Pontifícia Universidade Católica
do Rio de Janeiro

Francisco Figueiredo Goytacaz Sant'Anna

A Synchronous Reactive Language based on Implicit Invocation

MsC Dissertation

Dissertation presented to the Graduate Program in Computer Science of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Master in Computer Science

Adviser: Prof. Roberto Ierusalimschy

Rio de Janeiro
March 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Francisco Figueiredo Goytacaz Sant'Anna

A Synchronous Reactive Language based on Implicit Invocation

Dissertation presented to the Graduate Program in Computer Science of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Master in Computer Science. Approved by the following commission:

Prof. Roberto Ierusalimschy
Adviser
Departamento de Informática — PUC–Rio

Prof. Edward Hermann Haeusler
Departamento de Informática — PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico — PUC–Rio

Rio de Janeiro — March 16, 2009

All rights reserved. Copying portions or the entirety of the work is prohibited, except as otherwise permitted by the university, the author and the supervisors.

Francisco Figueiredo Goytacaz Sant'Anna

Researcher at the Telemidia Laboratory in Digital TV. Received a Computer Engineering degree at PUC-Rio in 2003.

Bibliographic data

Sant'Anna, Francisco

A Synchronous Reactive Language based on Implicit Invocation / Francisco Figueiredo Goytacaz Sant'Anna; adviser: Roberto Ierusalimschy. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2009.

v., 65 f: il. ; 29,7 cm

1. MsC Dissertation - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Bibliography included.

1. Informática – Dissertation. 2. Programming Languages. 3. Reactive Programming. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Title.

CDD: 004

Acknowledgments

To my adviser Professor Roberto Ierusalimschy for his knowledge and true exercise of advising.

To Professor Luiz Fernando Soares and colleagues of the Telemidia Lab for providing a favorable research environment.

To the professors of CAp-UERJ, responsible for my education before college.

To FINEP/FUNTTEL/MCT for funding my studies.

Resumo

Sant'Anna, Francisco; Ierusalimschy, Roberto. **Uma Linguagem Síncrona Reativa baseada em Invocação Implícita**. Rio de Janeiro, 2009. 65p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O paradigma de programação reativa cobre uma vasta gama de aplicações, tais como jogos e sistemas multimídia. As linguagens populares vigentes negligenciam a programação reativa, não possuindo primitivas com o foco em sincronismo e interação entre partes de uma aplicação. Nós propomos uma nova linguagem síncrona reativa, de estilo imperativo, cujas primitivas são baseadas em mecanismos de invocação implícita não convencionais. Com este trabalho, nossa intenção é unificar as características essenciais de linguagens reativas, mantendo a conveniência de um estilo imperativo de programação. Um escalonador reativo é responsável pela execução de reatores, nossas unidades de processamento, baseado nas relações de dependência entre eles construídas dinamicamente. Nessas relações, o fim da execução de um reator inicia a execução de outro. Além disso, um reator pode suspender sua execução de modo a esperar pelo término de outro reator. Reatores executam de acordo com a *hipótese de sincronismo*, mantendo a aplicação sempre pronta para reagir. Nossa linguagem provê, além do foco em reatividade, programação por *dataflow*, execução sequencial imperativa e uso determinístico de memória compartilhada.

Palavras–chave

Linguagens de Programação. Programação Reativa.

Abstract

Sant'Anna, Francisco; Ierusalimschy, Roberto. **A Synchronous Reactive Language based on Implicit Invocation**. Rio de Janeiro, 2009. 65p. MsC Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The reactive programming paradigm covers a wide range of applications, such as games and multimedia systems. Mainstream languages neglect reactive programming, lacking language-level primitives that focus on synchronism and interactions within application parts. We propose a new reactive synchronous language, with an imperative style, whose primitives are based on unconventional implicit invocation mechanisms. With this work, we intend to unite the essential features of reactive languages while keeping a convenient imperative style of programming. A reactive scheduler is responsible for executing reactors, our processing units, based on dependency relations between them built dynamically. Our language provides dataflow programming, sequential imperative execution, and deterministic use of shared-memory.

Keywords

Programming Languages. Reactive Programming.

Contents

1 Introduction **10**

2 Concurrent Systems **13**
2.1 Asynchronous Concurrency 13
2.2 Synchronous Concurrency 15
2.3 Why Synchronous Concurrency? 17

3 Synchronous Reactive Languages **20**
3.1 NCL 20
3.2 FrTime 22
3.3 Esterel 24
3.4 Event-driven programming 26

4 The Proposed Language **29**
4.1 Reactors 30
4.2 The Reactive Scheduler 32
4.3 Cycles and Glitches 35
4.4 Dataflow Programming 37
4.5 Determinism 39
4.6 Language Primitives 42
4.7 Implicit Invocation Roots 44

5 The LuaGravity Implementation **45**
5.1 Reactors 45
5.2 Organisms 47
5.3 Behaviors 49
5.4 Examples 51
5.5 Runtime System 56

6 Conclusion **59**

Bibliography **62**

List of Figures

2.1 Implementation schemes for synchronous systems 16

3.1 NCL State Machine 21
3.2 Graph for expression `(< seconds (+ 1 seconds))` 24

4.1 Abstract view of a program 29
4.2 Reactor State Machine 32
4.3 Graph for the introductory example 33
4.4 Graph derived from SPAWN 34
4.5 Reactor separated in two nodes 35
4.6 Graph subject to glitches 36
4.7 Graph to illustrate concurrent reactors 39
4.8 Graph with concurrent reactors calling an actuator 40

List of Tables

2.1 Concurrency models comparison 18

1
Introduction

In concurrent applications like games, program control is mostly guided by the constant interactions between their entities. Entities may be internal to the application, such as game characters, monsters, and the scenario; and also external (or environmental), such as keyboard, network, and also time.

We may consider that such interactions have a reactive nature, that is, they are defined by cause/effect rules: a triggered action in one entity causes a reaction into another. As an example, a keyboard press causes a character movement, which in turn may be sensed by a monster that starts chasing the character. Interaction, dependency, and reactivity are related concepts in this scope: interactions are specified by dependency rules, bringing a reactive nature to the application.

Games also have constraints on how they should be permanently synchronized with the environment, and are examples of real-time systems. Being a real-time system doesn't imply having critical or high-performance requirements. It implies, however, that operations not performed within a short bounded time are considered useless or even wrong. For instance, consider the annoyance of a game character that does not move immediately following user input, or animations running in a slow frame rate.

This work focuses on design principles, at language level, for applications with real-time constraints that are also subject to a high degree of interaction and dependency, not only with the environment, but also within their concurrent entities. We refer to this kind of application as *reactive systems*. Examples of such systems are, besides games, multimedia, windowing, and simulation systems.

Concurrency is usually centered on the idea of independent processes[1] that, while running, occasionally communicate through message passing or shared memory. Each process is in charge of its own control flow, and chooses when and how to synchronize with the outside world. This way, each process has a very own view of the surrounding environment, not necessarily consistent

[1]Process as the general concept of a sequence of computations running in a computer processor, not restricted to the same term used in operating systems and some languages.

with the rest of the system, as information may change a millisecond after being exchanged. We consider that process-centric systems follow an asynchronous model of concurrency.

Reactive systems, however, are usually programmed with control flow inverted, being guided by the environment and internal interdependencies. Instead of time-consuming processes as execution units, reactive systems are composed of (ideally) zero-delay reactions that sit back waiting for changes in the environment. As soon as stimulus is sensed, the system enters in a reaction chain to answer the environment. The faster the system reacts, the more it will be real-time compliant. Differently from processes, reactions share a global consensus about the surrounding environment due to the inherent synchronization with it. We consider that environment-centric systems follow a synchronous model of concurrency.

A common approach for programming with inversion of control using multipurpose languages is to use event-driven techniques [Meyer], wherein event handlers represent the zero-delay reactions of synchronous concurrency. However, this approach is too verbose, as the reactive logic demands the definition of large amounts of events and handlers. Even worse, sequential program flow is usually broken in a ”callback soup”, where several small code chunks access the same data. The lack of a context in callbacks (i.e. local stack) turns the understanding and maintenance of source code a challenge.

