Structured Reactive Programming with Céu

Francisco Sant'Anna
Departamento de Informática — PUC-Rio, Brazil
fsantanna@inf.puc-rio.br

Roberto Ierusalimschy
Departamento de Informática — PUC-Rio, Brazil
roberto@inf.puc-rio.br

Noemi Rodriguez
Departamento de Informática — PUC-Rio, Brazil
noemi@inf.puc-rio.br

Abstract

Structured reactive programming (SRP) augments classical structured programming with continuous interaction with the environment. We propose a new SRP abstraction mechanism for the synchronous language CÉU: *Organisms* extend objects with an execution body that composes multiple lines of execution to react to the environment independently. Compositions bring structured reasoning to concurrency and can better describe state machines typical of reactive applications. Organisms are subject to lexical scope and automatic memory management similar to stack-based allocation for local variables. We show that this model does not require garbage collection or a `free` primitive in the language, eliminating memory leaks by design.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Concurrency, Determinism, Esterel, Imperative, Structured Programming, Synchronous, Reactivity

1. Introduction

Reactive applications interact continuously and in real time with external stimuli from the environment [4, 20]. They represent a wide range of software areas and platforms: from games in powerful desktops and "apps" in capable smart phones, to the emerging internet of things in constrained embedded systems.

Research on special-purpose reactive languages dates back to the early 80's, with the co-development of two complementary styles [5, 27]: The imperative style of Esterel [8] organizes programs with structured control flow primitives, such as sequences, repetitions, and parallelism. The dataflow style of Lustre [19] represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming (FRP) [32] has modernized the dataflow style, inspiring a number of languages and libraries, such as Flapjax [25], Rx (from Microsoft), React (from Facebook), and Elm [12]. In contrast, the imperative style of Esterel is confined to the domain of real-time embedded control systems. As a matter of fact, imperative reactivity is now often associated to the *observer pattern*, typical in object-oriented systems, because it heavily relies on side effects [23, 28]. However, short-lived callbacks (i.e., the observers) eliminate any vestige of structured programming, such as support for long-lasting loops and automatic variables [3], which are elementary capabilities of imperative languages. In this sense, callbacks actually disrupt imperative reactivity, becoming "our generation's `goto`" [14, 16].

We believe that the full range of reactive applications can benefit from the imperative style of Esterel, which we now refer to as *Structured Reactive Programming (SRP)*. SRP extends the classical hierarchical control constructs of *Structured Programming (SP)* (i.e., concatenation, selection, and repetition [15]) to support continuous interaction with the environment. SRP retains structured and sequential reasoning of concurrent programs which constrasts with FRP, bringing the historical dichotomy between functional and imperative languages also to the reactive domain. However, the original rigorous semantics of Esterel, which focuses on static safety guarantees, is not suitable for other reactive application domains, such as GUIs, games, and distributed systems. For instance, the lack of abstractions with dynamic lifetime makes it difficult to deal with virtual resources such as graphical widgets, game units, and network sessions.

In practical terms, SRP provides three extensions to SP: an "`await <event>`" statement that suspends a line of execution until the referred event occurs, keeping all context alive; parallel constructs to compose multiple lines of execution and make them concurrent; and an orthogonal mechanism to abort parallel compositions. The `await` statement represents the imperative-reactive nature of SRP, recovering sequential execution lost with the observer pattern. Parallel compositions[1] allow for multiple `await` statements to coexist, which is necessary to handle concurrent events, common in reactive applications. Orthogonal abortion is the ability to abort an activity from outside it, without affecting the overall consistency of the program (e.g., properly releasing global resources).

In this work, we extend the Esterel-based language CÉU [29] with a new abstraction mechanism, the *organisms*, that encapsulate parallel compositions with an object-like interface. In brief, organisms are to SRP like procedures are to SP, i.e., one can abstract a portion of code with a name and manipulate (call) that name from multiple places. Unlike Simula objects [13], organisms react independently to the environment and do not depend on cooperation, i.e., once instantiated they become alive and reactive (hence the name organisms). Furthermore, both static and dynamic allocation of organisms are subject to lexical scope and automatic memory management, not relying on heap allocation at all, and behaving much like local variables in SP.

The rest of the paper is organized as follows: Section 2 presents SRP through CÉU, with its underlying synchronous concurrency model and parallel compositions. Section 3 describes the organisms

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] In this work, the term *parallel composition* does not imply many-core parallel execution.

```
1  input void RESET;   // declares an external event
2  var int v = 0;      // variable shared by the trails
3  par do
4      loop do         // 1st trail
5          await 1s;
6          v = v + 1;
7          _printf("v = %d\n", v);
8      end
9  with
10     loop do         // 2nd trail
11         await RESET;
12         v = 0;
13     end
14 end
```

Figure 1. Introductory example in CÉU.

abstraction with static and dynamic instantiation, lexical scope, and automatic memory management. Section 5 discusses related work. Section 6 concludes the paper.

