Structured Reactive Programming

Francisco Sant'Anna Noemi Rodriguez Roberto Ierusalimschy
Departamento de Informática — PUC-Rio, Brasil
{fsantanna,noemi,roberto}@inf.puc-rio.br

ABSTRACT

....
....
....
....
....
....
....
....
....

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. They represent a wide range of software areas and platforms: from games in powerful desktops, "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 [3, 17]: The imperative style of Esterel [5] organizes programs with structured control flow primitives, such as sequences, repetitions, and parallelism. The dataflow style of Lustre [12] represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming [20] modernized the dataflow style and became mainstream, deriving a number of languages and libraries, such as Flapjax [15], Rx (from Microsoft), React (from Facebook), and Elm [6]. In contrast, the imperative style did not follow this trend and is now 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 [14, 18]. However, short-lived callbacks (i.e., the observers) eliminate any vestige of structured programming, such as support for lasting loops and automatic variables [2], which are elementary capabilities of imperative languages. In this sense, the observer pattern actually disrupts imperative reactivity, becoming "our generation's `goto`" [8, 7, 10].

In this work, we revive the imperative style of Esterel, which we now refer as *Structured Reactive Programming (SRP)*. SRP extends the classical hierarchical control constructs of *Structured Programming (SP)* (i.e., concatenation, selection, and repetition [9]) to support continuous interaction with the environment. In practical terms, we consider that SRP must provide at least two fundamental extensions to SP: an "`await <event>`" statement that suspends a line of execution until the referred event occurs, keeping the whole data and control context alive; and parallel constructs that compose multiple lines of execution and make them concurrent. The `await` statement captures the imperative and reactive nature of SRP, restoring sequential execution and support for automatic variables. Parallel compositions allow for multiple `await` statements to coexist, which is necessary to handle concurrent events common in reactive applications.

TODO: present? We promote the use of the language Céu [19], a contemporary outlook of SRP that aims to expand it from the rigid embedded domain. Céu is based on Esterel and relies on a similar synchronous and deterministic execution model that simplifies the reasoning about concurrency aspects. Our main contribution are the *organisms*, a new abstraction mechanism to support parallel and `await` control structures and offer an object-like interface that other parts of the program can manipulate. In brief, organisms are to SRP like procedures are to SP, i.e., one can abstract a portion of code with a name and "call" that name from multiple places. There are, however, some semantic challenges that apply to organisms:

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

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

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

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

// ORGANISMS
class <T> with
    <interface>
do
    <body>
end
```

Figure 1: Syntax of Céu.

- Organisms are themselves alive and concurrent, acting as subprograms with continuous data and control state.
- Organisms are part of a concurrent program that can manipulate and affect their data and execution state.
- Organisms can be dynamically allocated, requiring a memory management model that must also apply to embedded systems.

The rest of the paper is organized as follows: Section 2 presents SRP through Céu, with its underlying synchronous concurrency model, powerful parallel compositions, and the organisms abstraction. Section 3 discusses related work. Section 4 concludes the paper and makes final remarks.

2. STRUCTURED REACTIVE PROGRAMMING WITH CÉU

Céu is a Esterel-based synchronous language, in which its lines of execution, known as *trails*, react continuously to external stimuli. Figure 1 shows a compact reference of Céu.

The introductory example of Figure 2 starts two trails with the `par` statement: the first increments variable `v` on every second and prints it on screen; the second resets `v` on every `RESET` external request. The input event `RESET` (line 1) represents the interface of the program with external requests. The loop in the first trail (lines 4-8) continuously waits for 1 second, increments variable `v`, and prints its current value. The loop in the second trail (lines 10-13) resets `v` on every occurrence of `RESET`. Programs in Céu can access *C* libraries available in the underlying platform by prefixing symbols with an underscore (e.g., `_printf(<...>)`).

2.1 Synchronous concurrency

```
1  input void RESET;   // declares an external event
2  var int v = 0;
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 2: Introductory example in Céu.

In the synchronous execution model of Céu, a program must react 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[1]. 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 have `await` statements on their bodies [19].

In the example of Figure 2, accesses to variable `v` are always atomic because reactions to `1s` and `RESET` can never interleave. In contrast, in an asynchronous model 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 aspects under the synchronous execution model.

Above all, the synchronous model empowers SRP with an *orthogonal abortion* construct. Orthogonal abortion is the ability to abort an activity [2] from outside it, without affecting the overall consistency of the system (e.g., properly releasing global resources). Figure 3 shows the hypothetical construct `par/or` that composes concurrent activities and rejoins when either of them terminates, properly aborting the other. The `par/or` is regarded as orthogonal because the composed activities do not know when and how they are aborted (i.e., abortion is external to them). In the example, each activity 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, supposedly reusing the space from the activities' locals going out of scope.

