Structured Synchronous Reactive Programming with Céu

Abstract

Structured synchronous reactive programming (SSRP) augments classical structured programming (SP) with continuous interaction with the environment. We advocate SSRP as viable in multiple domains of reactive applications and propose a new 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 in SP. 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, 18]. 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 [17] represents programs as graphs of values, in which a change to a node updates its dependencies automatically. Both styles rely on the *synchronous execution hypothesis* which states that the input and corresponded output in reactions to the environment are simultaneous because, in this context, internal computations should run infinitely faster than the rate of events [27].

In recent years, Functional Reactive Programming (FRP) [34] has modernized the dataflow style, inspiring a number of languages and libraries, such as Flapjax [24], 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, due to its heavy reliance on side effects [22, 25, 29]. 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`".[1][2]

We believe that all domains of reactive applications can benefit from the imperative style of Esterel, which we now refer to as *Structured Synchronous Reactive Programming (SSRP)*. SSRP extends the classical hierarchical control constructs of *Structured Programming (SP)* (i.e., concatenation, selection, and repetition [14]) to support continuous interaction with the environment. In contrast with FRP, SSRP retains structured and sequential reasoning of concurrent programs, 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, SSRP 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 SSRP, recovering sequential execution lost with the observer pattern. Parallel compositions[3] 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 [30] with a new abstraction mechanism, the *organisms*, that encapsulate parallel compositions with an object-like interface. In brief, organisms are to SSRP 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 procedure calls in multi-threaded applications, organisms have deterministic behavior and do not require explicit synchronization. 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, organisms are subject to lexi-

[1]"Callbacks as our Generations' Go To Statement": `http://tirania.org/blog/archive/2013/Aug-15.html`

[2]"Escape from Callback Hell": `http://elm-lang.org/learn/Escape-from-Callback-Hell.elm`

[3]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.

cal scope and automatic memory management for both static and dynamic instances, not relying on heap allocation at all.

The rest of the paper is organized as follows: Section 2 presents SSRP through CÉU, with its underlying synchronous concurrency model and parallel compositions. Section 3 describes the organisms abstraction with static and dynamic instantiation, lexical scope, and automatic memory management. Section 4 demonstrates two complete applications developed entirely with SSRP in CÉU: a network protocol for sensor networks and a video game for tablets. Section 5 discusses related work. Section 6 concludes the paper.

2. SSRP 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 defines an input event `RESET` (line 1), a shared variable `v` (line 2), and starts two trails with the `par` construct (lines 3-14): 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 directly by prefixing symbols with an underscore (e.g., `_printf(<...>)`, in line 7).

2.1 Synchronous concurrency

In CÉU, a program reacts to an occurring event completely 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[4]. 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 [30].

As a consequence of synchronous execution, all consecutive operations to shared variable `v` in Figure 1 are atomic (until reaching the next `await`) because reactions to events `1s` and `RESET` can never interrupt each other. In contrast, in asynchronous models with non-deterministic 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 SP with an orthogonal abortion construct that simplifies the composition of activities[5]. 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>
```

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 [6]. CÉU extends orthogonal abortion to also work 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 [6] 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. 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"* [21]. 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, SSRP 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 SSRP archetypes represent, respectively, the *sampling* and *timeout* patterns, which are typical of reactive applications.

The example in Figure 2 relies on hierarchical `par/or` and `par/and` compositions to describe the state machine of a data collection protocol for sensor networks [16, 30]. The input events `START`, `STOP`, and `RETRANSMIT` (line 1) represent the external interface of the protocol with a client application. The protocol enters

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

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

```
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                 par/and do
12                     await 1min;
13                 with
14                     <send—beacon—packet>
15                 end
16             end
17         end
18     with
19         <...> // the rest of the protocol
20     end
21 end
```

Figure 2. Parallel compositions can describe complex state machines.