2. SRP with Céu

CÉU is a concurrent language in which the lines of execution, known as *trails*, react all together continuously and in synchronous steps to external stimuli. The introductory example in Figure 1 starts two trails with the `par` construct: the first (lines 4-8) increments variable `v` on every second and prints its value on screen; the second (lines 10-13) resets `v` on every external request to `RESET`. Programs in CÉU can access C libraries of the underlying platform by prefixing symbols with an underscore (e.g., `_printf(<...>)`, in line 7).

2.1 Synchronous concurrency

In CÉU, a program reacts completely to an occurring event before handling the next. A reaction represents a logical instant in which all trails awaiting the occurring event awake and execute, one after the other, until they await again or terminate. During a reaction, the environment is invariant and does not interrupt the running trails[2]. If multiple trails react to the same event, the scheduler employs lexical order to preserve determinism, i.e., the trail that appears first in the source code executes first. To avoid infinite execution for reactions, CÉU ensures that all loops contain `await` statements [29].

As a result of synchronous execution, all consecutive operations to variable `v` in Figure 1 are atomic because reactions to events `1s` and `RESET` can never interrupt each other. In contrast, in asynchronous models with nondeterministic scheduling, the occurrence of `RESET` could preempt the first trail during an increment to `v` (line 6) and reset it (line 12) before printing it (line 7), characterizing a race condition on the variable. The example illustrates the (arguably simpler) reasoning about concurrency under the synchronous execution model.

The synchronous model also empowers SRP with an orthogonal abortion construct that simplifies the composition of activities[3]. The code that follows shows the `par/or` construct of CÉU which composes trails and rejoins when either of them terminates, properly aborting the other:

```
par/or do
    <trail-1>
with
    <trail-2>
end
<subsequent-code>
```

[2] The actual implementation enqueues incoming input events to process them in further reactions.

[3] We use the term activity to generically refer to a language's unit of execution (e.g., *thread*, *actor*, *trail*, etc.).

Each trail has a set of local variables and an execution body that lasts for an arbitrary time. After the `par/or` rejoins, a new set of local variables goes alive, reusing the space from the trails' locals which are now out of scope.

The `par/or` is regarded as orthogonal because the composed trails do not know when and how they are aborted (i.e., abortion is external to them). This is possible in synchronous languages due to the accurate control of concurrent activities, i.e., in between every reaction, the whole system is idle and consistent [7]. CÉU extends orthogonal abortion to work also with activities that use stateful resources from the environment (such as file and network handlers), as we discuss in Section 2.2.

Abortion in asynchronous languages is challenging [7] because the activity to be aborted might be on a inconsistent state (e.g., holding pending messages or locks). This way, the possible (unsatisfactory) semantics for a hypothetical `par/or` are: either wait for the activity to be consistent before rejoining, making the program unresponsive to incoming events for an arbitrary time; or rejoin immediately and let the activity complete in the background, which may cause race conditions with the subsequent code. Immediate rejoin also implies heap-based allocation for locals (which is discouraged in the context of embedded systems) because activities may coexist with the subsequent code for some time. In fact, asynchronous languages do not provide effective abortion: *Java*'s `Thread.stop` primitive has been deprecated [26]; *pthread*'s `pthread_cancel` does not guarantee immediate cancellation [2]; *Erlang*'s `exit` either enqueues a terminating message (which may take time), or unconditionally terminates the process (regardless of its state) [1]; and *CSP* only supports a composition operator that *"terminates when **all** of the combined processes terminate"* [22]. As an alternative, asynchronous activities typically agree on a common protocol to abort each other (e.g., through shared state variables or message passing), which increases coupling among them with implementation details that are not directly related to the problem specification.

2.2 Parallel compositions

In terms of control structures, SRP basically extends SP with parallel compositions, allowing applications to handle multiple events concurrently. CÉU provides three parallel constructs that vary on how they rejoin: a `par/and` rejoins when all trails in parallel terminate; a `par/or` rejoins when any trail in parallel terminates; a `par` never rejoins (even if all trails in parallel terminate). The code chunks that follow compare the `par/and` and `par/or` compositions side by side:

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

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

The code `<...>` represents a complex operation with any degree of nested compositions. In the `par/and` variation, the operation repeats on intervals of at least one second because both sides must terminate before re-entering the loop. In the `par/or` variation, if the operation does not terminate within 1 second, it is restarted. These SRP archetypes represent, respectively, the *sampling* and *timeout* patterns, which are typical of reactive applications.

