Structured Reactive Programming with Abstractions

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 [5, 23]: 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 [26] modernized the dataflow style and became mainstream, deriving a number of languages and libraries, such as Flapjax [21], Rx (from Microsoft), React (from Facebook), and Elm [9]. 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 [20, 24]. However, short-lived callbacks (i.e., the observers) eliminate any vestige of structured programming, such as support for loops and automatic variables [3], which are elementary capabilities of imperative languages. In this sense, the observer pattern actually disrupts imperative reactivity, becoming "our generation's `goto`" [11, 10, 13].

In this work, we revive the synchronous imperative programming style of Esterel, which we now refer as *Structured Reactive Programming (SRP)*. SRP extends classical structured programming (SP) [12] to support continuous interaction with the environment through hierarchical control constructs: *concatenation*, *selection*, *repetition*, and also *parallelism*. Concretely, we consider that SRP must provide at least two fundamental extensions to SP:

- An `await <evt>` statement that suspends a line of execution until the referred event occurs, keeping the whole data and control context alive.
- Parallel constructs that compose multiple lines of execution and make them concurrent.

The `await` statement captures the imperative and reactive nature of SRP, recovering from the inversion of control inherent to the observer pattern, and thus 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.

Our contribution is a new abstraction mechanism that aims to expand the rigid embedded domain of SRP with better support for code reuse and a tractable memory model. The *organisms*, which we propose, abstract the 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:

- Organisms are themselves alive and concurrent, being essentially subprograms with enduring 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 also applies to embedded systems.

We provide organisms in the Esterel-based programming language Céu [25].

The rest of the paper is organized as follows: Section 2 reviews the synchronous and asynchronous execution model, justifying the former as a better choice for SRP. Section 3 presents SRP through Céu, with its basic control mechanisms and the organisms abstraction. Section 4 discusses related work. Section 5 concludes the paper and makes final remarks.

2. THE SYNCHRONOUS CONCURRENCY MODEL

"Reactive systems" are not a new class of software and have been first described by Harel as being "repeatedly prompted by the outside world and their role is to continuously respond to external inputs" [17]. In comparison to traditional "transformational systems", he recognises reactive systems as "particularly problematic when it comes to finding satisfactory methods for behavioral description". Berry goes further and makes a subtle distinction between "interactive" and "reactive" systems [4]:

- Interactive programs interact at their own speed with users or with other programs; from a user point of view a time-sharing system is interactive.
- Reactive programs also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself.

This distinction is fundamental because the different control perspectives (i.e., *at the speed of the program* vs *at the speed of the environment*) implies the use of different underlying concurrency models. Overall, *synchronous languages* deal with reactive systems better, while *asynchronous languages*, with interactive systems [7]. Both mentioned authors propose synchronous languages for designing reactive systems (Statecharts [16] and Esterel [8]).

The synchronous execution model is based on the hypothesis that internal computations (*reactions*, in this context) run infinitely faster than the rate of events that trigger them. In other words, the input and corresponded output are simultaneous, because reactions takes no time.

Figure 1 shows two common implementation schemes for synchronous schedulers [5]. In the event-driven scheme, a loop iteration computes outputs for each event occurrence. In the sampling scheme, a loop iteration computes the inputs

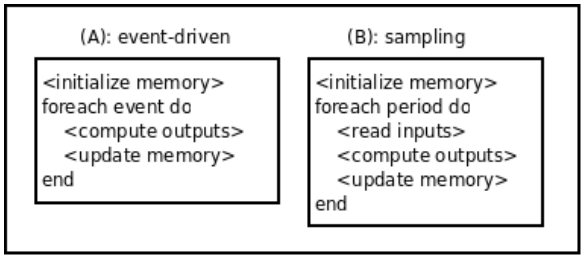

Figure 1: Schedulers for synchronous systems

and outputs on every clock tick. In both cases, each loop iteration represents a logical instant in which the system as a whole reacts synchronously before going to the next instant. During a reaction, the environment is invariant and does not affect the running iteration[1]. Both schemes are compliant with the synchronous hypothesis, in which input and resulting output happen at the same time, considering this notion of time as a sequence of discrete events or clock ticks.