```
var _pkt_t buffer;
<fill—buffer—info>
_send_enqueue(&buffer);
await SENDACK;
```

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

Figure 3. Finalization clauses safely release stateful resources.

the top-level loop and awaits the starting event (line 3). Once the client application makes a start request, the protocol starts three other trails: one monitors the stopping event (line 5); one periodically transmits a status packet (lines 7-17); and one handles the remaining functionality of the protocol (collapsed in line 19). The periodic transmission is another loop that starts two other trails (lines 8-16): one to handle an immediate retransmission request (line 9); and one that actually transmits the status packet (lines 11-15). The transmission (collapsed in line 14) 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 14, if not idle), and restarts the loop (line 7). The client may also 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 [30].

2.3 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 2 (line 14). 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 `SENDACK` acknowledging the packet transmission. In the meantime, the sending trail might be aborted by `STOP` or `RETRANSMIT` requests (lines 5 and 9 in Figure 2), 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 [30]. The code in the right of Figure 3 properly dequeues the packet when the block of `buffer` goes out of scope, i.e., the finalization clause (after the `with`) executes automatically on external abortion.

3. Organisms: SSRP 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:

CODE-1 blinks two LEDs with different frequencies in parallel and terminates after 1 minute.

CODE-2 abstracts the blinking LEDs in an organism class and uses two instances of it to reproduce the same behavior of *CODE-1*.

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 (in this example, 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 from organisms to 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 equivalent expansion of *CODE-3* suggests (and based on lexical scheduling described in Section 2.1).

The memory model for organisms is similar to stack-living local variables of procedures in SP, featuring 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 become inaccessible, but their bodies are automatically aborted, as the expansion of *CODE-3* makes clear: The `par/or` (lines 9-29)

```
par/or do
    loop do
        await 500ms;
        _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  = 500;
    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  = 500;
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
```

CODE-3: organisms expansion

Figure 4. Two blinking LEDs using organisms.

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), as discussed in Section 2.3. 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, the class `Unit` (lines 1-16) defines the position and destination properties `pos` and `dst` (lines 2-3), and the event `move` to listen for requests to move the unit position (line 4). The main program (lines 18-24) creates two units, requesting the first to move immediately to `dst=300`, and the second to move after 1 second to position `500`. On instantiation, the organism body 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 for 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.

```
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.dst = 300;
20 end
21
22 var Unit u2;
23 await 1s;
24 emit u2.move => 500;
```

Figure 5. Organism manipulation through events.

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 SSRP to handle virtual resources in reactive applications.

CÉU supports dynamic instantiation of organisms through the `spawn` primitive. The example that follows spawns a new instance of `Unit` (previously defined in Figure 5) on 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 of organisms. 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 the 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 to 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 (e.g., `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.

3.2 Pointer and references

As organisms react independently to the environment, it is often not necessary to manipulate 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;
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 an organism pointer (in the left) and the equivalent expansion (in the right).

```
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 pointer; 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 manage 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)
```

A more typical use of pointers to organisms is inside a *pool iterator* which acquire temporary pointers to all of its alive instances. To preserve pointer accesses safe, 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-u1-vs-u2> then
            emit u1:move => _rand() % 500;
            emit u2:move => _rand() % 500;
        end
    end
end
```

CÉU also provides references as a safer alternative to pointers. Unlike pointer attributions, references can only be associated to static organisms of (at least) the same scope. This way, references are guaranteed to be always valid and do not require `watching` supervision. Like in *C++*, references cannot be reassigned and, as simple aliases, they use the member operator '`.`' directly (instead of the dereference operator '`:`'):

```
var Unit& ref;
var Unit u1;
do
    var Unit u2;
    ref = u2;    // illegal (scope of u2 < ref)
end
ref = u1         // legal   (scope of u1 >= ref)
ref.pos = 0;     // affects u1
```

```
1  interface IUnit with
2      var   int pos;
3      var   int dst;
4      event int move;
5  end
6
7  class Archer with
8      interface IUnit;
9      <...>   // other interfaces
10 do
11     <...>   // execution body
12 end
13
14 class Knight with
15     interface IUnit;
16     <...>   // other interfaces
17 do
18     <...>   // execution body
19 end
20
21 pool IUnit[] units;
22 <...>
23 spawn Archer in units;
24 spawn Knight in units;
25 <...>
26 loop (IUnit*)u in units do
27     emit u:move => _rand() % 500;
28 end
```

Figure 7. Organism manipulation through abstract interfaces.

3.3 Abstract interfaces

In order to allow multiple classes with similar interfaces to cooperate, CÉU provides abstract interfaces similar to Java interfaces. Like classes, abstract interfaces declare public variables, method signatures, and internal events; but unlike classes, they do not define an execution body.

The code in Figure 7 declares the `IUnit` interface (lines 1-5) and the classes `Archer` and `Knight` that implement it (lines 7-12 and 14-19, respectively). Each concrete class may expose additional fields and implement other abstract interfaces (hidden in lines 9 and 16). The body implementation for knights and archers are presumably different (hidden in lines 11 and 18), with particular moving animations, attacking behavior, etc. The main program defines an unbounded pool of `IUnit` instances (line 21), which is further populated with archers and knights (lines 23-24). At some point, the pool is traversed and moves all units to random positions (lines 26-28), each respecting its actual implementation.

