- rely on the synchronous execution model - compositions
- than abstracting over it - finalization, safety - ortoghonal preemption - simple reasoning
- OS/drivers are stateful

Structured Reactive Programming

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

ABSTRACT

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

1. SYNCHRONOUS REACTIVE MODEL

Reactive applications interact in real time and continuously with external stimuli from the surrounding 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 [1]: The imperative style of Esterel [?] organizes programs with control flow primitives, such as sequences, repetitions, and also parallelism. The dataflow style of Lustre [?] represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming [?] modernized the dataflow style and became mainstream, deriving a number of languages and libraries, such as Flapjax [?], Rx (from Microsoft), React (from Facebook), and Elm [?]. 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 [?, ?]. However, short-lived callbacks (the observers) eliminate any vestige of structured programming, such as loops and automatic variables [?], which are an elementary capability of imperative languages. In this sense, the observer pattern actually disrupts imperative reactivity, becoming "our generation's `goto`" [?, ?, ?].

In this work, we present a comprehensive set of imperative abstractions for developing structured reactive applications through the programming language Céu [7]. 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 the application domain of this family with a new abstraction mechanism. 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.
- 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 allows for multiple `await` statements to coexist, which is fundamental to handle events concurrently (which reactive applications often require). An organism abstracts parallel and `await` statements and offer an object-like interface that other parts of the program can manipulate.

Permeating all aspects of the programming style, the synchronous execution model restricts but makes possible an underlying model that makes it all reasonable

CONTRIBUTIONS:

determinism

distribution

parallelism

The major contribution of this work

- dynamic synchronous - getting rid of . memory leaks, danging pointers, and garbage collection

sintaxe (separar section 1 / 2)

DATA+CONTROL and can be dynamically created.

Given concurrency the synchronous model

await inside a sequence or loop form to have multiple sequence allowing for handling multiple events at the same time

Céu has its roots on Esterel and relies on a similar synchronous and deterministic execution model that simplifies the reasoning about concurrency aspects.

these languages shows that imperative can be safe with a reasonable concurrency model and static analysis

SYNCHRONOUS DYNAMIC

sequentiality composibility observer pattern

The rest of the paper is organized as follows: Section 2 reviews the synchronous execution model, which is suitable for reactive programming, and is the base of Céu. Section 3

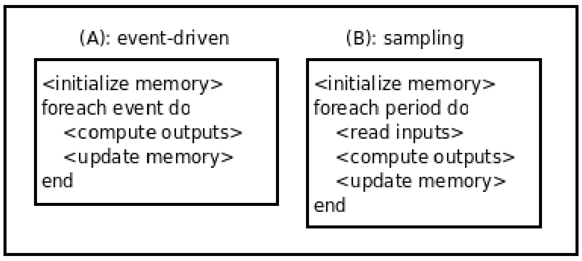
2. SYNCHRONOUS REACTIVE MODEL

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

- 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 characteristics (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 [?]. Both mentioned authors propose synchronous languages for designing reactive systems (Statecharts [4] and Esterel [3]).

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 in the synchronous model, because reactions takes no time.



```
<initialize memory>
foreach event do
   <compute outputs>
   <update memory>
end
```

```
<initialize memory>
foreach period do
   <read inputs>
   <compute outputs>
   <update memory>
end
```

Figure 1: Schedulers for synchronous systems

Figure **??** shows two common implementation schemes for synchronous schedulers [1]. 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, possibly buffering incoming input events to be processed further. Both schemes are compliant with the synchronous hypothesis, in which input and resulting output happen at the same time (in 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 of 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).

The synchronous model can handle two desired features considering concurrency: *deterministic execution* and *orthogonal abortion*. These features are the reason which we explore in the sections that follow. For these For structured reactive programming, we argue that the synchronous model is more appropriate.

The way each side is implemented does not affect the behavior, only the way you compose them (which is external to each implementation)

DET and ABRT both yield to better reasoning behave the same with or without concurrency

Figure **??** shows three implementations for an application that blinks two LEDs in parallel with different frequencies. We use two asynchronous languages (actor-based [6] and thread-based [?], and also the synchronous language Céu. The intent and syntactic structure of the implementations are similar: composing the two blinking activities[1] in parallel. The LEDs should blink together every 3 seconds (i.e., it is 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 Céu implementation remains synchronized forever. The example highlights how the inherent non-determinism in the asynchronous model makes hard to (blindly) compose

[1]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

activities supposedly synchronized: unpredictable scheduling as wall as latency in message passing causes asynchronism. In Céu, `await` is the only primitive that takes time, which the programmer uses explicitly to conform with the problem specification. However, the internal timings for communication and computation, which the programmer cannot control, are neglected in accordance to the synchronous hypothesis.