The example in Figure 4.1 relies on hierarchical `par/or` and `par/and` compositions to describe the state machine of a protocol for sensor networks [18, 29]. The input events `START`, `STOP`, and `RETRANSMIT` (line 1) represent the external interface of the protocol with a client application. The protocol enters the top-level loop and awaits the starting event (line 3). Once the client application makes

```
1  input void START, STOP, RETRANSMIT;
2  loop do
3      await START;
4      par/or do
5          await STOP;
6      with
7          loop do
8              par/or do
9                  await RETRANSMIT;
10             with
11                 await <rand> s;
12                 par/and do
13                     await 1min;
14                 with
15                     <send-beacon-packet>
16                 end
17             end
18         end
19     with
20         <...> // the rest of the protocol
21     end
22 end
```

Figure 2. Parallel compositions can describe complex state machines.

```
var _pkt_t buffer;
<fill-buffer-info>
_send_enqueue(&buffer);
await SEND_ACK;
```

```
var _pkt_t buffer;
<fill-buffer-info>
finalize
    _send_enqueue(&buffer)
with
    _send_dequeue(&buffer);
end
await SEND_ACK;
```

Figure 3. Finalization clauses safely release stateful resources.

a start request, the protocol starts three other trails: one monitors the stopping event (line 5); one periodically transmits a status packet (lines 7-18); and one handles the remaining functionality of the protocol (collapsed in line 20). The periodic transmission is another loop that starts two other trails (lines 8-17): one to handle an immediate retransmission request (line 9); and one to await a small random amount of time and transmit the status packet (lines 11-16). The transmission (collapsed in line 15) is enclosed with a `par/and` that takes at least one minute before looping, to avoid flooding the network with packets. At any time, the client may request a retransmission (line 9), which terminates the `par/or` (line 8), aborts the ongoing transmission (line 15, if not idle), and restarts the loop (line 7). Also, the client may request to stop the whole protocol at any time (line 5), which terminates the outermost `par/or` (line 4) and aborts the transmission and all composed trails. In this case, the top-level loop restarts (line 2) and waits for the next request to start the protocol (line 3), ignoring all other requests (as the protocol specifies).

The example shows how parallel compositions can describe complex state machines in a structured way, eliminating the use of global state variables for this purpose [29].

2.2.1 Finalization

The CÉU compiler tracks the interaction of `par/or` compositions with local variables and stateful *C* functions (e.g., device drivers) in order to preserve safe orthogonal abortion of trails.

Consider the code in the left of Figure 3, which expands the sending trail of Figure 4.1 (line 15). The `buffer` packet is a local variable whose address is passed to function `_send_enqueue`. The call enqueues the pointer in the radio driver, which holds it up to the emission of `SEND_ACK` acknowledging the packet transmission. In the meantime, the sending trail might be aborted by `STOP` or `RETRANSMIT` requests (Figure 4.1, lines 5 and 9), making the packet buffer go out of scope, and leaving behind a *dangling pointer* in the radio driver. CÉU refuses to compile programs like this and requires *finalization* clauses to accompany stateful *C* calls [29]. The code in the right of Figure 3 properly dequeues the packet if the block of `buffer` goes out of scope, i.e., the finalization clause (after the `with`) executes automatically on external abortion.

3. Organisms: SRP Abstractions

In SP, the typical abstraction mechanism is a procedure, which abstracts a routine with a meaningful name that can be invoked multiple times with different parameters. However, procedures were not devised for continuous input, and cannot retain control across reactions to the environment.

CÉU abstracts data and control into the single concept of organisms. A class of organisms describes an interface and an execution body. The interface exposes public variables, methods, and also internal events (exemplified later). The body can contain any valid code in CÉU, including parallel compositions. When an organism is instantiated, its body starts to execute in parallel with the program. Organism instantiation can be either static or dynamic.

The example in Figure 4 introduces static organisms with three code chunks:

- The leftmost code (*CODE-1*) blinks two LEDs with different frequencies in parallel and terminates after 1 minute.
- The code in the middle (*CODE-2*) abstracts the blinking LEDs in an organism class and uses two instances of it to reproduce the same behavior of *CODE-1*.
- The rightmost code (*CODE-3*) is the semantically equivalent expansion of the organisms bodies, which resembles the original *CODE-1*.