Interfaces are always manipulated through pointers and are also subjected to the pointer analysis described in Section 3.2.

4. Applications

We present two complete applications in different domains and platforms to argue that SSRP in CÉU can be generally adopted in the context of reactive systems. The examples combine all discussed functionality to build complete reactive applications in a structured way.

The first application is a simple network protocol for wireless sensor networks (WSNs). We integrated CÉU with the *TinyOS* operating system [20] which targets highly constrained ATmega embedded platforms[6] (e.g., 8-bit CPU with 4Kb of SRAM).

The second application is a casual two-player game for tablets published in the "Google Play" store[7]. In this case, we use the multi-platform SDL graphics library[8] targeting Android devices with extensive memory and CPU power.

Each platform provides an environment with different input sources (e.g., radio transceivers and touch screens), which are exposed as input events to the program. Regardless of the capabilities gap between the platforms, we use the same programming techniques in the examples, such as dynamic organisms and deep nesting of parallel compositions.

4.1 Source Routing Protocol

The *Source Routing Protocol* for WSNs [32] delivers packets from an origin to a destination node. The protocol stores the routing path in the 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 may 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 an incoming packet from other nodes and forward it to the next node in the path. A node has a single radio interface shared by all of its active roles.

TinyOS offers an event-driven API that relies on short-lived callbacks to keep nodes responsive, sharing the same limitations with the observer pattern: "all long-latency operations are split-phase: operation request and completion are separate functions" [15]. This way, the original protocol implementation keeps the state of active clients and forwarders in static- or heap-living globals to be accessible across separate functions in split-phase operations (e.g., timer request and expiration). Clients are defined statically in a global vector, while forwarders are dynamically allocated in the heap to adapt to the network traffic.

The implementation in CÉU takes advantage of SSRP and overcomes split-phase operations with simple sequential flow separated by `await` statements. It uses static organisms for clients and a dynamic pool for forwarders to behave like the original implementation. The file *main.ceu* in Figure 8 shows the main body for the protocol in CÉU. The `Forwarder` and `Client` classes are included in lines 6-7 and expanded in the right of the figure.

The input events `START` and `STOP` (lines 1-2 of *main.ceu*) control the global state of the protocol (similarly to example of Figure 2): the top-level loop awaits the starting event (line 10) and, on request, "watches" the stopping event (line 11) while executing the protocol (lines 12-25). The `watching` construct (described in Figure 9.1) aborts its nested block on the occurrence of the referred event (and is more idiomatic than the equivalent expansion to a `par/or`). At any time, a request to stop the protocol aborts all active clients and forwarders, restarts the loop (line 9), and waits for the next request to start (line 10).

The input events `RECEIVE` and `SENDACK` (lines 3-4) represent the interface with the radio driver: `RECEIVE` notifies the program of an incoming packet (carrying a pointer to it); `SENDACK` acknowledges that a previous call to `send_enqueue`, which requests a packet transmission, has completed (carrying its identifying pointer).

The core of the protocol (lines 12-25) first declares a pool of forwarders and a vector of clients (line 12-13). The identifier '_' for static instances makes them anonymous, which is recommended for organisms that the application does not manipulate directly: each client in the vector executes independently in parallel, frequently sending packets to other nodes in the network. The protocol then enters the `every` construct (described in Figure 9.2) to receive incoming packets continuously and take the proper action as they arrive (lines 15-25): if the packet has no hops left, it reached the destination and the protocol calls the user-defined function `_receive` to handle the packet (lines 17-18); otherwise, the protocol sets the

[6] ATmega micro-controllers: `http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx`

[7] The "Rocks!" game: `https://play.google.com/store/apps/details?id=org.droid_in_the_sky.rocks`

[8] The SDL library: `http://www.libsdl.org`

```
1  input void    START;
2  input void    STOP;
3  input _pkt_t* RECEIVE;
4  input _pkt_t* SENDACK;
5
6  #include "forwarder.ceu"
7  #include "client.ceu"
8
9  loop do
10     await START;
11     watching STOP do
12         pool Forwarder[N_FWDS] forwarders;
13         var  Client   [N_CLTS] _;
14
15         var _pkt_t* inc;
16         every inc in RECEIVE do
17             if inc:hopsLeft == 0 then
18                 _receive(inc);
19             else
20                 _pkt_setNextHop(inc);
21                 spawn Forwarder with
22                     _memcpy(&this.out, inc, inc:len);
23                 end
24             end
25         end
26     end
27 end
```