The asynchronous execution model is more general and does not make assumptions about implicit synchronization. Each activity in the system[2] is independent from one another and executes at its own pace. For instance, an activity can perform time-consuming operations (e.g., compression, cryptography) without affecting the overall progress of the system. This separation also makes many-core parallelism straightforward. However, in order to coordinate at specific points, the activities require explicit synchronization primitives (e.g., mutual exclusion or message passing).

In this work, we emphasize two desired features in concurrent systems that the synchronous model makes possible: *deterministic execution* and *orthogonal abortion*. In the context of reactive applications, we interpret determinism as *reproducible execution given the same sequence of stimuli*, i.e., the outcome depends exclusively of an external input timeline, in contrast with internal scheduling and communication timings. 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).

Figure 2 shows three implementations for an application that blinks two LEDs in parallel with different frequencies. We use two asynchronous languages (a CSP-based [19] and a thread-based [1] language), and also the synchronous language Céu. The intent and syntactic structure of the implementations are similar: composing the two blinking activities in parallel. The LEDs should blink together every 3 seconds (the least common denominator between 600ms and 1s). As we expected, the LEDs in the two asynchronous implementations loose synchronism after some time of execution, while the implementation in Céu remains synchronized forever. The example highlights how the inherent non-determinism in the asynchronous model makes hard to (blindly) compose activities supposedly synchronized: un-

[1]An actual implementation enqueues incoming input events to process them in the next iterations.
[2]We use the term activity to generically refer to a language's unit of execution (e.g., *thread*, *actor*, *process*, etc.).

```
// OCCAM-PI
PROC main ()
 CHAN SIGNAL s1,s2:
 PAR
  PAR
   tick(600, s1!)
   toggle(11, s1?)
  PAR
   tick(1000, s2!)
   toggle(12, s2?)
:
```

```
// ChibiOS
void thread1 () {
  while (1) {
    sleep(600);
    toggle(11);
  }
}
void thread2 () {
  while (1) {
    sleep(1000);
    toggle(12);
  }
}
void setup () {
  create(thread1);
  create(thread2);
}
```

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

Figure 2: Two blinking LEDs in OCCAM-PI, ChibiOS and Céu.
The lines of execution in parallel blink two LEDs (connected to ports 11 and 12) with different frequencies. Every 3 seconds the LEDs should light on together.

```
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.

predictable scheduling as wall as latency in message-passing eventually cause observable asynchronism. In Céu, the `await` is the only primitive that takes time, but which the programmer uses explicitly to conform with the problem specification. The internal timings for communication and computation, which the programmer cannot control, are neglected in accordance to the synchronous hypothesis. The language runtime compensates them in the subsequent reaction in order to conform with the model and remain synchronized [25]. Arguably, reasoning over `await` statements is simpler than also having to consider all statements of the language.

Consider now the problem of aborting an *activity A* as soon as an *activity B* terminates, and vice versa. 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 an orthogonal abortion construct, 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.

Orthogonal abortion in asynchronous languages is challenging [6]. For instance, when an activity terminates, the other activity to be aborted might be on a inconsistent state (e.g., suspended but holding a lock, or actually executing in another core). In order to properly abort an activity, the language runtime has two possible semantics for the `par/or`: either wait to rejoin (delayed termination); or rejoin immediately and wait in the background (immediate termination). Both options have problems:

- **delayed**: The program becomes unresponsive in the meantime.
- **immediate**: The programmer may assume that both activities have terminated. Also, 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" [18]; Java's `Thread.stop` primitive has been officially deprecated [22]; and pthread's `pthread_cancel` does not guarantee immediate cancellation [2]. 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 [6]. Céu provides the presented `par/or` composition, which is equivalent to Esterel's `trap` orthogonal abortion construct. We show in Section 3 how abortion integrates safely with activities that use stateful resources from the environment, such as file handling and network transmissions.

Even tough the deterministic and abortion examples can be properly implemented in asynchronous languages, they require to tweak the activities with mutual synchronization primitives. This increases the coupling degree among activities with concerns that are not directly related to the problem specification.

The two models suggest a tradeoff between unrestricted execution with real parallelism versus structural composability with deterministic behavior. For reactive applications with continuous and real-time concurrency, we believe that the synchronous execution model is more appropriate.

3. STRUCTURED REACTIVE PROGRAMMING WITH CÉU

TODO

3.1 Parallel Compositions and Finalization

In terms of control structures, the basic extensions of SRP are the parallel compositions, allowing applications to handle multiple events concurrently. Céu has 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 5 that follows 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