Abortion in asynchronous languages is challenging [4], because the activity to be aborted might be on a inconsistent state (e.g., holding locks or pending messages). The language runtime has to await the activity to be in a consistent state before aborting it, resulting in two unsatisfactory semantics for a `par/or` construct: either wait to rejoin, mak-

[1]The actual implementation enqueues incoming input events to process them in the incoming reactions.

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

```
par/or do
    // activity A
    <local—variables>
    <body>
with
    // activity B
    <local—variables>
    <body>
end
<local—variables>
```

Figure 3: A par/or composes two concurrent activities and rejoins when one terminates, aborting the other.

ing the program unresponsive in the meantime; or rejoin immediately and wait in the background, which may confuse the programmer (which assumes both activities have terminated), and also cause race conditions with the `par/or` continuation. Another problem is that local variables need to coexist in memory for some time, moving the language allocation strategy to the heap (which is discouraged in the context of embedded systems).

As matter of fact, asynchronous languages do not provide effective abortion: CSP only supports a composition operator that "terminates when all of the combined processes terminate" [13]; Java's `Thread.stop` primitive has been officially deprecated [16]; and pthread's `pthread_cancel` does not guarantee immediate cancellation [1]. Instead, asynchronous activities typically agree a on common protocol to abort each other (e.g., through shared state variables or message passing).

Synchronous languages, however, provide accurate control over the life cycle of concurrent activities, because in between every reaction, the whole system is idle and consistent [4]. Céu provides the presented `par/or` construct, which is equivalent to Esterel's `trap` abortion construct. In Section 2.3 we highlight the importance of abortion for the organisms abstraction.

2.2 Parallel compositions

In terms of control structures, the basic extension of SRP is the support for parallel compositions, allowing applications to handle multiple events concurrently. Céu provides three parallel constructs, which 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 example of Figure 4 compares the `par/and` and `par/or` compositions. The code `<...>` represents a complex operation that takes time to complete. In the `par/and` variation, the operation repeats every second at minimum, as both sides must terminate before re-executing the loop. In the `par/or` variation, if the block 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 code in Figure 5 describes part of a protocol for sensor networks ported to Céu [11, 19], which relies on `par/or` and `par/and` compositions to describe its state machine. The input events `START`, `STOP`, and `RETRANSMIT` (line 1) represent