In *CODE-2*, the Blink class (lines 1-9) exposes the `pin` and `dt` properties, corresponding to the LED I/O pin and the blinking period, respectively. The application then creates two instances, specifying those properties in the constructors (lines 12-15 and 17-20). Inside constructors, the identifier `this` refers to the organism under instantiation. The constructors automatically start the organisms bodies (lines 5-8) to run in parallel in the background, i.e., both instances are already running before the `await 1min` (line 22).

CODE-3 is semantically equivalent to *CODE-2*, but with the organism constructors and bodies expanded (lines 10-17 and 19-26). The generated `par/or` (lines 9-29) makes the instances concurrent with the rest of the application (i.e., the `await 1min`, in line 28). Note the generated `await FOREVER` statements (lines 17 and 26) to avoid the organisms bodies to terminate the `par/or`. The `_Blink` type (lines 1-4) corresponds to a simple datatype without an execution body. The actual implementation of CÉU does not expand the organisms bodies like in *CODE-3*; instead, a class generates a single code for its body, which is shared by all instances (in the same way as objects share class methods).

The main distinction between organisms and standard objects is how organisms can react independently and directly to the environment. For instance, organisms need not be included in observer lists for events, or rely on the main program to feed their methods with input from the environment. Although the organisms run independently from the main program, they are still subject to the disciplined synchronous model, which keeps the whole system deterministic, as the static expansion of *CODE-3* suggests (and based on the scheduler description of Section 2.1).

The memory model for organisms is similar to stack-living local variables of procedures, employing lexical scope and automatic management. Note that *CODE-2* uses a `do-end` block (lines 11-23) that limits the scope of the organisms for 1 minute (line 22). During that period, the organisms are accessible (through `b1` and `b2`) and reactive to the environment (i.e., blinking continuously). After that period, the organisms go out of scope and not only they be-

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

/* CODE-1: original blinking */
```

```
class Blink with
    var int pin;
    var int dt;
do
    loop do
        await (this.dt)ms;
        _toggle(this.pin);
    end
end

do
    var Blink b1 with
        this.pin = 11;
        this.dt  = 600;
    end;

    var Blink b2 with
        this.pin = 12;
        this.dt  = 1000;
    end;

    await 1min;
end

/* CODE-2: blinking organisms */
```

```
1  struct _Blink with
2      var int pin;
3      var int dt;
4  end;
5
6  do
7      var _Blink b1, b2;
8
9      par/or do
10         // body of b1
11         b1.pin = 11;
12         b1.dt  = 600;
13         loop do
14             await (b1.dt)ms;
15             _toggle(b1.pin);
16         end
17         await FOREVER;
18     with
19         // body of b2
20         b2.pin = 12;
21         b2.dt  = 1000;
22         loop do
23             await (b2.dt)ms;
24             _toggle(b2.pin);
25         end
26         await FOREVER;
27     with
28         await 1min;
29     end
30 end
31
32 /* CODE-3: organisms expansion */
```

Figure 4. Two blinking LEDs using organisms.

come inaccessible but their bodies are automatically aborted, as the expansion of *CODE-3* makes clear: The `par/or` (lines 9-29) aborts the organisms bodies after 1 minute (line 28), just before they go out of scope (line 30). The `par/or` termination properly triggers all active finalization clauses inside the organism bodies (if any). Lexical scope extends the idea of orthogonal abortion to organisms, as they are automatically aborted when going out of scope. In this sense, organisms are more than a cosmetic convenience for programmers because they tie together data and associated execution into the same scope.

In addition to properties and methods, organisms also expose internal events which support `await` and `emit` operations. In the example in Figure 5, class `Unit` (lines 1-16) defines the position and destination properties `pos` and `dst` with default values `0` (lines 2-3), and the event `move` to listen requests to move the unit position (line 4). The main program (lines 18-29) creates two units, requests the first to move immediately to `dst=300` (line 20), and the second to move after 1 second to position `500` (line 29). The body of the class enters a continuous loop (lines 6-15) to handle `move` requests (line 8) while performing the ongoing moving operation (lines 10-13) in parallel. The `par/or` (lines 7-14) restarts the loop on every `move` request which updates the `dst` position. The moving operation (collapsed in line 11) can be as complex as needed, for example, using another loop to apply physics over time. The `await FOREVER` (line 13) halts the trail after the move completes to avoid restarting the outer loop. An advantage of event handling over method calls is that they can be composed in the organism body to affect other ongoing operations. In the example, the `await move` (line 8) aborts and restarts the moving operation, just like the timeout pattern of Section 2.2.

3.1 Dynamic organisms

Static embedded systems typically manipulate hardware with a one-to-one correspondence in software, i.e., a static piece of software deals with a corresponding piece of hardware (e.g., a sensor or actuator). In contrast, more general reactive systems have to deal with resource virtualization that requires dynamic allocation, such as multiplexing protocols in a network, or simulating entire civilizations in a game. Dynamic allocation for organisms extends the power of SRP to handle virtual resources in reactive applications.