```
// 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
every <id> 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 4: Syntax of Céu.

```
// 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 5: The *sampling* and *timeout* patterns using parallel compositions.

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 archetypes represent, respectively, the *sampling* and *timeout* patterns, which are typical of reactive applications.

The code in Figure 6 is part of a collection protocol for sensor networks ported to Céu [14, 25] and relies on `par/or` and `par/and` compositions to describe its state machine. 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 7-18): 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

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

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 [25].

Programs in Céu can access *C* libraries available in the underlying platform by prefixing symbols with an underscore (e.g., `_printf(<...>)`). However, the interaction of stateful *C* functions with `par/or` compositions and automatic variables of Céu demands attention. Figure 7 shows, on the left, the code that sends the status packet of Figure 6 (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 6), 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 demands *finalization* clauses to accompany unsafe low-level calls [25]. The code in the right of Figure 7 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.

3.2 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.

```
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 7: Finalization clauses handle low-level resources from the platform.

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 8 introduces static organisms through three code chunks:

- The leftmost code (*CODE-1*) is a modified version of the two blinking LEDs of Figure 2 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 1-8) 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 11-14 and 16-19). A constructor starts the instance body to execute in parallel with the application. When reaching the `await 1min` (line 21), each instance already started its body (lines 5-7).

CODE-3 is semantically equivalent to the one in the middle, with the organisms constructors and bodies expanded (lines 11-15 and 19-23) in a `par/or` with the rest of the application (`await 1min`, in line 26). Note the `await FOREVER` statements (lines 16 and 24) 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.

```
1  class Unit with
2      event int move;
3  do
4      var int pos = 0;
5      var int dst = 0;
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, u2;
19 emit u1.move => 100;
20 await 1s;
21 emit u2.move => 200;
```

Figure 9: Organism manipulation through interface events.

Another positive aspect of organisms regards to lexical scope and automatic memory management. *CODE-2* uses a `do-end` block (lines 10-22) that limits the scope of the organisms for 1 minute (line 21). 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 9-27) that properly aborts all organisms bodies after that period (line 26), but before they go out of scope (line 28). 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 9 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 5.

3.2.1 Dynamic Organisms

TODO

3.2.2 References

TODO

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

/* CODE—1: original blinking */

```
class Blink with
    var int port;
    var int dt;
do
    every (dt)ms do
        _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;
        every (b1.dt)ms do
            _toggle(b1.port);
        end
        await FOREVER;
    with
        // body of b2
        b2.port = 1;
        b2.dt   = 4;
        every (b2.dt)ms do
            _toggle(b2.port);
        end
        await FOREVER;
    with
        await 1min;
    end
end
```

/* CODE—3: organisms expansion */

Figure 8: Two blinking LEDs using organisms.

4. RELATED WORK

TODO

5. CONCLUSION

TODO

6. REFERENCES

[1] ChibiOS Homepage. http://www.chibios.org/ (accessed in Jul-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, S. Ramesh, and R. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–98. ACM, 1993.

[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 Jul-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 Jul-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. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

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

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

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

[21] 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.

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

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

[24] 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.

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

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