File ”main.ceu”

```
1  class Forwarder with
2      var _pkt_t out;
3      event void ok;
4  do
5      loop do
6          var bool enq;
7          finalize
8              enq = _send_enqueue(&this.out)
9          with
10             _send_dequeue(&this.out);
11         end
12         if not enq then
13             await 50ms;
14             continue;
15         end
16         var _pkt_t* done = await SENDACK
17                            until (done == &this.out);
18         emit this.ok;
19         break;
20     end
21 end
```

File ”forwarder.ceu”

```
1  #include "forwarder.ceu"
2
3  class Client with
4  do
5      loop seqno do
6          par/and do
7              await 1min;
8          with
9              do Forwarder with
10                 _pkt_set(&this.out, seqno);
11             end
12         end
13     end
14 end
```

File ”client.ceu”

Figure 8. The Source Routing Protocol in CÉU.

next hop (line 20) and spawns a new forwarder to re-transmit the packet (lines 21-23). Note that multiple forwarders may coexist if incoming packets arrive faster than their re-transmissions.

The file *forwarder.ceu* in Figure 8 defines the `Forwarder` class which exposes the packet `out` to transmit, and the event `ok` to signal its completion (lines 2-3). The class body is a loop that lasts until the packet is successfully transmitted (lines 5-20): the enqueue operation is properly finalized with a corresponding dequeue (lines 7-11, as described in Section 2.3); if the queue is full, the forwarder waits for a short period and restarts the loop to retry (lines 12-15); otherwise, the body awaits until the radio driver acknowledges the transmission of the enqueued packet (`done==&out` in lines 16-17), signals the completion to potential listeners through the event `ok` (line 18, discussed further), and escapes the loop to terminate (line 19). The `await-until` construct (described in Figure 9.3) expands to a loop that checks every occurrence of the referred event, escaping when the condition is met. Note that for forwarders spawned in the main body (line 21 in *main.ceu*), the organism termination automatically frees it from memory, as discussed in Section 3.1.

The file *client.ceu* in Figure 8 defines the `Client` class with an empty interface, meaning that it does not depend on data from the main body. The class body (lines 5-13) is an infinite loop that sends a new packet every minute. The loop automatic variable `seqno` (line 5), which starts at `0` and is incremented on each loop iteration, represents the packet sequence number. The `par/and` (lines 6-12) restricts the loop iteration to occur at least once every minute, just like the sampling pattern of Section 2.2. To send a new packet, the body embeds a `Forward` organism and sets the contents of the `out` packet to depend on the current sequence number and a user-defined function (lines 9-11). The `do-<class>` construct (described in Figure 9.4) expands to a static organism declaration that awaits its own termination. The expansion expects the organism interface to define the event `ok` and the body to emit it on completion (e.g., that the transmission succeeded). The expansion is enclosed with an explicit `do-end` block to make the organism to go out of scope after the `await ok`, and proceed to the statement in sequence.

The `Forwarder` and `Client` classes illustrate how SSRP can break programs into modules that encapsulate, at the same time, data and execution control (e.g., message buffers and associated transmission activities). These modules are then instantiated inside the main execution body in a deliberate depth that restricts their scope. For instance, at any time, a `STOP` request terminates the `wacthing` construct (lines 11-26 of *main.ceu*) and aborts all clients and forwarders. In spite of being defined in separate files, possibly by different developers, all clients and forwarders are automatically finalized to a consistent state (lines 7-11 of *forwarder.ceu*).

The example also shows how organisms can nest to avoid code duplication: To forward a packet, the `Client` class “calls” a `Forwarder` organism (lines 9-11 of *client.ceu*) that behaves like a “reactive subroutine”.

In the protocol, the radio driver is subjected to a high degree of concurrency, with incoming packets to the node, requests from local clients, and forwarders routing data through the network. However, the execution model of CÉU synchronizes all reactions to radio events and eliminates race conditions by design, such as on calls to side-effect functions `_receive` and `_send_enqueue` (line 18 of *main.ceu* and line 8 of *forwarder.ceu*, respectively).

```
// a "watching"
watching <evt> do
    <...>
end

// expands to a "par/or"
par/or do
    await <evt>;
with
    <...>
end
```

(1) A `watching` expands to a `par/or`.

```
// an "every"
every v in <evt> do
    <...>
end

// expands to a loop
loop do
    v = await <evt>;
    <...>
end
```

(2) An `every` expands to a `loop`.

```
// an "await-until"
v = await <evt>
    until <condition-that-may-use-"v">;

// expands to a loop to meet the condition
loop do
    v = await <evt>;
    if <condition-that-may-use-"v"> then
        break;
    end
end
```

(3) An `await-until` expands to a `loop` to meet the condition.

```
// a "do-class"
do <Class> with
    <...>
end

// expands to a "do-end"+instantiation+"await ok"
do
    var <Class> org with
        <...>
    end
    await org.ok; // <Class> must implement and emit
end           //          the event "ok"
```

(4) A `do-<class>` waits for the organism to terminate.