```
1  class Unit with
2      var   int pos = 0;
3      var   int dst = 0;
4      event int move;
5  do
6      loop do
7          par/or do
8              dst = await this.move;
9          with
10             if dst != pos then
11                 <code-to-move-pos-to-dst>
12             end
13             await FOREVER;
14         end
15     end
16 end
17
18 var Unit u1 with
19     this.pos = 100;
20     this.dst = 300;
21 end;
22
23 var Unit u2 with
24     this.pos = 200;
25     this.dst = 200;
26 end;
27
28 await 1s;
29 emit u2.move => 500;
```

Figure 5. Organism manipulation through interface events.

CÉU supports dynamic instantiation of organisms through the `spawn` primitive. The example that follows spawns a new instance of `Unit` (defined in Figure 5) every second and moves it to a random position:

```
loop do
    await 1s;
    spawn Unit with
        this.pos = _rand() % 500;
        this.dst = _rand() % 500;
    end;
end
```

Dynamic instances also execute in parallel with the rest of the application, but have different lifetime and scoping rules then static ones: A static instance has an identifier and a well-defined scope that holds its memory resources; A dynamic instance is anonymous and outlives the scope that spawns it: in the example, the spawned units outlive the enclosing loop iterations. Due to the lack of an explicit identifier or reference, a dynamic instance can control its own lifetime: once its body terminates, a dynamic organism is automatically freed from memory. This does not apply for a static instance because its memory is statically preallocated and its identifier is still accessible even if its body terminates.

The code that follows redefines the body of the `Unit` class of Figure 5 to terminate after 1 hour, imposing a maximum life span in which a unit can react to `move` requests. After that, the body terminates and the organism is automatically freed (if dynamically spawned):

```
class Unit with
    <...>           // interface
do
    par/or do
        <...>       // moving trail
    with
        await 1h;
    end
end
```

The lack of scopes for dynamic organisms prevents orthogonal abortion, given that there is no way to externally abort the execution of a dynamic instance. To address orthogonal abortion, CÉU provides lexically scoped *pools* as containers that hold dynamic instances. The example that follows declares the `units` pool to hold a maximum of 10 instances (line 3):

```
1  input void CLICK;
2  do
3      pool Unit[10] units;
4      par/or do
5          loop do
6              await 1s;
7              spawn Unit in units with
8                  <...>   // constructor
9              end;
10         end
11     with
12         await CLICK;
13     end
14 end
```

A new unit is spawned in this pool once a second (note the `in units`, in line 7). Once the application receives a `CLICK` (line 12), the `par/or` (line 4) terminates, making the `units` pool go out of scope and abort/free all units alive.

Pools with bounded dimension (e.g., `pool Unit[10] units`), have static pre-allocation, resulting in efficient and deterministic organism instantiation. This opens the possibility for dynamic behavior also in constrained embedded systems. If a pool does not specify a dimension (e.g., `pool Unit[] units`), the instances go to the heap but are still subject to the pool scope. If a `spawn` does not specify a pool, as in "`spawn Unit;`", the instances go to a predefined dimension-less pool in the top of the current class (and are still subject to that pool scope).

Support for lexical scope for both static and dynamic organisms eliminate garbage collection, `free` primitives, and memory leaks altogether.

```
var Unit* ptr = spawn Unit;
ptr:pos = 0;
watching ptr do
    await 2h;
    emit ptr:move => 100;
end
```

```
var Unit* ptr = spawn Unit;
ptr:pos = 0;
par/or do
    await ptr:__killed;
with
    await 2h
    emit ptr:move => 100;
end
```

Figure 6. Watching a reference with equivalent expansion.

3.2 Pointers Analysis

As organisms react independently to the environment, it is often not necessary to hold pointers to them. Nonetheless, a `spawn` allocation returns a pointer to the new organism, which can be later dereferenced with the operator '`:`' (analogous to '`->`' of *C/C++*):