Writing sequential code is a feature most programmers would not like to renounce to, even when programming reactive systems. Asynchronous processes do offer sequential control flow, but, as mentioned, are not under control of the environment, demanding extra synchronization efforts. An effective alternative is to use synchronous control abstractions like *continuations*, which also offer sequential control flow. However, continuations require the notion of cooperation, rather than reactivity, between them.

The best world seems to reside in a language that combines the reactivity and loose coupling found in implicit invocation techniques (such as event-driven programming), with the sequential execution of continuations. This language should, however, eliminate the verbosity of implicit invocation and seamless integrate both. Unfortunately, a language with such requirements is not clearly identifiable at the present moment.

We propose a new abstract language for programming reactive systems, featuring continuation-like control abstractions as execution units that we call *reactors*. Reactors may be dynamically linked in cause/effect relations, so that one reactor automatically triggers its dependencies on termination. They may also be suspended to wait for other reactors to terminate. These features provide reactivity while allowing code to be written sequentially. Just like event handlers, reactors are demanded to return (or suspend) fast in order to keep system's real-time constraints.

A program in our language is then a collection of interconnected reactors waiting for environmental stimulus to react. A reactive scheduler is responsible for the dynamics of the system. Such scheduler starts and resumes reactors based on the dependency graph that evolves during runtime. As a proof of concept we present *LuaGravity*, a Lua implementation of the proposed language.

This document is organized as follows.

Chapter 2 reviews and compares asynchronous and synchronous classes of concurrency models, arguing that reactive systems fit better under the latter.

Narrowing the research to the synchronous realm, Chapter 3 presents some related work on synchronous reactive languages. Three languages are of special interest, as they served as the main inspirations for this work. NCL [LFGS06] is a declarative multimedia language with a reactive behavior. FrTime [Cooper04] is a result of recent research on *functional reactive programming* [Zhanyoung]. Esterel [Boussinot] has an imperative style and was one of the first reactive languages to appear back in early 80s. To finish the chapter, we evaluate the design of event-driven systems in traditional languages, illustrating how the studied synchronous languages compare.

Chapter 4 describes our proposed abstract language. We present its reactive scheduling policy based on implicit invocation, the concept of *reactors*, and how dataflow programming is achieved with the language's imperative features. We also discuss how the language deals with concurrent access to shared memory in a predictable way.

Chapter 5 presents LuaGravity, an implementation of the proposed language on top of the Lua language. We discuss how we extended Lua with reactivity primitives, and provided some abstractions built over it, such as *lifts* and *behaviors* brought from functional reactive programming.

Chapter 6 presents open issues and gives directions on how they could be addressed on future work. We conclude our research enumerating the contributions with the proposed abstract language and implementation.

2
Concurrent Systems

Concurrent systems are composed of interacting entities, running for an indefinitely period. Concurrency happens not only because entities run in parallel, but mainly because they communicate and compete for shared resources.

As system entities must be concretely written in a language and executed somehow, we need to define precisely what they really are and how they interact to each other. Instead of going through how each possible language arranges its concurrency subsystem, we present the main models used for concurrency.

Two concepts are closely related to interactions in concurrent systems and need to be distinguished: *communication* and *synchronization*. By communicating, entities exchange data, allowing them to know about each other states and take further actions. By synchronizing, entities coordinate their control flows at specific points, allowing them to progress together or isolated at certain sections of code. Usually, communication takes place in synchronization points.

2.1 Asynchronous Concurrency

As briefly described in the Introduction, asynchronous systems are composed of independent processes that are in charge of their control flow and synchronize at their will. Asynchronous processes, while performing computations, are blind to the surrounding system, that is, independently of what happens outside, they just keep running. The decision to synchronize is internal to each process, and not enforced by the environment. For this reason, we consider asynchronous systems to be process-centric systems. We use process as a general term, and depending on the actual language, they might be known as threads, actors, tasks and even processes.

Another fundamental characteristic of asynchronous systems is their inherent non-deterministic behavior [Lee]. From the perspective of a CPU, the composition of concurrent processes may be seen as the interleaving of their statements in a single process. First, it is impossible to predict the order in which statements are interleaved, as process scheduling timing is non-deterministic. Second, for the same reason, each time a concurrent system restarts, a different scheduling probably takes place. The lack of enforced synchronization between processes makes impossible to ensure a deterministic behavior.

Communication between asynchronous processes is always considered to take time - it is unlikely that both ends will try to communicate at the exactly same time. This characteristic leads to difficulty in achieving global consensus over information, as data obtained by the receiving end can reflect a past state in the sending end. Delayed communication also introduces the need for *buffers*, yet another concern to deal with (e.g. overflow and underflow).

Two asynchronous concurrency models are widely known: *shared-memory* and *message-passing*.

By far, the most popular concurrency model is shared-memory, being used in mainstream languages like Java [Lea], C/C++ (with *pthreads* [Nichols]) and C# [Birrel]. The idea of sharing memory is simple and tempting, but underestimates the importance and difficulty of coordinating access to the memory.

By using shared-memory, communication between processes (known as *threads* in this context) is almost implicit: reading and writing to a shared address is roughly the same as reading and writing to a local one. Synchronization between threads is usually achieved by serializing access to *critical sections* of code (where shared memory is manipulated) through primitives like *mutexes* and *monitors*. Nonetheless, threads are not required to synchronize when willing to communicate, and not always the programmer identifies that a communication that needs synchronization is taking place, leading to the frequent *race-condition* bugs. The use of monitors [Hoare] centralizes access to shared resources and is considered good practice.

In the message-passing model, communication and synchronization are integrated, with the same language primitive being used for both. A process willing to communicate must synchronize with its counterpart, that is, a message sent to a process must match a receiving in that process.

In shared-memory concurrency, synchronization is via mutual exclusion; communication is implicit and through side effects. In message-passing concurrency, synchronization makes processes to meet; communication is explicit, side effects free, and always in synchronization points.

Regardless of the differences, the models are considered to be equivalent [Lauer], being a matter of style and beliefs to choose one or another. Message-passing is considered by the academy a safer model as much as it is not used by the industry, where shared-memory is the rule.

2.2 Synchronous Concurrency

In synchronous systems, the leader of interactions is the environment, and internal entities must execute at its pace, in permanent synchrony. The idea of independent processes, each one interacting at their will, is discarded in this model. The description of synchronous concurrency is usually attached to its adoption in reactive systems, where the environment plays the principal role.

The *synchronous hypothesis* [Butucaru] is the key concept in which synchronous systems rely on. It states that [Berry92]:

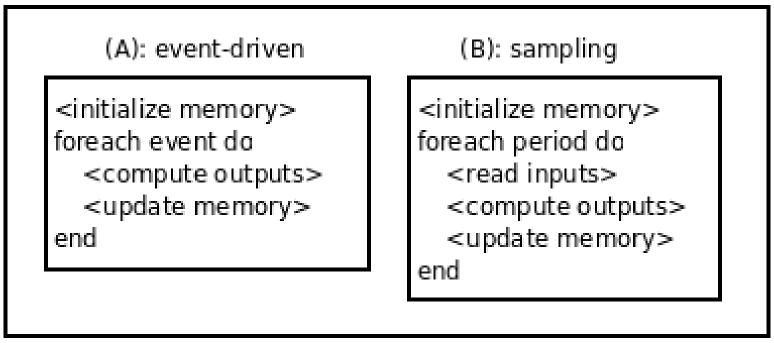
> Each reaction *(in the system)* is assumed to be *instantaneous* - and therefore atomic in any possible sense. Synchrony amounts to saying that the underlying execution machine *takes no time* to execute the operations involved in instruction sequencing, process handling, inter-process communication, and basic data handling (e.g., additions). To "take no time" has to be understood in a very strong sense. First, a reaction takes no time *with respect to the external environment*, which remains invariant during it. Second, each sub-process also takes no time *with respect to other sub-process*; sub-processes react instantly to each other. In synchronous languages, inter-process communication is done by *instantly broadcasting events*; all processes therefore share the same vision of their environment and of each other.