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. 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` terminates, a new set of local variables goes alive, supposedly reusing the space from the locals going out of scope.

Abortion of activities in asynchronous languages is challenging [2]. 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). Assuming that the language runtime awaits the activity to be consistent before aborting it, the semantics for a `par/or` can either wait to rejoin (synchronous termination), or rejoin immediately and wait in the background (asynchronous termination). Both options have problems:

- **synchronous**: The program becomes unresponsive in the meantime.
- **asynchronous**: 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" [5]; Java's `Thread.stop` primitive has been officially deprecated [?]; and pthread's `pthread_cancel` API does not guarantee immediate cancellation [?]. Activities must agree a 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 [2], because in between two reactions, the whole system is idle and consistent. Céu provides the `par/or` construct with the synchronous termination semantics (described above). We show in Section 3 how to integrate abortion with activities that use stateful resources from the environment, such as files and network transmissions.

, which is regarded as an orthogonal preemp- tion primitive [4], because composition <P> does not know when and why it gets aborted by event E.

Asynchronous for- malisms assume time independence among processes and require explicit synchronization mechanisms. Therefore, to achieve the desired specification, processes <P> and <Q> must be tweaked with synchronization commands, breaking the orthogonality assumption. [4]

deterministic execution and abortion are possible but require tweaking each other changing the implementation with synchronization concerns that the language could solve the non-concurrent version is different from the concurrent version

because a reactive core only deal with input/output

For synchronous languages, this is possible. In Céu we have exactly this primitive

all to the heap

When to terminate the other / start the continuation When to deallocate / reallocate memory Again, latency (locks, middle of transmission, middle of assignment, which granularity) which granularity? what if inside a lock?

Even though these examples can be properly implemented in asynchronous languages, they require tweaking the actor/threads with synchronization primitives. I.e., the non-concurrent version is different and requires coupling between them. Increase reasoning

On the one hand, xxx determinism reasoning On the other hand, the synchronous hypothesis does not hold for reactions that have latency (e.g., network communication or algorithm-intensive computations),

On the other hand, execution depends on the order reactions because reaction to events do not interleave in complex

parallelism latency

For reactive systems zzz For highly synchronous systems, the sole synchronization overhead, which is non-existent in XXX, may neutralize any gains with parallelism.

Illustrate with two examples: that explore the advantages of synchronous: reasoning in concurrency seamless composition abortion takes *at most* one tick alternatives strong/weak, but *at most* one tick async may take time (unresponsive in the meantime!) even if you simulate with communication, it will take time

CSP has only syntactic support for the *and* semantics p-threads discourages exit Java threads deprecated Exit

.: determinism and

1-determinism/synchronism 2-composition

communitacions is directed and takes time (because the receiver may not be waiting)

The synchronous model input => output broadcast possible global consensus possible determinism simpler model

In synchronous systems, communication is instantaneous. The zero-delay property of the synchronous hypothesis guarantees that no time elapses between event announcement and receiving. Also, as communication is via broadcast, all systems parts share the same information all the time. These two characteristics make global data consensus another property of synchronous systems.

hypothesis fail on however, does not apply when the computation involves latency: problem communication takes time either communication or time consuming operations because now, at the speed of the program

Termination and abortion: the real problem with the asynchronous model implies heap-based objects

Two examples: termination / memory / no composition time elapse / non-determinism / analogia do elevador

conclusion: no blind/free/orthogonal composition no deterministic/reproducible execution parallelism / no-forced synch time-consuming / independent

Both seminal works propose synchronous languages statecharts and Esterel so we call the synchrnous reactive model

to contrast with interactive or the new asynchronous reactive model

A recent

important distinction from interactive is only the "environment speed" a huge difference as it implies...

This definition is not the same as today:

The meaning of Reactive

synchronous reactive vs asynchronous reactive

berry classification bash asynchronous compositions PAR/OR not really parallel

diagrama com as duas formas de implementacao mostrar que em ambas, apenas um evento eh tratado "of course that reactions can trigger time-consuming operation that last multiple reactions, but it is important to recognize that the programmer is consious about that..."

Berry classified reactive applications as... From this, he denied threads (ada, now threads) and also CSP-like languages where communication is point-wise and takes time (csp, now erlang) asynchronous execution or asynchronous communication (with latency)

in terms of communication - broadcast (single global vision of an event) - p2p (loose simultaneity when simulating broadcast

3. STRUCTURED REACTIVE PROGRAMMING

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

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.

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

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

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

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

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

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

[6] 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.

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