Structured Reactive Programming with Céu

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 [16, 4]. 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 [5, 21]: The imperative style of Esterel [8] organizes programs with structured control flow primitives, such as sequences, repetitions, and parallelism. The dataflow style of Lustre [15] represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming [25] modernized the dataflow style and became mainstream, deriving a number of languages and libraries, such as Flapjax [19], Rx (from Microsoft), React (from Facebook), and Elm [9]. In contrast, the imperative style of Esterel 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 [18, 22]. 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`" [11, 10, 13].

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 [12]) to support continuous interaction with the environment. In practical terms, SRP provides two extensions to SP: an "`await <event>`" statement that suspends a line of execution until the referred event occurs, keeping all context alive; and parallel constructs that compose multiple lines of execution and make them concurrent. The `await` statement represents the imperative and reactive nature of SRP, recovering sequential execution lost with the observer pattern. Parallel compositions[1] allow for multiple `await` statements to coexist, which is necessary to handle concurrent events, common in reactive applications.

We advocate SRP through the Esterel-based language Céu [23], a contemporary outlook of imperative reactivity that aims to expand it from the rigid embedded domain. We believe that the excessively rigorous semantics of Esterel, whose focus is 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 difficult to deal with virtual resources such as graphical widgets, game units, and network sessions.

[1]We use the term *parallel composition* as a synonym for *side-by-side lexical composition*, which does not imply many-core parallel execution.

Our main contribution is a new abstraction mechanism for Céu, the *organisms*, that encapsulate parallel compositions with an object-like interface. In brief, organisms are to SRP like procedures are to SP, i.e., one can abstract a portion of code with a name and manipulate (call) that name from multiple places. There are, however, additional challenges that apply to organisms:

- Organisms are themselves alive, 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 4 discusses related work. Section 5 concludes the paper and makes final remarks.

2. SRP WITH CÉU

Céu is a concurrent language, in which its lines of execution, known as *trails*, react continuously to external stimuli, all together in synchronous steps. Figure 1 shows a compact reference of Céu.

```
// 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)

// 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

// C INTEGRATION
_f();                    // C call (prefix '_')
finalize <...>           // finalization
    with <...> end

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

Figure 1: Syntax of Céu.

The introductory example of Figure 2 starts two trails with the `par` construct: the first (lines 4-8) increments variable `v` on every second and prints its value on screen; the second (lines 10-13) resets `v` on every external request to `RESET`. Programs in Céu can access *C* libraries of the underlying platform by prefixing symbols with an underscore (e.g., `_printf(<...>)`).

```
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.

2.1 Synchronous concurrency

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[2]. If multiple trails react to the same event, the scheduler employs lexical order to preserve determinism, i.e., the trail that appears first in the source code executes first. To avoid infinite execution for reactions, Céu ensures that all loops have `await` statements on their bodies [23].

As a result of synchronous execution, all operations to variable `v` in Figure 2 are 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.

The synchronous model also empowers SRP with an *orthogonal abortion* construct that simplifies the composition of activities[3]. Orthogonal abortion is the ability to abort an activity from outside it, without affecting the overall consistency of the system (e.g., properly releasing global resources). The example that follows shows the `par/or` construct of Céu to compose trails and rejoin when either of them terminates, properly aborting the other:

```
par/or do
    // trail A
    <local-variables-A>
    <body-A>
with
    // trail B
    <local-variables-B>
    <body-B>
end
<local-variables-rejoin>
<body-rejoin>
```

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

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

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

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). The language runtime needs to wait for the activity to be consistent before aborting it, but what results in two unsatisfactory semantics for a hypothetical `par/or`: either wait to rejoin, making the program unresponsive to incoming events for an arbitrary time; or rejoin immediately and let the activity complete termination in the background, which may cause race conditions with the subsequent code. Immediate rejoin also leads to heap-based allocation (which is discouraged in the context of embedded systems) because activities may coexist with the subsequent code for some time.

As matter of fact, asynchronous languages do not provide effective abortion: *Java*'s `Thread.stop` primitive has been officially deprecated [20]; *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"* [17]; Instead, asynchronous activities typically agree a on common protocol to abort each other (e.g., through shared state variables or message passing), increasing coupling among them with concerns that are not directly related to the problem specification.