Throughout the text we will the terms *instantaneous*, *immediately*, *zero-delay*, etc. interchangeably in the sense of the synchronous hypothesis.

Figure 2.1 shows two common implementation schemes for synchronous systems. [Halbwachs98]

In the first scheme, a change in the environment is described as an input event. When an event is triggered, the *foreach loop* is awakened and its body executed, updating memory and yielding output. Each input event can be seen as a logical instant, in which all system parts must react synchronously before going to next instant. During a loop iteration, the environment is invariant (possibly buffering incoming input events to be processed further). Here, time is modeled as a sequence of discrete input events.

In the second scheme, an instant is a predefined "physical" time interval, where, at each instant, the environment is polled for changes. Then, such



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

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

Figure 2.1: Implementation schemes for synchronous systems

sampling scheme proceeds like the event-driven approach. This form is similar to how digital circuits are designed, where signals propagate from input to output gates in a single clock tick.

Both schemes are compliant with the synchronous hypothesis, where input and resulting output happen at the same time, at least with this notion of time as a sequence of discrete events or clock ticks.

Reactive behavior, instead of preemptive multitasking, guarantees a deterministic execution to synchronous systems. Given the same input configuration, the system will always yield the same output. Again, the notion of time is a sequence of external events, and not something per-process specific. The sole order in which the environment generates inputs to the system is the source of non-determinism (for instance, it is impossible to predict a user interaction or the network latency).

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.

The first scheme of implementation resembles the *main-loops* of event-driven programming in traditional languages. In fact, event-driven programming is the commonest way of programming synchronous systems. Among the common problems with this pattern are programming with control inversion and the lack of stack for managing state [von Behren]. We can consider that event-driven programming is a low-level way of programming synchronous systems, not much different than dealing with interrupt handling in hardware.

”Higher-level” synchronous languages do exist; however, they are restricted to academic and very specific industrial niches. Two high-level styles for programming synchronous systems have arisen.

In the *dataflow style*, data is in the form of signals, which are values varying over time. Programs are networks of operators connecting signals. Control is implicit - at each clock tick, the values of signals are updated based on their equations. A delay operator is provided to support the use of signals' past values in their own definitions. The dataflow style provides a declarative way for programming reactive systems, and is effective to describe dependency between data. Examples of languages supporting this style are Lustre [Halbwachs91] and Signal [Le Guernic], which are used in applications such as signal processing and circuit modeling.

The example below shows a node (Lustre's abstraction for reuse) defining a generic counter and two instances of it:

```
node COUNTER (init, inc: int; reset: bool) returns (n: int);
    let n = init ->
        if reset then
            init
        else
            pre(n)+inc;
    tel;
even = COUNT(0, 2, false);        //=> 0, 2, 4, 6, 8, 10, ...
mod5 = COUNT(0, 1, pre(mod5=4));  //=> 0, 1, 2, 3, 4, 0, ...
```

In the *control flow style*, as its name suggests, control is commanded explicitly. Statecharts [Harrel] uses a generalization of finite state machines to specify systems graphically. Esterel [Boussinot], described in Section 3.3, provides an imperative style with control abstractions like sequential and parallel compositions, loops and preemption.

After initial research on synchronous languages, dating back to early 80s, the advent of *functional reactive programming* [Zhanyoung] has stimulated new research on this field. A language resulting from recent work, FrTime [Cooper04], is described in Chapter 3.2.

2.3 Why Synchronous Concurrency?

We defend that the synchronous approach is more appropriate for programming reactive systems. Table 2.1 summarizes the main differences between the synchronous and asynchronous classes of concurrency.

Reactive systems are mainly guided by the environment. Taking a windowing system as an example, most functionality is derived from user interaction: mouse clicks to select options, key combinations to trigger shortcuts,

Table 2.1: Concurrency models comparison

	Synchronous	Asynchronous
Control	environment	process
Scheduling	reactive	preemptive
Synchronization	implicit permanent	explicit occasional
Communication	instantaneous broadcast	delayed addressed (in message passing) side effects (in shared-memory)
Determinism	deterministic	non-deterministic

keyboard input to edit text, and so on. Asynchronous processes in this context are seen more as an operating system facility to allow multiple applications to run in parallel (hardly interacting).

Synchronization is a permanent characteristic of reactive systems, not a desired feature. In a multimedia application, subtitles must be always synchronized with its correspondent video. Game animations must also be synchronized with a common time base. In these examples, the physical time, as elapses, is the continuous environmental stimulus that makes the system to react synchronously, repositioning animation objects, and advancing subtitles.

Broadcast communication eases the construction of relationships, as they can be decoupled from its actual participants. For simulation software, new agents may be added with little effort, as existing agents need not to be changed in order to communicate with new ones.

Instantaneous communication and zero-delay reactivity, in contrast to delayed communication and preemptive scheduling, are simpler ways to reason about a system behavior.

Finally, non-determinism is generally an undesired feature. It should be added only carefully in specific points or where it is inevitable.

Instead of completely refusing asynchronous concurrency, we argue that there are uses in which they are the right/only answer. CPU intensive algorithms that seldom interact have nothing to gain with synchronization. Network communications are also incompatible with the synchronous approach, as instantaneous communication is impossible, and broadcast communication, rather expensive. In a synchronous world, such operations must be wrapped into asynchronous calls.

In [Berry00], Berry compares the concurrency models, making an analogy with real world physics. He pairs the synchronous model with Newtonian mechanics, and asynchronous with Einstein's relativity theory. He argues that

Newtonian physics is good enough to model most problems, yet much simpler to reason and implement.

3
Synchronous Reactive Languages

After discouraging asynchronous concurrency for designing reactive systems, we dig into existing synchronous alternatives.

We start the chapter investigating three synchronous languages that served as inspirations and references for this work. Each of these languages has features that we consider essential for a reactive language. NCL [LFGS06], although not historically linked to research on synchronous reactive languages, is a declarative multimedia language with reactive behavior. FrTime [Cooper04] is a result of more recent research on *functional reactive programming* [Zhanyoung]. Esterel [Boussinot] has an imperative style and was one of the first reactive languages to appear back in the early 80s.

We finish the chapter reviewing the event-driven programming paradigm, which shares similarities with the presented languages. We classify design characteristics to be considered when adopting implicit invocation mechanisms in conventional languages, and point out how such characteristics were adopted in the studied synchronous languages.

3.1 NCL

NCL (Nested Context Language) [LFGS06] is an XML application language based on the Nested Context Model (NCM) [LFGS05], a model for hypermedia document specification, with temporal and spatial synchronization relationships among media objects.

The middleware Ginga [LFGS07], part of the Brazilian standard for digital TV, adopted NCL as its main authoring language for writing interactive applications.

An NCL document only specifies how media objects relate on time and space, not carrying actual media contents. As a glue language, NCL does not restrict or define what kinds of media are supported, a role delegated to plug-ins.

The application below exhibits an image in parallel with a video playback:

```
<media id="myvideo" src="video.mpg"/>
```

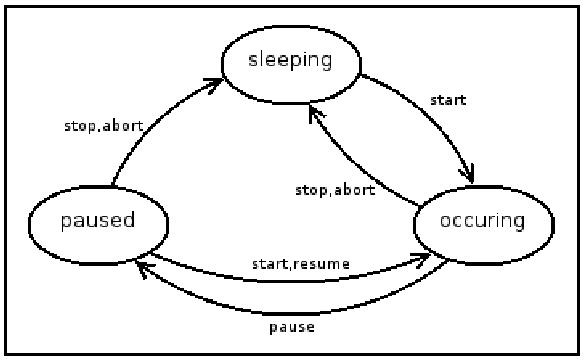


Figure 3.1: NCL State Machine

```
<media id="myimage" src="image.jpg"/>
<link>
       <bind role="onBegin" component="myvideo"/>
       <bind role="start"   component="myimage"/>
</link>
<link>
       <bind role="onEnd"   component="myvideo"/>
       <bind role="stop"    component="myimage"/>
</link>
```

The main concepts of NCL, already present in the above example, are *nodes* and *links*.