Figure 9. Syntactic sugars to describe typical control patterns in SSRP.

4.2 The "Rocks" game

The "Rocks!" game of Figure 10 is a spaceship shooter that two opponents play simultaneously on the same tablet. Each player controls a spaceship by swiping and tapping different areas of the screen: a swipe changes the ship acceleration to follow it; a tap fires in the direction of the opponent. At random periods, a new meteor enters the screen and moves to a random position. If a ship collides with a meteor or an opponent shot, it explodes and the game restarts. The scores on the bottom of the screen show the opponents' number of deaths.

The game is considerably more complex than the protocol of Section 4.1, so we generally describe the abstract interfaces and focus on the top-level block which puts all pieces together. Figure 11 shows the relevant parts of the main file. The game interacts with 4 input events (lines 1-4): `SDL_QUIT` requests the application to terminate (e.g., user closes the game window); `SDL_DT`, which is the source of all animations, is emitted on every frame

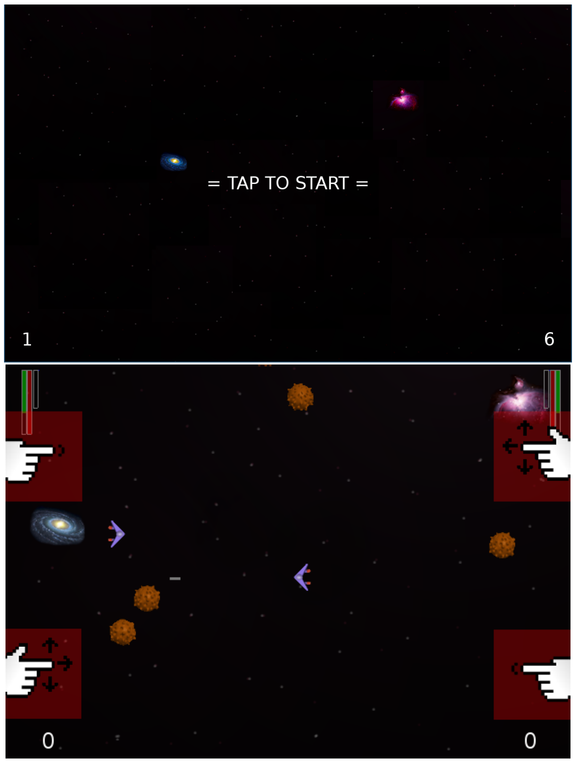

Figure 10. Screenshots for the game "Rocks!": starting screen and gameplay with the player controls highlighted.

passing the number of milliseconds elapsed since the previous frame; `SDL_REDRAW` requests screen updates and is also emitted on every frame; `SDL_TOUCH` signals screen touch events (i.e., tapping and swiping). The interfaces (lines 6-27), with implementations included in corresponding files (lines 29-32), represent the abstract concepts that the main program manipulates:

`IScore` (lines 6-9) represents the player scores, expecting a `position` in the screen (line 7), and exposing the event `go_increment` which the application emits to increment the score (line 8). The actual implementation holds the score current state (e.g., points and graphical texture) and knows how to redraw itself on screen in reactions to `SDL_REDRAW`.

`IController` (lines 11-14) represents the player controllers, exposing the current acceleration `ax`/`ay` charged to the ship (line 12), and emitting `ok_shoot`[9] events which the application awaits to spawn new shots (line 13). The actual implementation updates the acceleration according to reactions to the relevant `SDL_TOUCH` inputs.

`ICollidable` (lines 16-20) represents all objects that require collision detection: the ships, shots, and meteors. The field `id` (line 17) identifies the object when applying collisions (e.g., collisions between two meteors are ignored). The field `rect` (line 18) exposes the current position and dimension for collision detection. The application emits the `go_hit` event to signal that the object has collided (line 19).

`IShip` (lines 22-27) represents the ships and extends the `ICollidable` interface (line 23). A ship expects a `controller` reference (line 24) and holds a pool of 3 shots (line 25, described further). It also emits the event `ok_destroyed` after dyeing (line 26).