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 [6]. In Section 2.2 we show how CÉU deals with abortion of trails that use stateful resources from the environment, such as file handling and network transmissions.

2.2 Parallel compositions

In terms of control structures, SRP extends SP with 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 that follows compares the `par/and` and `par/or` compositions side by side:

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

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

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

The code <...> represents a complex behavior with any degree of nested compositions. In the `par/and` variation, the behavior repeats every second at minimum because both sides must terminate before re-executing the loop. In the `par/or` variation, if the behavior 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 3 relies on hierarchical `par/or` and `par/and` compositions to describe the state machine of a protocol for sensor networks ported to CÉU [14, 23]. The input events `START`, `STOP`, and `RETRANSMIT` (line 1) represent the external interface of the protocol with a client application. The protocol enters the top-level loop and awaits the starting event (line 3). Once the client application makes 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 (lines 11-16). The transmission (collapsed in line 15) is enclosed with a `par/and` that takes at least one minute before looping to avoid flooding the network. At any time, the client may request a retransmission (line 9), which terminates the `par/or` (line 8), aborts the ongoing transmission (line 15, if not idle), and restarts the loop (line 7). Also, the client may request to stop the whole protocol (line 5), which terminates the outermost `par/or` and aborts the transmission and all composed trails. 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 as specified.

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 [23].

2.2.1 Finalization

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

/* CODE—1: original blinking */
```

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

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

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

    await 1min;
end

/* CODE—2: blinking organisms */
```

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

do
    var _Blink b1, b2;

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

/* CODE—3: organisms expansion */
```

Figure 5: Two blinking LEDs using organisms.

```
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 4: Finalization clauses safely release low-level resources.

The CÉU compiler tracks the interaction of `par/or` compositions with automatic variables and stateful native *C* functions (e.g., device drivers) to ensure proper trail abortion.

Consider the code on the left of Figure 4, which expands the sending trail of Figure 3 (line 15). The `buffer` packet is a local variable whose pointer is passed to native function `_send_enqueue`. The call enqueues the pointer in the radio driver, which holds it up to the occurrence of `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 3), 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 requires *finalization* clauses to accompany unsafe native calls [23]. The code on the right of Figure 4 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.

3. ORGANISMS: SRP ABSTRACTIONS

Applications often need to replicate a behavior and rely on abstractions that the language provide. In SP, the typical abstraction mechanism are procedures, which abstract recurrent routines with meaningful names that can be invoked multiple times with different parameters. However, procedures have a single entry point, and were not devised for continuous and concurrent input. For instance, procedures cannot express parallel compositions as described in Section 2.2. The prevailing imperative abstraction mechanism for reactive applications are objects used in conjunction with the observer pattern. Even though objects abstract data with proper encapsulation, they cannot encapsulate control flow because methods have to return control as quickly as possible to keep the application responsive [?].

CÉU abstracts data and control state into the single concept of organisms. A class of organisms describes 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. On an organism is instantiated, its bode starts to execute in parallel with the program. Organism instantiation can be either static or dynamic.

The example of Figure 5 introduces static organisms through three code chunks:

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

pansion without organisms, which should resemble 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 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 also 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 organisms constructors and bodies expanded (lines 11-16 and 20-25). The generated `par/or` makes the instances and the rest of the application (i.e., `await 1min`, in line 28) concurrent. Note the `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. Note that actual implementation of CÉU does not expand the organisms bodies like in *CODE-3* (the implementation is not discussed in this paper). In fact, 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 they can react independently and directly to the environment, i.e., once instantiated they become alive and reactive (hence the name *organisms*). 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. 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.

The memory model for organisms 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.

Regarding lexical scope and automatic memory 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. In the equivalent expansion of *CODE-3*, the `par/or` (lines 9-29) aborts the organisms bodies after that period (line 28), just before they go out of scope (line 30). In addition, the `par/or` termination properly triggers all active finalization clauses inside the organisms. Lexical scope extends the idea of orthogonal abortion to organisms, as they are automatically aborted when going out of scope.