Nodes come in two flavors, *composite nodes* and *content nodes*. Composite nodes are collection of nodes, while content nodes (the `<media>` tag in the example) represent actual media objects. Each node has an associated presentation state machine that controls its exhibition. Figure 3.1 shows the existing states and transitions in the machine.

The link is the reactivity primitive of NCL, connecting conditions to actions in media nodes. The `<bind>` tags are used to specify the conditions and actions in a link, where their `component` parameter defines the node to be related, and the `role` parameter indicates the condition/action. As shown in the example, an action role uses the transition name in a node state machine (*start*, *stop*), while a condition role uses a special name for the transition being used as condition (for example, *onBegin* is related to the *start* transition, while *onEnd*, to the *stop* transition).

The `<context>` tag of NCL groups nodes and links in a common container. Just like content nodes, contexts also have presentation state machines, and may be synchronized in links likewise. A *start* in a composite node activates its internal links and starts the media node defined as its entry port. A *stop* transition breaks all internal links, and stops the media objects currently running. As an illustrative example, suppose a level in a game is represented as a context node with enemies, items, timers, and relations between them. A *stop* in the level context would stop all entities and relations automatically, serving as a straightforward bookkeeper. Composite nodes of NCL are powerful abstractions and unlikely to exist in asynchronous languages, where stopping a thread is considered unsafe. [Sun Microsystems]

NCL shows an uncommon characteristic as it *symmetrically* maps transitions to transitions in relationships between media nodes. Transitions, more than representing the occurrence of events, also carry actual actions (for example, the *start* transition really starts media playback), justifying such symmetric mapping in relationships. Event-driven systems usually map events to function calls (callbacks), leading to more verbosity, as extra steps to define and post events are required.

3.2 FrTime

FrTime [Cooper08, Cooper04, Cooper06] is a functional reactive language built on top of Scheme. Research on functional reactive programming (FRP) emerged in the last 10 years [Zhanyoung, Nilsson, Elliot], following initial work on dataflow languages in the early eighties, such as Lucid [Ashcroft], Lustre [Halbwachs91] and Signal [Le Guernic].

The key concept of FRP is the *signal*, or time-varying value, which comes in two forms, *events* and *behaviors*.

Events are not much unlike those of traditional event-driven programming, and are used to model external interactions, such as user input. In FrTime, events are represented as streams, where modified versions of traditional list-processing functions such as *map* and *filter* can be applied.

Behaviors, the other form of signals, carry dependency among themselves, and are the most exciting feature of FRP and dataflow languages. For instance, if `a` and `b` are behaviors, the result of writing the expression `a + b` is another behavior that is updated when `a` or `b` is changed, recalculating the initial assignment. This way, the result of the expression will always hold the sum of `a` and `b`.

FrTime offers several system behaviors to be used in programs. One of them is `seconds`, holding the current elapsed number of seconds at every moment. The simple evaluation of `seconds` in the DrScheme interpreter [Findler] exhibits the elapsing number on screen. Another example is `mouse-pos`, holding the current mouse position. The evaluation of the program below exhibits a blue circle that "magically" follows the mouse position:

```
(display-shapes
  (list (make-circle mouse-pos 20 "blue")))
```

In the example, `display-shapes` receives a list of shapes to show on screen. `make-circle` receives a container with x and y coordinates (`mouse-pos`, in this example), a radius, and a color. When the mouse moves, `mouse-pos` is updated, making the circle to follow the mouse.

Writing this program using event handlers would demand the use of callbacks and explicit state control (i.e. side effects). Besides requiring more lines, the definition would not be as self-contained as that of the previous example.

Another primitive found in FRP languages, including FrTime, is the *integral* [Cooper04]. This primitive is essential for any kind of animations. The following example shows a circle crossing the screen in both axis with the speed of one pixel per second.

```
(display-shapes
  (list (make-circle
          (make-posn (integral 1) (integral 1))
          20
          "blue")))
```

In the example, `make-posn` creates a container for x and y coordinates. By using integral, the coordinates are incremented one pixel each second.

When using behaviors in expressions with function applications or operators, it is expected that the result become also a behavior. However, functions and operators in conventional languages like Scheme are not prepared to accept behaviors as parameters. It is necessary, then, to modify each of these operations to work with behaviors, a process known as *lifting*. FrTime modifies Scheme in such a way that function/operator lifting is transparent to the programmer.

FrTime employs on its implementation the event-driven approach of Figure 2.1, which Cooper names as *push-driven update mechanism* [Cooper06]:

> Events initiate computation, and changes cause dependent parts of the program to recompute.

A graph is used to keep dependency among behaviors. When a behavior is changed, the graph is traversed to recalculate dependencies. Figure 3.2 shows the graph for the expression `(< seconds (+ 1 seconds))`.

One concern that must be addressed is how to traverse the graph on updates. The previous expression `(< seconds (+ 1 seconds))` is illustrative

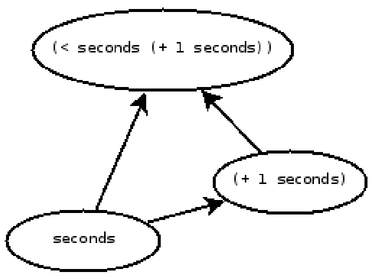


Figure 3.2: Graph for expression `(< seconds (+ 1 seconds))`

for this purpose. As an invariant, the value for the expression is always *true*, as `seconds` is always smaller than its own increment. Looking at the graph in Figure 3.2, there are two edges leaving the `seconds` node, and one of them is updated first. When `seconds` changes, if the whole expression is updated before the subexpression `(+ 1 seconds)` is, the invariant will be broken for an infinitesimal time (until the subexpression is updated). To avoid this phenomenon, known as *glitch* [Cooper06], the graph must be traversed in *topological order*.

Another concern is the presence of cycles in the graph, possible when two behaviors depend on each other. The same algorithm used to calculate topological order might be used to detect cycles. Cycles are a recurrent issue in synchronous languages [Cooper06, Berry-Primer], and are worked around with the use of a delay operator, which FrTime also offers.

FrTime considers events and behaviors to be duals [Cooper08], and puts research efforts on behaviors, where most discussion is taken. Events are extended with a functional taste, but are still used in traditional fashion.

3.3 Esterel

Esterel [Berry92, Berry00, Boussinot] is among the oldest synchronous languages. Contemporary languages to Esterel such as Lustre and Signal use a declarative style and are suitable for dataflow applications, as seen in Sections 2.2 and 3.2. Esterel uses the imperative style, and no other popular imperative synchronous language has appeared.

Boussinot [Boussinot] summarizes the motivations behind Esterel as

> Esterel = reactivity + atomicity of reactions + instantaneous broadcast + determinism

These characteristics have been described in Section 2.2.

Esterel designers usually advocate that Esterel programs are similar to their specification [Berry00, Boussinot]. Such claim is exemplified with the following specification:

Emit the output `O` as soon as both the inputs `A` and `B` have been received. Reset the behavior whenever the input `R` is received.

In Esterel, the implementation for this specification is as follows:

```
module ABRO:
    input A, B, R;
    output O;
    loop
        [ await A || await B];
        emit O
    each R
end
```

The program first defines its input and output signals. Then, it enters in a loop that is restarted on each `R` received. The loop body first awaits both `A` and `B`, and then emits `O`.

In Esterel, `||` is the parallel operator, while `;` is the sequencing operator. This way, `await A` and `await B` run in parallel, and `emit O` is only executed after both awaits return. The `await` primitive suspends the running activity until the given signal is emitted somewhere.

The communication units in Esterel are the *signals*[1]. A signal is equivalent to an *event* of event-driven programming, and can be instantly broadcast to the entire application, waking up its listeners. Signals are emitted with the `emit` primitive and caught with `await` and other temporal constructs like `loop-each`. The `emit` command may pass a value along with the signal, as in `emit X(1)`. The value of a signal may be accessed prefixing it with `?`, as in `v := v + ?X`.