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

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

Figure 4: The *sampling* and *timeout* patterns using parallel compositions.

```
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 5: Parallel compositions can describe complex state machines.

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 a start request, the protocol starts three other trails: one to await the stopping event (line 5); one to periodically transmit a status packet (lines 7-18); and one with the remaining functionality of the protocol (collapsed in line 20). As compositions can be nested, 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. 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. At any time, the client may request a retransmission (line 9), which terminates the `par/or` (line 8), aborts the ongoing transmission (if not idle), and restarts the loop (line 7). Also, the client may request to stop the protocol (line 5), which terminates the outermost `par/or`, aborts the transmission, the retransmission handling, and the rest of the protocol. In this case, the top-level loop restarts and waits for the next request to start the protocol (line 3), remaining unresponsive to other requests.

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 [19].

2.2.1 Finalization

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

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

Figure 6: Finalization clauses safely release low-level resources.

The Céu compiler tracks the interaction of `par/or` compositions with automatic variables and stateful *C* functions (e.g., device drivers) to force proper finalization.

Consider the code on the left of Figure 6, which sends the status packet of Figure 5 (collapsed in line 15). The call to `_send_enqueue` enqueues a pointer to the local packet buffer in the radio driver and waits for a `SEND_ACK` that acknowledges the packet transmission. In the meantime, the sending trail might be aborted from `STOP` or `RETRANSMIT` requests (lines 5 and 9 of Figure 5), making the packet buffer to go out of scope, and leading to a *dangling pointer* in the radio driver. Céu refuses to compile programs like this and requests for *finalization* clauses to accompany unsafe low-level calls [19]. The code on the right of Figure 6 properly dequeues the packet if the block of `buffer` goes out of scope.

Finalization clauses are fundamental to preserve the orthogonality of `par/or` compositions in SRP.

2.3 Organisms

Céu provides organisms as an abstraction mechanism that reconciles data and control state into a single concept. They provide an object-like interface (data state) as well as multiple lines of execution (control state). A class of organisms is composed of an interface and a single execution body. The interface exposes public variables, methods, and also internal events. The body can contain any valid code in Céu (including parallel compositions) and starts on instantiation, executing in parallel with the program. Organism instantiation can be either static or dynamic.

We propose a memory model for organisms that eliminates known issues in allocation: *memory leaks*, *dangling pointers*, and the need for *garbage collection*. The model is similar to stack-living local variables of procedures, providing lexical scope and automatic bookkeeping of organisms. We also restrict explicit references to organisms to avoid indirect manipulation.

The example of Figure 7 introduces static organisms through three code chunks:

- The leftmost code (*CODE-1*) is a modified version of the two blinking LEDs of Figure **??** that terminates after 1 minute.
- The code in the middle (*CODE-2*) abstracts the blinking LEDs in an organism class and uses two instances to reproduce the same behavior.
- The rightmost code (*CODE-3*) is the equivalent expansion without organisms, which should resemble the original leftmost code.

In *CODE-2*, the Blink class (lines 1-9) exposes the `port` and `dt` fields, which correspond to the LED port and blinking period to be configured for each instance. The application creates two instances, specifying the fields in the constructors (lines 12-15 and 17-20). A constructor starts the instance body to execute in parallel with the application. When reaching the `await 1min` (line 22), each instance already started its body (lines 5-8).

CODE-3 is semantically equivalent to the one in the middle, with the organisms constructors and bodies expanded (lines 11-16 and 20-25) in a `par/or` with the rest of the application (`await 1min`, in line 28). Note the `await FOREVER` statements (lines 17 and 26) that avoid the organisms bodies to terminate the `par/or`. The `_Blink` type (lines 1-4) corresponds to a simple datatype without execution body.

The main characteristic of organisms is that, unlike objects, they react to the environment by themselves, i.e., they become alive once instantiated (hence the name *organisms*). For instance, they need not be included in an global observer list, or rely on the main program to feed their methods with input from the environment. Even tough the organisms run independent from the main program, they are still subject to the disciplined synchronous model, which keeps the whole system deterministic.

Another positive aspect of organisms regards to lexical scope and automatic memory management. *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. Note that the equivalent expansion of *CODE-3* relies on a `par/or` (lines 9-29) that properly aborts all organisms bodies after that period (line 28), but before they go out of scope (line 30). Furthermore, the `par/or` termination trigger all active finalization clauses inside the organisms.

Organisms can be explicitly manipulated through their interface variables, methods, and internal events. We illustrate events here, given that variables and methods work in the same way as in object oriented programming. Figure 8 defines a class `Unit` (lines 1-16) that exposes the event `move` (line 2) which requests the unit to move to a position. The main program (lines 18-21) creates two units and requests them to move to different positions in a interval of 1 second. The body of the class initializes the current unit's position `pos` and destination position `dst` (lines 4-5). Then, the body enters in a continuous loop (lines 6-15) to handle `move` requests (line 8) while performing the actual moving operation (lines 10-13) in parallel. The `par/or` restarts the loop on every `move` request, which updates the `dst` position. The moving operation 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. An advantage of event handling over method calls is that they can be composed in the organism body and affect other ongoing operations. In the example, the `await move` aborts and restarts the moving operation, just like the timeout pattern of Figure 4.

2.3.1 Dynamic organisms

TODO

```
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 port;
    var int dt;
do
    loop do
        await (dt)ms;
        _toggle(port);
    end
end

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

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

    await 1min;
end
```

/* CODE—2: blinking organisms */

```
struct _Blink with
    var int port;
    var int dt;
end;

do
    var _Blink b1, b2;

    par/or do
        // body of b1
        b1.port = 0;
        b1.dt   = 2;
        loop do
            await (b1.dt)ms;
            _toggle(b1.port);
        end
        await FOREVER;
    with
        // body of b2
        b2.port = 1;
        b2.dt   = 4;
        loop do
            await (b2.dt)ms;
            _toggle(b2.port);
        end
        await FOREVER;
    with
        await 1min;
    end
end
```

/* CODE—3: organisms expansion */

Figure 7: Two blinking LEDs using organisms.

```
class Unit with
    event int move;
do
    var int pos = 0;
    var int dst = 0;
    loop do
        par/or do
            dst = await this.move;
        with
            if dst != pos then
                <code—to—move—pos—to—dst>
            end
            await FOREVER;
        end
    end
end

var Unit u1, u2;
emit u1.move => 100;
await 1s;
emit u2.move => 200;
```

Figure 8: Organism manipulation through interface events.

2.3.2 References

TODO

3. RELATED WORK

TODO

4. CONCLUSION

TODO

5. REFERENCES

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

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

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

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

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

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

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

[8] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

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

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

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

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

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

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

[15] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

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

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

[18] G. Salvaneschi, G. Hintz, and M. Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the of the 13th international conference on Modularity*, pages 25–36. ACM, 2014.

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

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