[9] The prefixes `go_` and `ok_` for internal events informally represent direction: the application emits `go` events to interfaces, and awaits `ok` events emitted from them.

```
 1  input void SDL_QUIT;
 2  input int  SDL_DT;
 3  input void SDL_REDRAW;
 4  input _SDL_TouchFingerEvent* SDL_TOUCH;
 5
 6  interface IScore with
 7      var   _SDL_Point position;
 8      event void       go_increment;
 9  end
10
11  interface IController with
12      var   float ax, ay;
13      event void  ok_shoot;
14  end
15
16  interface ICollidable with
17      var   int       id;
18      var   _SDL_Rect rect;
19      event void      go_hit;
20  end
21
22  interface IShip with
23      interface ICollidable;
24      var   IController& controller;
25      pool  Shots[3]     shots;
26      event void         ok_destroyed;
27  end
28
29  #include "scores.ceu"
30  #include "controllers.ceu"
31  #include "collidables.ceu"
32  #include "ship.ceu"
33
34  par/or do
35      await SDL_QUIT;
36  with
37      every SDL_REDRAW do
38          _SDL_RenderCopy(<bg−image>,<center>);
39      end
40  with
41      var Score score1 with
42          this.position = <bottom−right>;
43      end
44      var Score score2 with
45          this.position = <bottom−left>;
46      end
47
48      loop do
49          do
50              <...> // "TAP TO START" message
51          end
52          do
53              <...> // gameplay with ships & meteors
54          end
55      end
56  with
57      every SDL_REDRAW do
58          _SDL_RenderPresent(<...>);
59      end
60  end
```

Figure 11. File ”main.ceu” with the top-level block of the game.

The body of the game is a `par/or` (lines 34-60) that terminates on a `SDL_QUIT` request (line 35). Reactions to `SDL_REDRAW` relies on deterministic scheduling respecting the lexical order of trails: first, the trail in lines 37-39 redraws the background; then, all organisms inside the trail in lines 41-55 have the chance to redraw themselves (i.e., scores, ships, etc.); finally, the trail in lines 57-59 updates the screen.

The core of the game resides in the trail in lines 41-55. We first instantiate the scores as static organisms that survive the whole game (lines 41-46). Then, we switch between the starting tap and gameplay behaviors of Figure 10, putting them in sequence and surrounded with loop (lines 48-55). The `do-end` enclosing each behavior isolates one from another, i.e., no variables or organisms should survive when switching between them.

```
 1  watching SDL_TOUCH do
 2      loop do
 3          await 500ms;        // message off
 4          watching 500ms do   // message on
 5              every SDL_REDRAW do
 6                  _SDL_RenderCopy(<tap−msg>,<center>);
 7              end
 8          end
 9      end
10  end
```

Figure 12. The starting tap behavior in ”main.ceu”.

```
 1  var TouchController controller1 with
 2      this.move_region = <swipe−region−1>;
 3      this.fire_region = <tap−region−1>;
 4  end
 5  var TouchController controller2 with
 6      this.move_region = <swipe−region−2>;
 7      this.fire_region = <tap−region−2>;
 8  end
 9
10  var Ship ship1 with
11      this.id         = SHIP1;
12      this.rect       = <pos−dim−1>;
13      this.controller = controller1;
14  end
15  var Ship ship2 with
16      this.id         = SHIP2;
17      this.rect       = <pos−dim−2>;
18      this.controller = controller2;
19  end
20
21  pool Meteor[] meteors; // Meteor in collidables.ceu
22
23  par/or do
24      every (1000 + _rand()%2000)ms do
25          spawn Meteor in meteors;
26      end
27  with
28      every SDL_DT do
29          loop (ICollidable*) c1 in <all> do
30              loop (ICollidable*) c2 in <all> do
31                  if collides(c1,c2) and
32                     enemies(c1,c2) then
33                      emit c1:go_hit;
34                      emit c2:go_hit;
35                  end
36              end
37          end
38      end
39  with
40      await ship1.ok_destroyed;
41      emit points2.go_inc;
42  with
43      await ship2.ok_destroyed;
44      emit points1.go_inc;
45  end
```

Figure 13. The gameplay behavior in ”main.ceu”.

The behavior for the starting tap, expanded in Figure 12, is to blink the *“TAP TO START”* message until the user taps the screen. The `watching` composition terminates on the occurrence of `SDL_TOUCH` (line 1), switching to the gameplay (lines 52-54 of Figure 11). In the meantime, the loop (lines 2-9) blinks the text on screen by alternating between an idle period of 500ms (line 3), and a displaying period of 500ms (lines 4-8).