```
var Unit* ptr = spawn Unit;
ptr:pos = 0;                // this access is safe
await 2h;
emit ptr:move => 100;       // this access is unsafe
```

Pointers can be dangerous because they may last longer than the organisms to which they refer. The code above first acquires a pointer `ptr` to a `Unit`. Then, it dereferences the pointer in two occasions: in the same reaction, just after acquiring the reference; and in another reaction, after *2h*, when the pointed organism may have already terminated and been freed, leading to unspecified behavior in the program.

As a protection against dangling pointers, CÉU enforces all pointer accesses across reactions to use the `watching` construct which supervises organism termination, as illustrated in the left of Figure 6. The whole `watching` construct aborts whenever the referred organism terminates, eliminating possible dangling pointers in the program. The code in the right shows the equivalent expansion of the `watching` construct into a `par/or` that awaits the special event `__killed` (which all classes have internally).

CÉU also refuses to assign the address of an organism to a pointer of greater scope, as illustrated below:

```
var Unit* ptr;
do
    var Unit u;
    ptr = &u;   // illegal attribution
end
ptr:pos = 0;    // unsafe access ("u" went out of scope)
```

Pools supports iterators to acquire temporary pointers to all alive instances. To preserve safe pointer accesses, iterators cannot await. The example that follows iterates over the `units` pool to check for collision among units:

```
pool Unit[10] units;
<...>
loop (Unit*)u1 in units do
    loop (Unit*)u2 in units do
        if <check-collision> then
            emit u1:move => _rand() % 500;
            emit u2:move => _rand() % 500;
        end
    end
end
```

4. Applications

- constrained - general

To demonstrate the expressiveness of CÉU, we implemented three applications in different domains and platforms.

The first example explores Wireless Sensor Networks (WSNs), which are networks composed of a large number of tiny devices capable of sensing the environment and communicating among them. We integrated CÉU with the *TinyOS* operating system [21] in order to use the abstracted radio services the operating system provides.

The second example uses CÉU with the SDL graphics library[4] under linux. With a more powerful platform, we can explore some simulation techniques that require fast processing.

The two demos also illustrate different ways to integrate CÉU with an underlying platform.

The applications are somewhat simple to fit the paper (ranging from 70 to 200 lines), but still complete enough to explore the programming techniques promoted by CÉU.

4.1 A Simple Network Protocol

The *Source Routing Protocol* for WSNs delivers packets from an origin node to a destination node [30]. The protocol stores the routing path in the data packet itself, as a vector of node addresses to traverse in sequence. Each hop in the path forwards the packet to the next address in the vector up to the final destination node. All nodes in the network play the role of *clients* and *forwarders* at the same time: a client periodically sends a packet to a destination node; a forwarder listen for a incoming packet from other nodes and forward it to the next node in the path. Each node has a single radio interface shared by all active roles. Multiple clients reside in the same node (defined statically), and, at a given time, multiple forwarders may also be active (depending on the dynamic network traffic).

TinyOS offers an event-driven API that relies on short-lived callbacks to remain responsive: "all long-latency operations are split-phase: operation request and completion are separate functions" [17]. This way, the original protocol implementation requires to track the state of active clients and forwarders with global vectors (for static clients) and allocated structures (for dynamic forwarders), so that they are accessible across separate functions in split-phase operations (e.g., timers and radio transmissions). The implementation also requires to manipulate a queue to multiplex transmissions from all clients and forwarders in the single radio.

The implementation in CÉU uses static organisms for clients and a dynamic pool for forwarders that adapt to the network traffic. The code in the left of Figure **??** shows the top-level block for the protocol. The `Client` and `Forwarder` classes are collapsed in lines 4-5 and expanded in the middle and right of the figure, respectively.