Esterel supports a rich set of preemption constructs, used to structure activities in hierarchies. Similar to *contexts* of NCL, they ease the management of activities. The following example, extracted from [Berry00], uses the `every-do-end`, `loop-each`, and `abort-when` constructs:

```
module Runner:
    input Morning, Step, Second, Meter, Lap;
    every Morning do
        abort
            loop
                abort <RunSlowly> when 15 Second;
```

[1]Not to be confused with those of FrTime.

```
                abort
                    every Step do
                        <Jump> || <Breathe>
                    end
                when 100 Meter;
                <FullSpeed>
            each Lap
        when 2 Lap
    end
end
```

Conventional variables are also supported in Esterel, however they cannot be freely shared between concurrent statements. In a statement like `[ v := 1 || v := 2 ]`, the value of `v` would become non-deterministic, a situation that is not acceptable in Esterel's semantics. If a variable is written in any parallel activity, it cannot be read or written elsewhere.

Besides temporal loops [Berry-Primer] and the `await` primitive, statements in Esterel are considered to be instantaneous, following the synchronous hypothesis. Although NCL and FrTime are also compliant with the synchronous hypothesis, the programmer need not to be aware of it, as the zero-delay reactivity is working under the hoods, in the implementation of the language. In Esterel, concurrency is explicit, hence, the programmer has to keep in mind which statements are instantaneous and which take time. Even so, a program like `loop x := x + 1 end` is rejected by the compiler as it constitutes a zero-delay loop.

Among all languages that we have studied during our research, only Esterel achieved our intent, raised in the Introduction, to conciliate sequential execution with reactivity. We consider that sequencing along with the `await` primitive are the most appealing features of Esterel. However, for dataflow programming, as accomplished by FrTime, Esterel is not suitable.

3.4 Event-driven programming

All synchronous languages we have looked during this research have a lot in common with event-driven programming due to instantaneous broadcast communication and reactivity. Also, the vocabulary of concepts and implementation style resembles those of event-driven programming.

Event-driven is the style of programming where procedures are implicitly called on the occurrence of events [Meyer]. The use of event-driven programming in traditional languages appears with several names, such as *publish-subscribe* and *observer* patterns, *delegates* of .NET, etc.

In a running event-driven application, procedures may be registered or unregistered to be called on events. Event generators are not required to know in advance which procedures will be implicitly called. Any generated event may trigger several registered procedures. These characteristics are determinant to cope with the *loose coupling* found in event-driven system's parts.

Nowadays there is a consensus on what is and how to implement event-driven systems. Regardless of the specific framework in use, an event-driven system implementation is composed of the following parts:

- Main loop
- Events
- Event queue
- Event dispatcher
- Event handlers

Usually, only events and handlers are visible for the application programmer.

The event-driven subsystem is normally built on top of conventional languages, which do not offer primitive facilities with events and inversion of control in mind. As a result, programming such systems tends to excess verbosity and complexity. Another resulting inefficiency is the presence, in the same system, of different subsystems or APIs for programming with events. For instance, an application that handles user input, XML parsing with SAX, and network communications, will likely have three different APIs in use. Moreover, each API will have all the machinery listed above reimplemented. This happens because the event paradigm is present in the *library* being used, rather than in the *language*.

Garlan [Garlan] describes the addition of an implicit invocation subsystem to the language ADA. As the paper is anterior to the consolidation of event-driven programming, several non-conventional design choices have been discussed. The following paragraphs enumerate each of them, pointing out current choices in traditional languages, and also in the languages presented in this chapter:

Event definition, or how exactly an event is defined in an application. The paper suggests a *fixed event vocabulary*, *static event declaration*, *dynamic event declaration*, or *no event declarations at all.* Static languages commonly use static declaration, while dynamic languages use strings or symbols to identify events, not requiring explicit declarations. Esterel uses static declarations of signals; FrTime uses no declaration of events; NCL uses fixed events (the pre-defined transitions in presentation state machines).

Event structure, or what kind of information goes along with an event. The paper suggests *raw events*, *fixed parameter list*, *per event type parameter list*, or *parameters by announcement*. Static languages usually allow parameters to vary by event type; Dynamic languages vary parameters per announcement. In Esterel, a signal can hold only a single value; FrTime varies parameters per announcement; In NCL, each event type has a fixed parameter list.

Event binding, that is, how to determine which procedures are called on event announcements. *Static* or *dynamic* bindings are considered. Dynamic binding, where bindings between events and handlers are created in runtime, is the rule nowadays.

Event announcement, or how events are broadcast to an application. The paper suggests using a *single procedure*, *per event procedure*, a *language primitive*, or *implicit announcement*. It's common to use a single procedure to announce events, such as `postEvent()`. Esterel provides the primitive `emit` for this purpose; FrTime uses the single procedure `send-event`; In NCL, event announcement is implicit, on state changes in nodes.

Delivery policy, that is, whether all procedures bounded to an event should be invoked on announcement. The paper suggests *full delivery*, *single delivery*, *parameterized selection*, or *state-based policy*. Full delivery is often used, as it can simulate the other options via proxy procedures.

Concurrency of handlers; pondering if the execution of event handlers should be parallelized. Although it is apparently an implementation issue, parallel execution of event handlers is not possible in conventional event-driven systems due to race conditions in shared state [Zeldovich]. Event-driven programming and multi-core execution have been considered antagonistic. Even though, a known manual technique to overcome this limitation is to give colors to handlers - handlers not sharing colors can be parallelized [Zeldovich].

4
The Proposed Language

In this chapter, we propose a new programming language with the intend to unite the essential features of reactive languages. The language extends a conventional imperative host language with new reactivity primitives.

In our language, a program is a dynamic dependency graph of *reactors* (our execution units) waiting for external changes to react. In the graph, nodes are reactors whose dependency relations are represented by directed edges connecting them. A program runs in accordance with the synchronous hypothesis, given that, from an external stimulus, the program instantly reacts to it.[1]

Figure 4.1 shows an abstract view of a program. The dotted edges characterize a reaction propagation chain in the program, starting from a stimulus from the environment. During propagation, the environment is considered to be immutable.

[1] In a real implementation, instantaneous means being fast enough, in a way that processing time is not noticed.

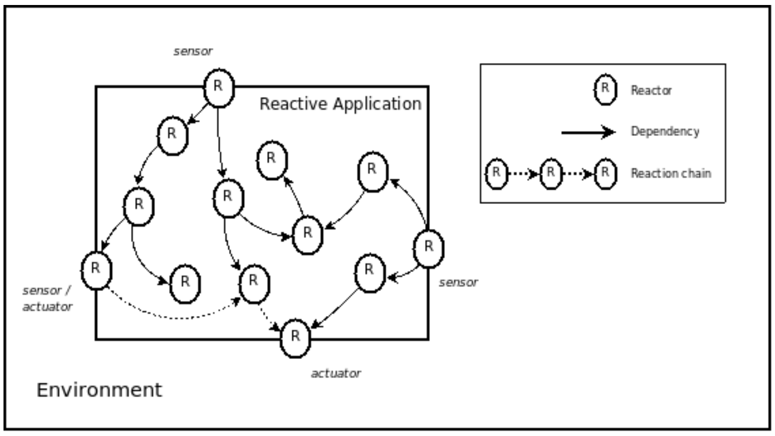


Figure 4.1: Abstract view of a program

4.1 Reactors

The execution unit of our language is known as *reactor*. A reactor is much like a procedure, with control flow, assignments, and conventional features of imperative languages. Unlike procedures, reactors may be linked to each other in cause/effect relations, so that when a reactor terminates, all its dependent reactors are implicitly called. A reactor may also suspend its own execution to wait for a specific reactor to terminate, creating a temporary link between them.

The following list shows the commands that reactors are allowed to perform:

– Create new reactors.

– Start and stop reactors.

– Create and destroy links between reactors.

– Make itself await other reactors.

– Perform conventional imperative statements.

The complete description for each language primitive is presented in Section 4.6. Throughout the text, we use capitalized words to identify language primitives, such as `REACTOR` (a reactor constructor), giving a brief description as they first appear.

