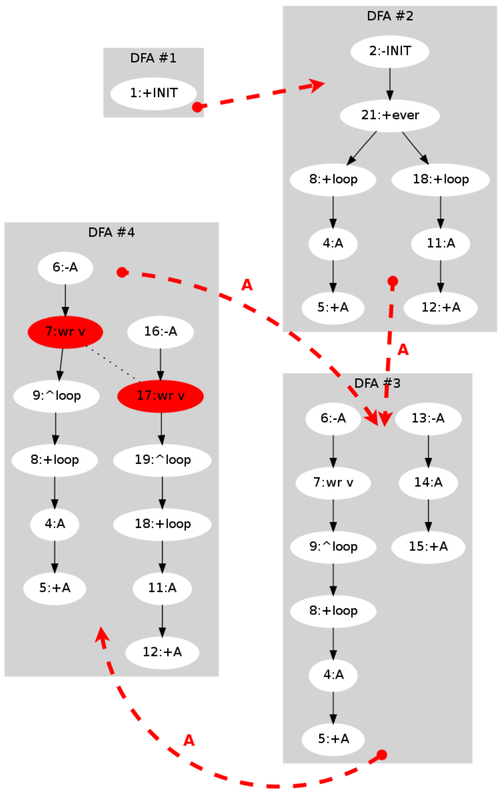

Figure 1: DFA for the nondeterministic example.

0.1 Safety warranties

Safety is an important aspect in embedded systems, given their reliability requirements. Besides lock-free concurrency, Céu ensures at compile time that reactions to events execute in bounded time and are deterministic.

Bounded execution

Reactions to the environment should run in bounded time to guarantee that programs are responsive and can handle upcoming input events. 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                      loop do
    if v > 1000 then             if v > 1000 then
        break;                       break;
    end                          else
    v = v + 1;                       await 1s;
end                              end
                                 v = v + 1;
                             end
```

The first example is refused at compile time, because the `if` true branch may never execute, resulting in an infinite loop. The second variation is accepted, because for every iteration, the loop either escapes or awaits.

Note that Céu does not extend this analysis for *C* function calls. In this case, the responsibility to ensure bounded execution is left to the programmer.

Deterministic execution

Determinism is usually a desired safety property for programs, making concurrency predictable and easier to debug.

Céu performs a compile-time analysis in order to detect nondeterminism in programs. The static analysis relies on the synchronous execution model to generate a deterministic finite automata that covers all possible points a program can reach during runtime.

As an example, the DFA in Figure 1 corresponds to the following program:

```
input void A;
int v;
par do
    loop do
        await A;
        v = 1;
    end
with
    loop do
        await A;
        await A;
        v = 2;
    end
end
```

After two occurrences of event A, variable `v` is accessed concurrently in the program, which is qualified as nondeterministic and is refused at compile time (note the outlined nodes in state *DFA #4*). The same reasoning is done for concurrent *C* calls with side effects, which are illustrated in Section 3.1.

1. DEMO APPLICATIONS

In this section, we present two demos that explore the high-level and safety capabilities of Céu described in the previous section. The applications are somewhat simple to fit the paper (70 and 170 lines), but still complete enough in order to expose the programming techniques promoted by the language.

The first demo targets Wireless Sensor Networks (WSNs), which are networks composed of a large number of tiny devices (known as "motes") capable of sensing the environment and communicating among them [2]. The second demo uses the Arduino open-source platform[1], a popular choice among hobbyists aiming to experiment with electronics in multidisciplinary projects. Both platforms have low processing power and memory capacity, showing that Céu is applicable to highly constrained platforms.

1.1 WSN ring

In the first demo, we implement a fixed ring topology with N motes placed side by side within their radio ranges.

All motes should follow the same behavior: receive a message with an integer counter, show it on the LEDs, wait for 1 second, increment the counter, and forward it to the mote on its right.

Because the topology constitutes a ring, the counter will be incremented forever while traversing the motes. If a mote does not receive a message within 5 seconds, it should blink the red LED every 500 milliseconds until a new message is received.

The mote with *id=0* is responsible for initiating the process at boot time. Also, on perceiving the failure, it should wait for 10 seconds before retrying the communication.

The communicating trail, which continuously receives and forwards the messages, is as follows:

```
1:  loop do
2:     _message_t* msg = await Radio_receive;
3:     int* cnt = _Radio_getPayload(msg);
4:     _Leds_set(*cnt);
5:     await 1s;
6:     *cnt = *cnt + 1;
7:     _Radio_send((_NODE_ID+1)%N, msg);
8:  end
```

The code is an endless loop that first awaits a radio message (line 2), gets a pointer to its data buffer (line 3), shows the received counter on the LEDs (line 4), and then awaits 1s (line 5) before incrementing the counter in the message (line 6) and forwarding it to the next mote (line 7).

The program uses several services provided by the underlying operating system ([8]), which are all non-blocking *C* functions for LEDs and radio manipulation.

Because this code does not handle failures, it is straight to the point and easy to follow. Actually, this is the final code for this task, as error handling is placed in a parallel trail.

Note that the program accesses the message buffer multiple times in reaction to events. Because the `await` primitive provides sequential flow across reactions to events, every access to the buffer in the code is undoubtedly deterministic. However, in typical event-driven systems [7], the accesses would be split in multiple callbacks, requiring a thoughtful reasoning about concurrency issues.

To handle failures, we use a monitoring trail (lines 10-28) in parallel with the communicating trail:

```
  0:  par do
(1-8):  // COMMUNICATING TRAIL (previous code)
  9:  with
 10:     loop do
 11:        par/or do
 12:           await 5s;
 13:           par do
 14:              loop do
 15:                 emit retry;
 16:                 await 10s;
 17:              end
 18:           with
 19:              _Leds_set(0);  // clear LEDs
 20:              loop do
 21:                 _Leds_led0Toggle();
 22:                 await 500ms;
 23:              end
 24:           end
 25:        with
 26:           await Radio_receive;
 27:        end
 28:     end
 29:  end
```

Lines 12 to 24 describe the network-down behavior. After 5 seconds of inactivity is detected (line 12), two new activities run in parallel: one that retries communication every 10 seconds by signaling the internal event `retry` (lines 14-17); and another that blinks the red LED every 500 milliseconds (lines 19-23).

The trick to restore the normal behavior of the network is to await event `Radio_receive` (line 26) in a `par/or` (line 11) with the network-down behavior to kill it whenever a new message is received. By surrounding everything with a `loop` (line 10), we ensure that the error detection is continuous. Note that this is exactly the watchdog pattern described in Section 2.1.

[1]`http://arduino.cc`

Finally, we need to implement the initiating/retrying process that sends the first message from mote with *id=0*. Again, we place the code (lines 30-40) in parallel with other activities:

```
 0:  par do
(1-8):   // COMMUNICATING TRAIL
 9:  with
(10-28): // MONITORING TRAIL
29:  with
30:     if _NODE_ID == 0 then
31:        loop do
32:           _message_t msg;
33:           int* cnt = _Radio_getPayload(&msg);
34:           *cnt = 1;
35:           _Radio_send(1, &msg)
36:           await retry;
37:        end
38:     else
39:        await Forever;
40:     end
41:  end
```

We start by checking whether the mote has *id=0* (line 30). If this is not the case, we simply await forever on this trail (line 39)[2]. Otherwise, the `loop` (lines 31-37) sends the first message as soon as the mote is turned on (line 35). It then waits for a `retry` emit (line 36) to loop and resend the initial message. Remind that event `retry` is emitted on network-down every 10 seconds (line 15).

The static analysis of CÉU correctly complains about concurrent calls to `_Radio_send` (line 35) *vs.* `_Leds_set` and `_Leds_led0Toggle` (lines 19,21), which all execute after the program detects 5 seconds of inactivity (line 12). However, because these functions affect different devices (i.e. radio *vs.* LEDs), they can be safely executed concurrently. The following annotation (to be included in the program) states that these specific functions can be called concurrently with deterministic behavior, allowing the program to be compiled without errors:

```
deterministic _Radio_send with
              _Leds_set, _Leds_led0Toggle;
```

This example shows how complementary activities in an application can be written in separate and need not to be mixed in the code. In particular, error handling (monitoring trail) need not to interfere with regular behavior (communicating trail), and can even be incorporated later. To ensure that parallel activities exhibit deterministic behavior, the CÉU compiler rejects harmful concurrent *C* calls by default.

1.2 Spaceship game

In the next demo, we control a spaceship that moves through space and has to avoid collisions with meteors until it reaches the finish line.

We use a two-row LCD display with two buttons connected to an Arduino to exhibit and control the spaceship. Figure 2 shows the picture of a running quest.

We describe the behavior of the game, along with its implementation, following a top-down approach. The program is constituted of `CODE 1` (set game attributes), `CODE 2` (central loop), and `CODE 3` (game over), which are expanded further. The outermost loop of the game (lines 1-61) is responsible for restarting the game every new phase or on "game over":

[2]`Forever` is a reserved keyword in CÉU, and represents an input event that never occurs.

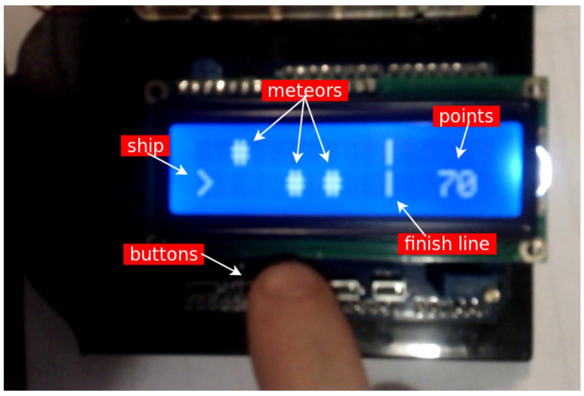

Figure 2: The "spaceship" game

```
 1:  loop do
(2-12): // CODE 1: set game attributes
13:
14:     _map_generate();
15:     _redraw(step, ship, points);
16:     await Key;  // starting key
17:
18:     win =
(19-45): // CODE 2: the central loop
46:
(47-60): // CODE 3: game over
61:  end
```

Every time the loop is executed, it resets the game attributes, such as points and speed (`CODE 1`, lines 2-12), generates a new map (line 14), redraws it on screen (line 15), and waits for a starting key (line 16). Then, the program executes the main logic of the game, in the central loop, until the spaceship reaches the finish line or collides with a meteor (`CODE 2`, lines 19-45). Based on the return status (line 18), the "game over" code (`CODE 3`, lines 47-60) may display an animation before restarting the game.

The game attributes (`CODE 1`) change depending on the result of the previous iteration of the outermost loop (held on variable `win`):

```
     // CODE 1: set game attributes
 2:  ship = 0;          // starts on 1st LCD row
 3:  if !win then
 4:     dt     = 500;   // game speed (500ms/step)
 5:     step   = 0;     // current step
 6:     points = 0;     // number of steps alive
 7:  else
 8:     step = 0;
 9:     if dt > 100 then
10:        dt = dt - 50;
11:     end
12:  end
```

For the first game execution and whenever the spaceship collides with a meteor, variable `win` is false, hence, the attributes are reset to their initial values (lines 4-6). Otherwise, if the player reached the finish line, then the game gets faster, keeping the current points (lines 8-11).

The central loop of the game (`CODE 2`) is responsible for moving the spaceship as time elapses and for checking whether the spaceship reaches the finish line or collides with a meteor:

```
     // CODE 2: the central loop
```

```
19:  par do
20:     loop do
21:        await (dt)ms;
22:        step = step + 1;
23:        _redraw(step, ship, points);
24:
25:        if _MAP[ship][step] == '#' then
26:           return 0;  // a collision
27:        end
28:
29:        if step == _FINISH then
30:           return 1;  // finish line
31:        end
32:
33:        points = points + 1;
34:     end
35:  with
36:     loop do
37:        int key = await Key;
38:        if key == _KEY_UP then
39:           ship = 0;
40:        end
41:        if key == _KEY_DOWN then
42:           ship = 1;
43:        end
44:     end
45:  end;
```

The central loop is actually split in two loops in parallel: one that runs the game steps (lines 20-34), and the other that handles input from the player to move the spaceship (lines 36-44). Note that we want the spaceship to move only during the game action, this is why we did not place the input handling in parallel with the whole application.

The game steps run periodically, depending on the current speed of the game (line 21). For each loop iteration, the step is incremented and the current state is redrawn on screen (lines 22-23). Then, the spaceship is checked for collision with meteors (lines 25-27), and also with the finish line (lines 29-31). CÉU supports returning from blocks with an assignment, hence, lines 26 and 30 escape the whole `par` and assign the appropriate value to variable `win` in the outermost loop (line 18), also cancelling the input handling activity. The points are incremented before each iteration of the loop (line 33).

To handle input events, we wait for key presses in a loop (line 37) and change the spaceship position accordingly (lines 39, 42). Note that there are no possible race conditions on variable `ship` (lines 25 *vs.* 39 and 42) because the two loops in the `par` statement react to different events (i.e. time and key presses).

After returning from the central loop, we run the code for the "game over" behavior, which starts an animation if the spaceship collides with a meteor:

```
     // CODE 3: game over
47:  par/or do
48:     await Key;
49:  with
50:     if !win then
51:        loop do
52:           await 100ms;
53:           _lcd.setCursor(0, ship);
54:           _lcd.write('<');
55:           await 100ms;
56:           _lcd.setCursor(0, ship);
57:           _lcd.write('>');
58:        end
59:     end
60:  end
```

The animation loop (lines 51-58) continuously displays the spaceship in the two directions, suggesting that it has hit a meteor. The animation is interrupted when the player presses a key (line 48), proceeding to the game restart. Note the use of the `_lcd` object, available in a third-party $C++$ library shipped with the LCD display, and used unmodified in the example.

This demo makes extensive use of global variables, relying on the deterministic concurrency analysis guaranteed by the CÉU compiler. We used a top-down approach to illustrate the hierarchical compositions of blocks of code. For instance, the "game over" animation, which is self-contained in lines 51-58, can be easily removed or switched for a new behavior without considering other parts of the program.

2. RELATED WORK

A number of previous works propose new programming languages as high-level alternatives to reduce code complexity in embedded systems [6, 11, 10, 9]. Roughly speaking, they offer adaptations to constrained platforms of classical programming models, such as cooperative multithreading, finite state machines, Esterel, and message-passing.

In cooperative multithreading, the programmer himself is responsible for controlling the life cycle of activities in the program (e.g. creating, starting, rejoining, and destroying). With this approach, there are no possible race conditions on shared variables, as the points that transfer control are explicit (and, supposedly, are never inside critical sections). CÉU goes one step further and, through parallel compositions, makes manual bookkeeping of activities unnecessary.

Regarding finite state machines (FSMs), they are applied in control-intensive algorithms, such as network protocols. However, implementing sequential flow in FSMs is tedious, requiring to break programs 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 with support for parallel hierarchical FSMs [11]. However, adopting parallelism precludes the use of shared state, or at least requires a static analysis such as that of CÉU.

Esterel also provides an imperative reactive style with a similar set of parallel compositions (in which CÉU was based). However, Esterel has no support for shared variables: "if a variable is written in a thread, then it can be neither read nor written in any concurrent thread" [5]. We believe that shared-memory concurrency can be convenient when backed by a safety analysis.

Finally, message passing systems employ time independence among processes, as discussed in Section 2.1, hindering the development of highly synchronized applications.

None of the referred works provide a compile-time analysis that enforces deterministic access to shared variables and *C* calls. Also, some of them lack bounded execution warranties for loops ([6, 10, 11]) and fully orthogonal compositions ([6, 9]) as discussed in Section 2.

3. CONCLUSION

This paper promotes high-level and safe development of embedded systems through capabilities found in the synchronous programming language CÉU. Applications can be implemented as hierarchical compositions of sequences, loops, assignments and parallelism. Furthermore, support for awaiting time and events eliminates the need of callbacks for designing reactive systems.

The presented demos show how typical patterns in embedded systems such as sampling and watchdogs can be easily implemented. They also explore parallel compositions for specifying complementary activities in separate. Communication among activities can either use internal events, or safe access to global variables.

The synchronous execution model eliminates the need for mutual exclusion primitives, enabling a simpler lock-free form of concurrency. Regarding safety, programs rely on the static analysis of CÉU to ensure deterministic behavior for concurrent access to globals and *C* calls. The language also guarantees bounded execution on reactions to events, which is an important requirement for real-time systems.

The applications are targeted at highly constrained embedded platforms, which fit the small memory footprint of the CÉU runtime (around 2Kbytes of ROM and 50bytes of RAM). The examples also make recurrent use of *C* through a seamless interaction with globals, types, and third-party libraries from the underlying platform.

4. REFERENCES

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

[2] I. F. Akyildiz et al. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

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

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

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

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

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

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

[9] C. L. Jacobsen et al. Concurrent event-driven programming in occam-pi for the Arduino. In *In Proceedings of CPA '11*, volume 68, pages 177–193, June 2011.

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

[11] 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.

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