The input events `START` and `STOP` control the global state of the protocol (similarly to example of Figure): the top-level loop awaits the starting event (line 8) and, on request, starts the protocol (lines 12-25) and monitors the stopping event (line 10) in parallel. At any time a request to stop the protocol terminates the `par/or` (line 9), aborts all active clients and forwarders, restarts the loop (line 7), and waits for the next request to start. The input events `RECEIVE` and `SENDONE` (line 2) interface with the radio driver: `RECEIVE` notifies the program with a pointer to an incoming packet; `SENDDONE` acknowledges a packet transmission started with a `send_enqueue` call from the application.

The core of the protocol first creates a vector of clients and a pool to dynamically allocate forwarders (line 12-13). Then, it enters a loop to continuously receive packets as they arrive and take the proper action (lines 15-25): if the packet has no hops left, it reached the destination and the call to `_receive` passes the packet to the XXX (lines 17-18); otherwise, the protocol spawns a new forwarder that takes care of redirecting the packet to the next hop (19-24).

control the global state of the

one periodically transmits a status packet (lines 7-18); and one handles the remaining functionality of the protocol (collapsed in line 20).

[4] `http://www.libsdl.org`

SRP looks up the next hop in the source route in the data packet header and forwards the data packet to the next hop. Once the data packet reaches the destination, it is considered as being successfully delivered.

The routing information

through a static routing tables shared by all nodes

Specifically, source routing allows the origin node to specify the route the packet should take in the network. In this respect, source routing is considered as one implementation of the forwarding engine in network protocols.

Figure **??** shows part of our port of the SRP routing protocol [30] to CÉU. The protocol specifies a fixed number of *forwarders* responsible for routing received messages to neighbours based on a static table. Given that a forwarder holds internal state (i.e. a message buffer and the forwarding activity), we define a `Forwarder` class and create multiple instances to serve requests.

The first column of Figure **??** shows the receiving loop of the protocol, which invokes `emit go` when a message needs to be forwarded. The event is declared as global, so that `Forwarder` instances have access to it. The forwarders are declared in a vector, creating `COUNT` different instances. As the vector is local, all instances are automatically killed when the protocol is stopped. (Note the use of the start/stop pattern of Figure **??** again.)

The second column of Figure **??** shows the `Forwarder` class. Initially, all forwarders are in the same state, waiting for the global event `go`. Once the receiving loop emits the event in the top-level body, the forwarders awake in the order they were declared. The first forwarder atomically sets the `gotcha` variable, indicating that the message will be handled and that other forwarders should ignore it: all other forwarders will await again for the next `go` emission. With this technique, we eliminated the need of an explicit queue. In the case that all forwarders become busy, the `go` emission will be missed (with `gotcha=0`), acting just like a full queue.

Note that CÉU organisms are not global entities and do not use the heap for memory. Instead, they are bounded to the scope they are declared, and all memory is statically allocated, just like CÉU does for standard local variables. Also, when an organism goes out of scope, the same automatic bookkeeping of `par/or` compositions holds, all internal trails are killed and finalization blocks execute (if any). Hence, the "garbage collection" for both the memory and code in organisms is efficient and static. Although CÉU does not support dynamic creation (which could lead do unbounded memory), scoped organisms offer some degree of flexibility when compared to systems providing global objects only [6, 31].

No queues state vars incr. scope 30and same equivalent memory footprint (RAM-3, ROM+5)

4.2 Interactive Game

4.3 Discussion

5. Related work

Simula is a simulation language that introduced the concepts of objects and coroutines [13]. The syntactic structure of classes in Simula is very similar to CÉU, exposing an interface that encapsulates an execution body. However, the underlying execution models are fundamentally distinct: CÉU employs a reactive scheduler to resume trails based on external stimuli, while Simula relies on cooperation between processes (i.e., `detach` and `resume` calls, at the lowest level). Simula has no notion of compositions, with each process having a single line of execution. In particular, the lack of a `par/or` precludes orthogonal abortion and many derived CÉU features, such as lexically scoped organisms, finalization, and reference watching. Without scopes, Simula objects and processes have to live on the heap and rely on garbage collection. As far as we