Reactors run in accordance with the synchronous hypothesis - we consider that any sequence of commands is atomic and executes instantaneously.

As an introductory example, consider reactors `rA`, `rB`, and `rC`:

```
rA = REACTOR ()
    val = 'a'
END
rB = REACTOR ()
    val = 'b'
END
rC = REACTOR ()
    val = 'c1'
    AWAIT(rB)
    val = 'c2'
END
LINK(rA, rC)
```

When `rA` executes, it sets `val` to *a*. Just after that, `rC` is executed, due to the link from `rA`, setting `val` to *c1*. Then, `rC` awaits `rB`. The value of `val` remains equal to *c1* until `rB` is triggered somewhere. When `rB` is executed, it sets `val` to *b*, and awakes `rC`, which changes `val` to *c2*.

The `AWAIT` call saves the *continuation* of the running reactor before suspending it, keeping the local environment and point of suspension to restore on resume.

Reactors may exchange data on link propagations. In the following code, the *for body* passes values to `rA`, which, in turn, returns their doubles. Due to the link between `rA` and `rB`, the new values returned by `rA` are passed to `rB`, as in a pipeline.

```
rA = REACTOR (v)
    return v*2
END
rB = REACTOR (v)
    print('value '..v)
END
LINK(rA, rB)
for i=1 to 10
    rA(i)
end
--> Output:
--> value 2
--> value 4
--> ...
--> value 20
```

(a) Reactor State Machine

We model a reactor with the state machine shown in Figure 4.2.

In any given moment, from the point of view of the environment, a reactor may be in one of the two states:

`READY`: The reactor is ready to act, either by an explicit spawning, or implicitly, when a reactor it depends on terminates.

`AWAITING`: The reactor is awaiting the termination of other reactors to continue.

The transient state `REACTING` represents the zero delay execution performed by reactors, while the environment remains unchanged:

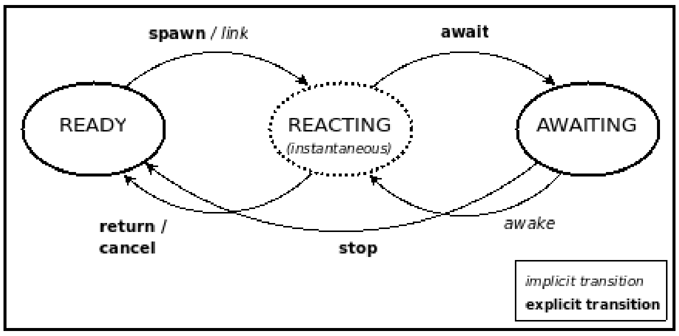


Figure 4.2: Reactor State Machine

`REACTING`: All processing is performed while in this state. The possible commands are described in Section 4.6. The reactor must leave this transient state within an infinitesimal time to keep the system reacting to the environment. The own reactor is responsible for explicitly terminating or suspending itself, transiting to one of the `READY` or `AWAITING` states.

Transitions in italic are performed by the reactive scheduler due to chain propagations (see Section 4.2). Other transitions are explicit, and have the same name of the respective primitive described in Section 4.6.

Invalid transitions in reactor state machines are considered errors. For instance, it is an error to spawn a reactor in the `AWAITING` state.

4.2 The Reactive Scheduler

In our language, a reactor never executes at its own will. The only way for a reactor to execute is as consequence of external stimulus. Also, after starting to execute (i.e. going to the `REACTING` state), a reactor must instantaneously terminate or wait (i.e. going to one of `READY` or `AWAITING` states). Accordingly, a reactor never keeps executing after a stimulus.

It follows that, in the meantime between two external stimulus, the running application is completely idle. The scheduling policy of reactors is only determined by the dependency graph dynamically built during runtime, leading to what we call a *reactive scheduler*.

Starting from an external stimulus, the scheduler traverses the graph running all dependent reactors until it reaches "leaf" reactors. We call this process a *full propagation chain*, which, due to the synchronous hypothesis, takes an infinitesimal time to complete. A full propagation chain is also our definition for an instant within the notion of discrete time of synchronous languages.

The reactivity primitives are responsible for populating the dependency graph with three kinds of edges:

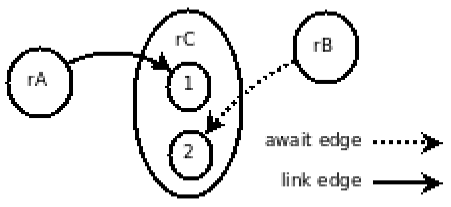


Figure 4.3: Graph for the introductory example

Link edges: Created by `LINK` calls. The edge connects a *source reactor* to a *destiny reactor*, and states that when the source reactor successfully terminates, the destiny reactor is implicitly triggered.

Await edges: Created by `AWAIT` calls. The edge connects the *reactor to await* to the *continuation of the suspended reactor*. Await edges are temporary, as the scheduler removes them as soon as the suspended reactor is awakened or stopped.

Promise edges: Created by calls to promises (see further). The edge connects the *spawned reactor* to the *suspended reactor*. Like await edges, promise edges are also temporary. Unlike await edges, promise edges are also triggered on cancelled reactors (see the `STOP` and `CANCEL` primitives).

Figure 4.3 shows the dependency graph for the introductory example in Section 4.1.

The sub-nodes `1` and `2` in the figure represent the code chunks in the reactor `rC` separated by the call to `AWAIT`.

The `SPAWN` primitive is an exception to our language's reactive scheduling policy through graph traversal, as it explicitly schedules the execution of a given reactor.[2] The `SPAWN` call acts like a fork, instantaneously scheduling the spawned reactor and the continuation of the running reactor to run concurrently. As the spawned reactor may not terminate with a value immediately, the `SPAWN` call returns to the running reactor a *promise* to that value (also known as *future*). In our language, a promise is a function that, when called, awaits the termination of the correspondent spawned reactor. The return value of a promise is the same value returned by the spawned reactor when it terminates.[3]

The following example illustrates the use of `SPAWN`. Figure 4.4 shows the dependency graph for this example.

[2]The reactive property of the language is not broken though, as a `SPAWN` call is only possible as a consequence of an external stimulus anyways.

[3]To simplify the language definition, we chose not to support promises in expressions. However, if the host language supports lambda functions, it is possible to bypass this limitation.

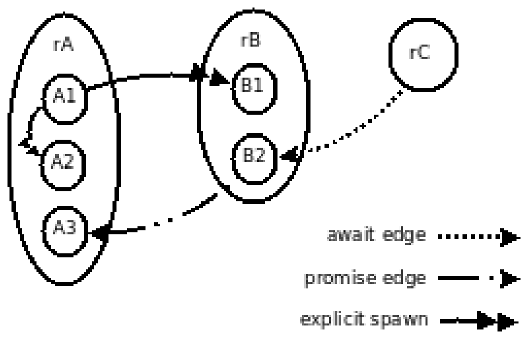


Figure 4.4: Graph derived from SPAWN

```
rA = REACTOR ()
    vA = 'a1'  -- (A1)
    p = SPAWN(rB)
    vA = 'a2'  -- (A2)
    p()
    vA = 'a3'  -- (A3)
END

rB = REACTOR ()
    vB = 'b1'  -- (B1)
    AWAIT(rC)'
    vB = 'b2'  -- (B2)
END

rC = REACTOR ()
    vC = 'c1'
END
```

The reactor `rA` assigns *'a1'* and spawns reactor `rB`. The call to `SPAWN` immediately schedules `rB` and the continuation of `rA` (*chunk A2*) to execute concurrently. The scheduler chooses non-deterministically which one to execute first. When *chunk A2* is executed, it assigns *'a2'* and calls the promise for `rB`, creating a temporary *promise edge* from `rB` to the last continuation of `rA` (*chunk A3*). When `rB` is spawned, it assigns *'b1'* and awaits `rC`, creating the temporary edge from `rC` to the continuation of `rB` (*chunk B2*). The execution of `rC` awakes `rB`, which, in turn, awakes `rA`, and both temporary edges are destroyed.