In addition to properties and methods, organisms also expose internal events which support `await` and `emit` operations. In the example of Figure 6 the class `Unit` (lines 1-16) defines the position property `pos` with default vaule `0` (line 2), and the event `move` which exposes requests to move the

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

Figure 6: Organism manipulation through interface events.

unit position (line 3). The main program (lines 18-28) creates two units and requests them to move to position `500` with a interval of 1 second. The body of the class initializes the current unit's destination position `dst` to the current position (line 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 to affect other ongoing operations. In the example, the `await move` (line 8) aborts and restarts the moving operation, just like the timeout pattern of Section 2.2.

3.1 Dynamic organisms

Static embedded systems typically handle real sensors and actuators hardware through a one-to-one correspondence with software, i.e., a static (possibly complex) piece of software deals with a piece of hardware. In contrast, more general reactive systems have to deal with resource virtualization, such as multiplexing protocols in a network, or simulating entire civilizations in a game. Dynamic allocation for organisms extends the power of SRP to handle virtual resources in reactive applications.

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

```
every 1s do
    var Unit* u = spawn Unit with
                      this.pos = _rand() % 500;
                  end;
    emit u:move => _rand() % 500;
end
```

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

Figure 7: Spawns and moves a new `Unit` every second.

The `spawn` allocates proper memory and returns a pointer to the new organism, which can be dereferenced through the operator '`:`' (analogous to '`->`' of *C/C++*).

Dynamic instances also execute in parallel with the rest of the application, but have different lifetime and scoping rules. On the one hand, a static instance has a unique identifier and a well-defined scope that holds its memory resources. On the other hand, a dynamic instance outlives the scope that spawns it and is anonymous (but may have pointers to it, as in the previous example).

To recognise the reactive and independent nature of organisms, Céu supports that they control their own lifetime: once the body of a dynamic organism terminates, it is automatically reclaimed from memory. The example that follows redefines the body of the `Unit` class of Figure 6 to terminate after 1 hour, imposing a maximum life span in which a unit can react to `move` requests:

```
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, as there is no way to abort the execution of dynamic instances. To overcome this limitation, Céu provides scoped *pools* as containers to hold dynamic organisms instances. The example of Figure 7 declares the `units` pool that holds a maximum of 10 instances (line 3). A new unit is spawned in this pool on every second (note the `in units`, in line 6). Once the application receives a `CLICK` (line 11), the `par/or` terminates, making the `units` pool to go out of scope and abort all units alive.

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

3.2 Pointers and References

As organisms react independently from the environment, it is often not necessary to hold pointers to organisms. In fact, pointers can be dangerous because they may last longer than the organisms they refer:

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

The program first assigns a `Unit` reference to pointer `u`. Then, it dereferences the pointer in two occasions: in the same reaction, just after acquiring the reference; in another reaction, after *2h*, when the pointed organism may have already terminated and freed.

Céu enforces all pointer accesses across reactions to use the `watching` construct which supervises organism termination:

```
var Unit* u = <...>;
u:pos = 0;
watching u do
    await 2h;
    emit u:move => 100;
end
```

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

The `watching` construct terminates when the referred organism terminates, eliminating possible dangling pointers in the program. The code in the right shows the equivalent expansion of the `wacthing` construct to a `par/or` that awaits the special event `__killed`.

4. RELATED WORK

Esterel provides module instantiation through the `run` command, which syntactically replaces the command by the module body [7]. Similarly, ESPranto [24] extends Esterel with a macro-based abstraction mechanism that enables the creation of domain specific languages. However, macros expand at compile-time, being impossible to describe dynamic instantiation. Furthermore, each use of a macro results in code replication, having a negative impact on code size.

Simula simulation applies to reactivity internal vs external - corroutines and objects

two fundamental limitations - no composition - no scopes - all on the heap - interface however

In some sense Esterel + Simula

5. CONCLUSION

TODO

6. 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 *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 *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993.

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

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

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

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

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

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

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

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

[24] R. van Herk and L. Holenderski. Espranto: a framework for developing applications for adaptive hardware. In *Proceedings of ADAPTATIVE'2010*, pages 184–193, Lisbon, Portugal, 2010.

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