```
input void    START,   STOP;
input _pkt_t* RECEIVE, SENDDONE;

<...> // "class Client"
<...> // "class Forwarder"

loop do
   await START;
   par/or do
      await STOP;
   with
      var  Client   [N_CLTS] clients;
      pool Forwarder[N_FWDS] forwarders;

      var _pkt_t* pkt;
      every pkt in RECEIVE do
         if pkt:left == 0 then
            _receive(pkt);
         else
            pkt:left = pkt:left - 1;
            spawn Forwarder with
               this.src := pkt;
            end;
         end
      end
   end
end
```

```
class Client with
do
   loop seq do
      par/and do
         await 1min;
      with
         var _pkt_t pkt;
         _pkt_setRoute(&pkt, seq);
         _pkt_setContents(&pkt, seq);

         loop do
            finalize
               _send_enqueue(&pkt)
            with
               _send_dequeue(&pkt);
            end

            var _pkt_t* done;
            done = await SENDDONE;
            if done == &pkt then
               break;
            end
            await (_rand()%100)ms;
         end
      end
   end
end
```

```
class Forwarder with
   var _pkt_t* src;
do
   var _pkt_t pkt;
   _memcpy(&pkt, src, src:len);
   loop do
      finalize
         _send_enqueue(&pkt)
      with
         _send_dequeue(&pkt);
      end

      var _pkt_t* done;
      done = await SENDDONE;
      if done == &pkt then
         break;
      end
   end
end
```

Figure 7. The Source Routing Protocol in CÉU.

know, Simula processes cannot be terminated explicitly from other processes.

Some previous work extend Esterel to provide dynamic synchronous abstractions [9–11]. In particular, ReactiveML [24] is a functional variant of Esterel with rich dynamic synchronous abstractions through *processes*. However, these languages rely on heap allocation and/or garbage collection and may not be suitable for constrained embedded systems. They also lack a finalization mechanism that hinders proper orthogonal abortion in the presence of stateful resources.

Finally, the main distinction to existing work is how CÉU incorporates to SRP the fundamental concept in SP of lexically scoped variables. All constructs of CÉU have a clear and unambiguous lifespan that can be inferred statically from the source code. Lexical scope permeates all aspects of the language: Any piece of data or control structure has a well-defined scope that can be abstracted as an organism and safely aborted through finalization. Even dynamic instances of organisms reside in pools with well-defined scopes with the same properties.

Functional Reactive Programming [32] contrasts with SRP as a complementary programming style for reactive applications. We believe that FRP is more suitable for data-intensive applications, while SRP, for control-intensive applications. For instance, describing a sequence of steps in FRP requires to encode explicit state machines so that functions can switch behavior depending on the current state. In contrast, FRP uses declarative formulas to specify continuous functions over time, such as for physics or data constraints among entities, while SRP requires explicit loops to update data dependencies continuously.

6. Conclusion

CÉU provides comprehensive support for structured reactive programming, extending classical structure programming with continuous interaction with the environment.

CÉU introduces organisms which reconcile data and control state in a single abstraction. In contrast with objects, organisms have an execution body that can react independently to stimuli from the environment. An organism body supports multiple lines of execution that can await events without loosing control context, offering an effective alternative to the infamous "callback hell". Both static and dynamic instances of organisms are subject to lexical scope with automatic memory management, which eliminates memory leaks and the need for a garbage collector.

CÉU is suitable for wide range of reactive applications and platforms. We have been experimenting with it in constrained platforms for sensor networks as well as in full-fledged computers and tablets for games and graphical applications[5]. CÉU successfully participated in the *Google Summer of Code*[6] with a student that had no previous experience with the language. We have also been teaching CÉU as an alternative language for sensor networks for the past two years in high-school and undergraduate levels. Our experience shows that students take advantage of the sequential style of CÉU and can implement non-trivial reactive programs in a couple of weeks.

References

[1] Erlang manual. `http://www.erlang.org/doc/reference_manual/processes.html` (accessed in Aug-2014).

[2] UNIX man page for pthread_cancel. man pthread_cancel.

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

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

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

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

[5] Uses of CÉU: `http://www.ceu-lang.org/wiki/index.php?title=Uses`

[6] LabLua GSoC'14: `http://google-opensource.blogspot.com/2014/08/google-summer-of-code-new-organizations.html`

[7] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *LNCS*, pages 72–93. Springer, 1993.

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

[9] F. Boussinot et al. Reactive objects. In *Annales des télécommunications*, volume 51, pages 459–473. Springer, 1996.

[10] F. Boussinot and L. Hazard. Reactive scripts. In *RTCSA'96*, pages 270–277. IEEE, 1996.

[11] F. Boussinot and J.-F. Susini. The sugarcubes tool box: A reactive java framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.

[12] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *PLDI'13*, pages 411–422, 2013.

[13] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[14] M. de Icaza. Callbacks as our generations' go to statement. `http://tirania.org/blog/archive/2013/Aug-15.html` (accessed in Aug-2014), 2013.

[15] E. W. Dijkstra. *Notes on structured programming*. Technological University Eindhoven, 1970.

[16] Elm Language Web Site. Escape from callback hell. `http://elm-lang.org/learn/Escape-from-Callback-Hell.elm` (accessed in Aug-2014).

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

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

[19] N. Halbwachs et al. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, September 1991.

[20] D. Harel and A. Pnueli. *On the development of reactive systems*. Springer, 1985.

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

[22] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[23] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[24] L. Mandel and M. Pouzet. Reactiveml: a reactive extension to ml. In *Proceedings of PPDP'05*, pages 82–93. ACM, 2005.

[25] L. A. Meyerovich et al. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[26] ORACLE. Java thread primitive deprecation. `http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html` (accessed in Aug-2014), 2011.

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

[28] G. Salvaneschi et al. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of Modularity'13*, pages 25–36. ACM, 2014.

[29] F. Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of SenSys'13*. ACM, 2013.

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

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

[32] Z. Wan and P. Hudak. Functional reactive programming from first principles. *SIGPLAN Notices*, 35(5):242–252, 2000.