The execution of the introductory example is deterministic, as only one traversal path is possible. In the previous example, however, the scheduler has two options to proceed on the `SPAWN` call, and any choice is acceptable. In

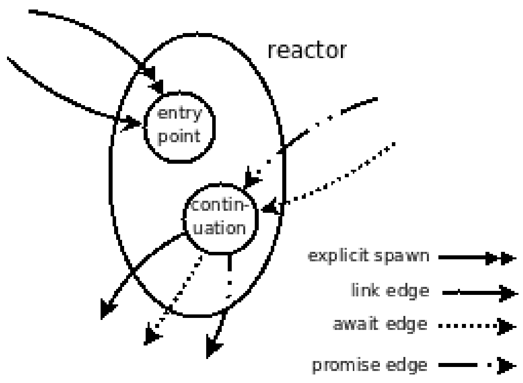


Figure 4.5: Reactor separated in two nodes

Section 4.5 we show that the final effect of a non-deterministic scheduling is either actually *deterministic*, or yields a well formed error, when inevitable.

In the graphs for the examples shown in this section, reactors are expanded in several sub-nodes to represent their continuations. This expanded view seems the correct approach for representing the dependency graph; however it demands syntactic analysis of reactors along with a complex definition of how to split them in continuations.

Our current approach avoids this complexity and takes all decisions at runtime. For instance, we have no way to predict `SPAWN` calls without syntactic analysis; otherwise we could substitute *explicit spawns* in the graphs for a kind of *spawn edges*, unifying the scheduling policy for all primitives.

We actually use only two nodes to represent a reactor, as Figure 4.5 shows. One node represents the reactor entry point, triggered only by *link edges* or `SPAWN` calls. The other node represents the current continuation of the reactor, triggered by *await* or *promise edges*. A continuation may terminate the reactor, triggering outgoing edges of any type. As a reactor can only be waiting in a single internal point at a time, then only one node for continuations is needed.[4]

4.3 Cycles and Glitches

(a) Tight Cycles

A *tight cycle* [Cooper04] happens when a reactor is re-executed during the same propagation chain due to a dependency on itself. For instance, the statement `LINK(rA, rA)` creates a cycle, given that, when `rA` executes, it will trigger itself indefinitely. As a full propagation chain represents a time unit, it

[4]The same argument justifies why we use only one `AWAITING` state per reactor (see Section 4.1(a)).

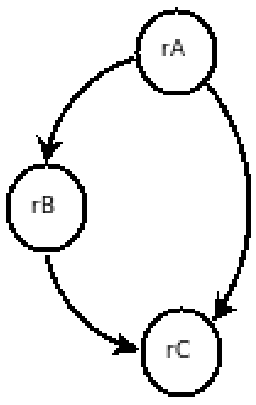


Figure 4.6: Graph subject to glitches

is conceptually wrong to have a reactor executing infinitely in the very same instant.

A path in the dependency graph that passes through *delayed reactors*[5] never characterizes tight cycles. Hence, the statement `LINK(rA, rA)` may actually not create a tight cycle if `rA` is a delayed reactor.

As a workaround to break tight cycles, we provide the `PAUSE` primitive, which suspends the execution of the running reactor for the current propagation cycle, scheduling the reactor to run in the following instant. In Section 4.4, we show a common application of this primitive.

The scheduler detects tight cycles when adding edges to the dependency graph. However, because the `SPAWN` call circumvents the graph structure, the scheduler also needs to check if a reactor is being re-executed during a propagation chain.

(b) Glitches

A *glitch* is an unwanted situation when a reactor is re-executed from *different paths* during a graph traversal (i.e. not depending on itself).

Suppose a program as follows, leading to the graph of Figure 4.6:

```
rA = REACTOR () a = random() END
rB = REACTOR () b = random() END
rC = REACTOR () c = a + b    END
LINK(rA, rB)
LINK(rA, rC)
LINK(rB, rC)
```

[5]Reactors that are suspended before terminating, either by calling `AWAIT`, `PAUSE`, or a promise.

If the scheduler traverses the graph using *depth first search* or *breadth first search*, `rC` will be possibly executed before `rB` and, consequently, before the variable `b` is updated. This way, the variable `c` would evaluate to the sum of the *updated value* of `a` with the *not updated value* of `b`. Only after `rC` is executed again, now after the termination of `rB`, that the variable `c` would hold the correct value.

This example is equivalent to that of glitches in behaviors of FrTime (see Section 3.2). However, different consequences for glitches are possible here; even if `rC` just printed a constant value on screen, the glitch would indeed cause the constant to be erroneously printed twice.

The solution applied to our scheduler is the same taken by FrTime: the graph should be traversed in topological order. With this objective, the scheduler must keep the height of each reactor, which is its maximum distance in edges from a reactor that has no incoming edges.

4.4 Dataflow Programming

We intended to have dataflow support in our language built on top of its imperative primitives. For that matter, we see the `LINK` primitive as an essential feature in the language. In Section 4.1 we showed a simple example of dataflow between two reactors through a link.

Until now, the description and examples for chain propagations showed at most a pair of reactors exchanging data through links. However, as Figure 4.6 shows, more than one source may cause a reactor to be triggered, each passing a different argument. Hence, reactors receive a variable number of arguments:[6]

```
REACTOR (...)  -- variable number of arguments
    -- commands
END
```

(a) Behaviors

Although our language does not provide a primitive for behaviors from functional reactive programming (see Section 3.2), we can implement them on top of the available primitives.

The example below is equivalent to the expression $C = A + B$, where A, B, and C are behaviors.

```
a = 0                        b = 0                        c = 0
```

[6]We assume that the imperative host language that we extend handles variable number of arguments.

```
A = REACTOR (v)         B = REACTOR (v)         C = REACTOR ()
    if v == a then          if v == b then          new = a + b
        CANCEL()                CANCEL()            if new == c then
    end                     end                         CANCEL()
    a = v                   b = v                   end
    RETURN(v)               RETURN(v)               c = new
END                     END                         RETURN(new)
                                                END

LINK(A, C)
LINK(B, C)
A(1)        --> c = 1
B(2)        --> c = 3
```

The current values of variables `a`, `b`, and `c` are normally accessed through their names. However, to set `a` and `b`, their respective reactors `A` and `B` must be called with the value to be assigned. This way, the value of `c` is automatically updated due to the links from `A` and `B` to `C`.

As a trivial optimization, if a reactor receives a new value equal to its variable current value, the reactor terminates with `CANCEL`, not propagating dependent reactors.

As shown in Section 4.3(b), the variable `c` is free from glitches even if `A` and `B` are updated in the same instant.

Undoubtedly, our implementation for behaviors lacks a little of syntactic sugar and a form of encapsulation. We deal with these issues in Section 5.3.

Integral

An *integral* behavior is built similarly to behaviors. Suppose a system reactor `DT` that constantly executes after a full propagation chain, returning the infinitesimal time taken for that. In the code below, the variable `elapsed` counts the elapsed time since the application starts:

```
elapsed = 0
INT1 = REACTOR (v)
    elapsed = elapsed + v
END
LINK(DT, INT1)
```

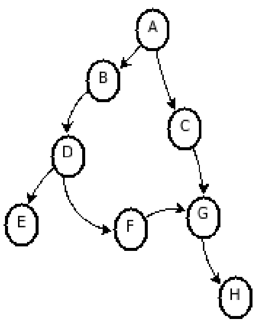


Figure 4.7: Graph to illustrate concurrent reactors

Mutually dependent behaviors

It is common to have behaviors depending on each other. As an example [Cooper06], suppose a GUI control that displays colors in RGB and HSV views, so that changing one view influences the other.

If `RGB` and `HSV` are reactors representing behaviors, simply calling `LINK(RGB, HSV)` and `LINK(HSV, RGB)` is wrong, as it creates a tight cycle.

