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.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Concurrency, Dataflow, Determinism, Embedded Systems, Esterel, Synchronous, Reactivity

1. INTRODUCTION

Reactive applications interact in real time and continuously 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]: The imperative style of Esterel [8] organizes programs with structured control flow primitives, such as sequences, repetitions, and also parallelism. The dataflow style of Lustre [13] represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming [22] 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 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* of object oriented languages, because it heavily relies on side effects over shared data among objects [18, **?**]. 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 an elementary capability of imperative languages. In this sense, the observer pattern actually disrupts imperative reactivity, becoming "our generation's `goto`" [11, 10, 12].

In this work, we present a comprehensive set of imperative abstractions for developing structured reactive applications through the programming language Céu [21]. Céu is based on Esterel and relies on a similar synchronous and deterministic execution model that simplifies the reasoning about concurrency aspects. Céu provides a contemporary outlook of imperative reactivity that aims to expand its application domain. In comparison to standard structured programming, Céu provides three fundamental extensions:

- An `await <evt>` statement to suspend a line of execution until the referred event occurs, keeping the whole data and control context.
- Parallel constructs (`par`, `par/or`, and `par/and`) to compose multiple lines of execution.
- An abstraction mechanism, named as *organisms*, to reconcile data and control state in a single concept.

The `await` statement captures the imperative and reactive nature of the language, 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 fundamental to handle concurrent events common in reactive applications. An organism abstracts parallel and `await` statements and offer an object-like interface that other parts of the program can manipulate.

Our general contribution is a language design to develop reactive systems through structured control primitives. The

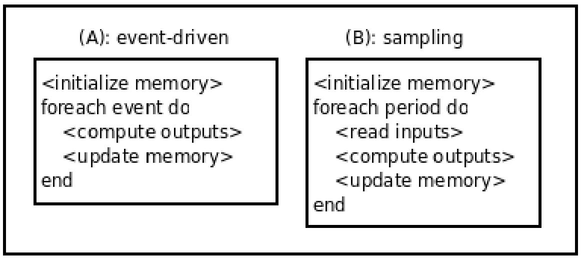

Figure 1: Schedulers for synchronous systems

language relies on a well-founded synchronous concurrency model that generates deterministic We introduce a new abstraction mechanism

TODO

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 structured reactive programming. Section 3 presents.

TODO

2. SYNCHRONOUS REACTIVE 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" [15]. 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 [14] 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 and outputs for each clock tick. In both cases, each loop iteration represents a logical instant in which the system as a whole reacts synchronously before going to 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 entity in the system (e.g., a thread or actor) is independent from one another and executes at its own speed. In order to coordinate at specific points, they require explicit synchronization primitives (e.g., mutual exclusion or message passing).

In this work, we emphasize two desired features that the synchronous model makes possible for concurrent systems and that strengthen structured reactive programming: *deterministic execution* and *orthogonal abortion*.

TODO

Figure 2 shows three implementations for an application that blinks two LEDs in parallel with different frequencies. We use two asynchronous languages (an actor-based [17] 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[2] 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: unpredictable 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 according 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 itself compensates them in the subsequent reaction in order to conform with the model and remain synchronized [21].

Consider now the problem of aborting an *activity A* as soon as an *activity B* terminates, and vice versa. Figure 3 shows a 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

[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: TODO.TODO

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 of activities 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 abort the activity on a consistent state, 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 but one is still terminating. 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" [16]; Java's `Thread.stop` primitive has been officially deprecated [20]; and pthread's `pthread_cancel` does not guarantee immediate cancellation [2]. Instead, activities must 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. Both abort activities immediately, which are always in a consistent state. 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 between activities with concerns that are not directly related to the problem specification.

TODO

3. STRUCTURED REACTIVE PROGRAMMING

- besides standard strucutred (loops, conditionals, sequences) - add compositions

- await, sequence, vars - parallel, patterns, ortoghonal - finalize for C integration and keep orthogonality - organisms, static, dynamic. do T;

same implementation for an activity w/o concurrency previously written no changes

4. RELATED WORK

- asynchronous langs - Esterel + descendants - CRP - simula - FRP

interactive vs reactive asynchronous vs synchronous dataflow vs control dynamic vs static

imperative - sequential (eliminates state machines) - better resource control - less abstract

dataflow

exemplo data melhor vs control melhor

The synchronous concurrency model...

We show composibility, sequential, imperative safety Then, we extend synchronous reactive programming with dynamic

control vs data reactivity

Adding dynamic state and references to SRP is a significant contribution. However the presentation assumes the audience already appreciates the advantages of SRP over standard imperative semantics. Most observers will not notice the difference at all and think this is just a weird syntactic sugar for standard multithreaded programming. SRP needs to be better explained and justified, perhaps by showing how it simplifies the examples.

Céu is a Esterel-based reactive language that targets constrained embedded platforms. Relying on a deterministic semantics, it provides safe shared-memory concurrency among lines of execution. Céu introduces a stack-based execution policy for internal events which enables advanced control mechanisms considering the context of embedded systems, such as exception handling and a limited form of coroutines. The conjunction of shared-memory concurrency with internal events allows programs to express dependency among variables reliably, reconciling the control and dataflow reactive styles in a single language.

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

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

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

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

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

[17] 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.

[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 Jul-2014), 2011.

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

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