The code for the gameplay is expanded in Figure 13. First, we declare the two controller organisms (lines 1-8), passing the move and fire regions in which they operate inside the screen (as illustrated in Figure 10). After the declarations, each controller body reacts to `SDL_TOUCH` swipe and tap events to either update its `ax` and `ay` fields, or emit `ok_fired` events to the application, as specified by the `IController` interface. Then, we declare the two

```
1  class Ship with
2      <...>   // see IShip
3  do
4      par/or do
5          var int hits = 3;
6          every this.go_hit do
7              hits = hits - 1;
8              if hits == 0 then
9                  break;
10             end
11         end
12     with
13         every this.controller.ok_fired do
14             spawn Shot in this.shots with
15                 <...>   // direction, speed
16             end
17         end
18     with
19         var float vx = 0;
20         var float vy = 0;
21         var int dt;
22         every dt in SDL_DT do
23             vx = vx + this.controller.ax*dt;
24             vy = vy + this.controller.ay*dt;
25             this.rect.x = this.rect.x + vx*dt/1000;
26             this.rect.y = this.rect.y + vy*dt/1000;
27         end
28     with
29         every SDL_REDRAW do
30             _SDL_RenderCopy(<image>,<pos>);
31         end
32     end
33     emit ok_destroyed;
34 end
```

Figure 14. The `Ship` class in "ship.ceu".

ship organisms (lines 10-19), specifying the collision identifiers, starting position and dimensions, and controllers for each ship. A ship knows how to move according to `SDL_DT` frame events and its controller acceleration, as well as to redraw itself on screen on every `SDL_REDRAW` request. New meteors are spawned at most every 2 seconds (lines 24-26) and reside in a dedicated pool (line 21). The meteors move to random positions and terminate themselves when leaving the screen, being automatically removed from the pool. On every frame, we check for collision between all `ICollidable` organisms in the game, two by two, including all meteors, ships, and shots (lines 28-38)[10]: we compare their `rect` and `id` fields (inside `collides` and `enemies` calls in lines 31-32) and, if it is the case, we signal both instances that they have collided (lines 33-34). The gameplay terminates when either of the ships signal its destruction (line 40 or 43). In this case, the program increments the enemy points (line 41 or 44) and aborts the whole `par/or`, switching back to the starting tap behavior.

Remind that a `do-end` encloses the gameplay behavior and isolates its state from the starting tap behavior (lines 52-54 of Figure 11), making the controllers, ships, shots, and all meteors to go out of scope. However, the `Score` organisms have a broader scope and persist for the whole game session (lines 41-46 of Figure 11).

The code for the `Ship` class is presented in Figure 14. A ship has to be inflicted 3 hit points to be destroyed (lines 5-11). Once this happens, the enclosing `par/or` terminates (lines 4-32) and emits the `ok_destroyed` event to the application (line 33), which makes the gameplay to also terminate (lines 40-44 of Figure 13). Whenever the controller senses a tap and emits the `ok_fired` event (lines 13-17), the ship spawns a new `Shot` organism in the direction of the opponent. The pool of shots can only hold 3 simultaneous instances, meaning that further `spawn` invocations fail until one of the shots leaves the screen (forcing the player to fire wisely). For the ship movement (lines 19-27), we integrate its position with respect to the `dt` "delta time" acquired every frame, also taking into account the current controller acceleration. Finally, we redraw the ship image in its current position for every `SDL_REDRAW` request (lines 29-31).

[10]The real code (which is considerable more complex in this case) avoids comparing each pair twice and uses a temporary helper vector to hold all static and dynamic `ICollidables` in the same structure (illustrated by the <`all`> placeholder in lines 29-30).

```
1  class Missile with
2      var Meteor* to_follow; // closest meteor
3      <...>
4  do
5      par/or do
6          watching this.to_follow do
7              every SDL_DT do
8                  if rect.y > to_follow:rect.y then
9                      this.ay = -0.1;
10                 else
11                     this.ay =  0.1;
12                 end
13             end
14         end
15     with
16         <...>   // missile moving, redrawing, etc
17     end
18 end
```

Figure 15. The guided missile watches a pointer to a meteor.

The complete game is below 500 lines of code and includes sound effects, explosions, power-up bonuses, guided missiles, indicative bars for hit points and available shots, among other features. For the guided missiles, for instance, we use the `watching` construct to track the closest meteor that approaches the ship, as the code in Figure 15 shows. The guided missile adjusts its acceleration on every frame (lines 7-13), based on the target position. If the supervised meteor disappears (or is destroyed), the `watching` construct terminates (lines 6-14) and aborts the `par/or` (lines 5-17), making the guided missile to self-destruct.

4.3 Discussion

SSRP and CÉU rely on control compositions to express program flow concisely and in a structured way. In contrast, solutions based on the observer pattern rely on explicit manipulation of global variables and state machines for flow control across reactions. The presented applications express all control patterns exclusively with hierarchical compositions of activities and organisms, corroborating our previous experiments towards the eradication of state variables [30], In addition, control compositions enable the conception of new higher-level self-contained constructs, such as the syntactic sugars of Figure 9 used in the applications.

Organisms are fundamental to expand SSRP to the development of more complex systems. Organisms tie data and control together to form abstractions that can be deployed deliberately to a limited scope in applications. In the protocol, the `Forwarder` and `Client` classes are defined in separate and applied to a restricted scope that does not mingle with the rest of the code. Furthermore, the example uses anonymous (static and dynamic) instances that are completely autonomous and do not require explicit manipulation from the main program. Finally, requests to stop the protocol trigger orthogonal abortion and finalization for all clients and forwarders, leaving the memory in a consistent state.

The distinction between static and dynamic organisms reflects more precisely the life cycle of each component in an application. In the game, the two ships are declared static because they are alive during the whole scope of the gameplay; the meteors, in contrast, are declared dynamic because they terminate when disappearing from the screen, which is a runtime condition. Nonetheless dynamic instances must still reside in static pools, also restricting them to a maximum lexical scope. Even though this distinction exists in O.O. languages like C++, it does not apply effectively because the dependency in short-lived callbacks for reactive applications require all allocations to be dynamic in practice.

Static scopes helps developers to better understand the overall state of memory in applications: blocks in parallel coexist in memory; blocks in sequence do not; static organisms and pools coexist with their enclosing block. This information is more difficult to infer when deallocation is explicit or relies on garbage collection, because both cases depend on runtime behavior that cannot be analyzed statically. In the protocol, after a stop request, it is guaranteed that no forwarders or clients are in memory. In the game, the memory for the starting screen and gameplay can never coexist because they are enclosed by blocks in sequence.

Explicit reference manipulation may introduce memory leaks even in garbage collected languages [19]. For instance, a *lapsed listener* [25] is a pointer to an object—supposedly unreferred—that is not explicitly unregistered as a listener from an event, preventing the garbage collector to release the object from memory. When absolutely necessary, explicit manipulation in CÉU must be protected with the `watching` construct, which ensures that pointers to organisms do not hold them in memory. In the game, the code for the guided missile that manipulates a reference to the closest meteor is automatically aborted when the latter goes out of scope.

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 (i.e., `detach` and `resume` calls, at the lowest level). Simula has no notion of compositions, with each object 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 have to live on the heap and rely on garbage collection.

Some previous work extend Esterel to provide dynamic synchronous abstractions [9–11]. In particular, ReactiveML [23] 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 SSRP 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 scoped pools with the same properties.

Functional Reactive Programming [34] contrasts with SSRP as a complementary programming style for reactive applications. We believe that FRP is more suitable for data-intensive applications, while SSRP, for control-intensive applications. On the one hand, FRP uses declarative formulas to specify continuous functions over time, such as for physics or data constraints among entities, while SSRP requires explicit loops to update data dependencies continuously. On the other hand, 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 the asynchronous spectrum of concurrency, a number of actor-based languages extend objects with independent execution contexts that communicate exclusively through message passing [7, 28, 31, 33]. On the one hand, the inherent nondeterministic execution of actors demands full state isolation which makes distribution and many-core parallelism more straightforward. On the other hand, the implicit synchronization in CÉU provides safe data sharing and global consensus about the overall state of the system, enabling abortion and lexical scopes for compositions.

6. Conclusion

CÉU provides comprehensive support for structured synchronous 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[11]. 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] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *LNCS*, pages 72–93. Springer, 1993.

[7] B. Bloom et al. Thorn: robust, concurrent, extensible scripting on the jvm. In *ACM SIGPLAN Notices*, volume 44, pages 117–136. ACM, 2009.

[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.

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

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

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

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

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

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

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

[19] E. Henry and E. Lycklama. How do you plug Java memory leaks? *Dr. Dobb's Journal of Software Tools*, 25(2):115–119, 121, Feb. 2000.

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

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

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

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

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

[25] R. Nystrom. *Game Programming Patterns*. ISBN: 978-0-9905829-0-8 `http://gameprogrammingpatterns.com/` (to appear).

[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] H. Rajan et al. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

[29] 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.

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

[31] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP 2010–Object-Oriented Programming*, pages 275–299. Springer, 2010.

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

[33] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

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