A simple solution is to introduce a mediator reactor, say `X`, with a `PAUSE` statement, as the following code shows:

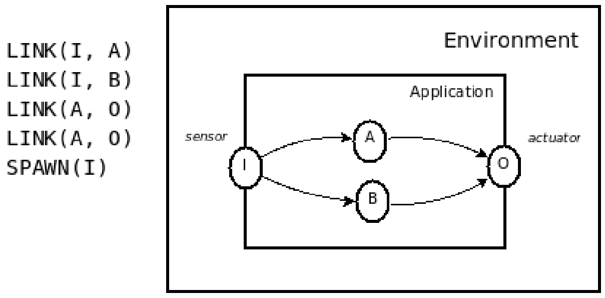
```
X = REACTOR (v)              LINK(RGB, X)
    PAUSE()                  LINK(HSV, X)
    RETURN(v)                LINK(X, RGB)
END                          LINK(X, HSV)
```

4.5 Determinism

In Section 4.2 we showed an example that resulted in the graph of Figure 4.4, and commented that its traversal is non-deterministic. In fact, any call to `SPAWN`, or the termination of a reactor linked to several others, lead to a situation where reactors are scheduled to run concurrently.

Two reactors are considered to be running concurrently if they both execute during the same propagation chain, but not as a consequence of one another. In the graph shown in Figure 4.7, all reactors run as consequence of `A` (which is not concurrent to others). `B` is only concurrent to `C`, as all other reactors depend on `B`; `C` is concurrent to `B`, `D`, `E` and `F`; `G` is only concurrent to `E`; and so on.

We argue that, although our scheduler does not define deterministic criteria to execute concurrent reactors, the final effect is predictable, independently on the order in which they are executed. Furthermore, any semantics given for a deterministic scheduling would be arbitrary, as in such situations, the re-



```
LINK(I, A)
LINK(I, B)
LINK(A, O)
LINK(A, O)
SPAWN(I)
```

Figure 4.8: Graph with concurrent reactors calling an actuator

actors are really conceptually concurrent. Also, imposing a serial execution would disallow the use of real parallel execution in multi-core architectures.

To justify that the final effect of a non-deterministic execution is always predictable, we identify the two situations that can lead to non-deterministic effects, showing how our language deals with them.

The first situation is when executing external actuators (i.e. output interfaces to the surrounding environment), whose side-effects are unpredictable. As Figure 4.8 shows, actuators are represented as reactors, and thus, are executed respecting the topological order of the dependency graph. This way, concurrent reactors willing to invoke the same actuator synchronize before triggering it. The actuator will be called once, deterministically, receiving all reactors' invocation parameters. Each actuator implementation decides whether it accepts or not more than one reactor as teasers, possibly raising an error.

The second situation where non-deterministic effects could appear is when concurrent reactors share memory. In the following code, reactors `rA` and `rB` are spawned concurrently and they both assign to variable `a`. The final value for `a` may be `1` or `2`, depending on which reactor executes last.

```
rA = REACTOR (v) a = v END
rB = REACTOR (v) a = v END
SPAWN(rA, 1)
SPAWN(rB, 2)
```

Our language refuses this kind of concurrent access, raising a *non-deterministic access* error. During a full propagation chain, if a variable is written, it cannot be read or written concurrently.

At each propagation chain, we track access to variables, holding the reactor and mode on each access. If concurrent reactors access the same variable in incompatible mode (i.e. *write vs. read* or *write vs. write*), then the scheduler raises an error.

(a) Intended Non-determinism

Some programs are inherently non-deterministic. Suppose that a given resource, shared among reactors, can only be used exclusively. In the following code, we define a *holder* variable to indicate which reactor is currently holding the shared resource.

```
holder = nil        -- resource is initially free

rFree = REACTOR ()
    holder = nil
END

-- reactors sharing the resource
r[1..N] = REACTOR ()
    -- tries to hold the resource
    while true
        if holder == nil then
            holder = <i>  -- reactor index (1..N)
            break
        end
        AWAIT(rFREE)
    end
    ...             -- safely uses the resource

    SPAWN(rFREE)    -- frees the resource
    ...             -- continues w/o the resource
END

for i=1 to N do
    SPAWN(r[i])
end
```

The *while body* (inside each sharing reactor) is safe, as it tests whether the variable `holder` is *nil* before assigning to it. However, as all reactors are spawned at the same time, concurrent access to the `holder` variable raises a *non-deterministic access* error, as one reactor writes to the variable while others read it.

We provide the `NONDET` primitive, which receives a variable as parameter and allows it to be concurrently accessed. In the example, a previous call to `NONDET(holder)` would allow the program to run without errors.

Keep in mind that non-deterministic variables hold unpredicted values when accessed in incompatible mode by concurrent reactors. Use of non-deterministic variables is unsafe and requires caution. Even so, comparing to shared memory of multi-threading, we believe that our approach is superior. First, the programmer must explicitly turn on such unsafe mechanism, which is restricted to variables set by `NONDET`. Second, mutual exclusion for protecting critical sections of code is not needed, as every code chunk in reactors is already atomic.

4.6 Language Primitives

Follows the comprehensive list of the language primitives:

`reactor <- REACTOR (...) <body> END`

Creates a new reactor with the given body of commands. Returns a reference to the created reactor.

A reactor receives a variable number of arguments (...), with the return values of the source reactors responsible for triggering it.

`promise <- SPAWN (reactor, param)`

Spawns the execution of the given reactor, passing a parameter. The spawned reactor runs concurrently with the continuation of the running reactor.

The call returns a promise, a function that, when executed, awaits the termination of the spawned reactor. The return value of a promise is the value returned by the spawned reactor when it terminates.

`STOP (reactor)`

Terminates the execution of the given reactor, cancelling it.

Reactors linked to (or awaiting) it are not triggered. Associated promises have their return values set to `CANCELLED`.

`RETURN (value)`

Terminates the running reactor successfully with the given `value`.

It will eventually trigger the execution of reactors linked to it. Associated promises have their return values set to `value`.

`CANCEL ()`

Terminates the running reactor, cancelling it.

Reactors linked to (or awaiting) it are not triggered. Associated promises have their return values set to `CANCELLED`.

`break <- LINK (srcReactor, dstReactor)`

Creates a link between `srcReactor` and `dstReactor`. Returns a command that, when executed, breaks the link.

A link determines that as `srcReactor` terminates successfully, `dstReactor` is implicitly triggered. Cancelled or stopped reactors (see above) do not trigger reactors linked to it. The returned value of `srcReactor` is passed to the body of `dstReactor`.

If a source reactor links to several destiny reactors, its termination makes all of them to run concurrently.

`retJ, ..., retK <- AWAIT (r1, r2, ..., rN)`

Makes the running reactor await the successful termination of one of the received reactors. Further actions are not executed until the reactor is awakened.

The call returns the values *retJ,...retK* after being awakened. These values are the return values of the reactors responsible for awaking the suspended reactor.

`PAUSE ()`

Suspends the execution of the running reactor for the current propagation cycle, scheduling the reactor to run in the following instant. May be used to break tight cycles.

`NONDET (variable)`

Allows the received variable to be accessed non-deterministically by concurrent reactors.

Normally, once a variable is written, it cannot be read or written by concurrent reactors.

4.7 Implicit Invocation Roots

In Section 3.4 we enumerated key design choices on how to add an implicit invocation subsystem to a conventional language. Our work can be seen as a more incisive way of achieving this goal.

We now expose the design decisions for our language following the classification used in Section 3.4.

Event Definition: In our language, all reactors are considered to be events, leading to a kind of *implicit event definition* approach.

Event Structure: We follow the *parameters by announcement* approach, where the content of events depends on the returned values of reactors, which are passed to their listeners.

Event Binding: We use dynamic bindings, but, diverging from a traditional *register procedure* for this purpose, we provide a rich set of binding primitives like `LINK`, `AWAIT`, and promises.

Event Announcement: In our language, as each reactor is an event, their simply termination implicitly announces events.

Delivery Policy: We use full delivery of events.

Concurrency of Handlers: We provide a safe deterministic execution for concurrent reactors, allowing the exploration of parallelism.

In the Introduction, we pointed as our main criticisms to implicit invocation mechanisms (i.e. event-driven programming) the excess of verbosity, and lack of local state due to inversion of control.

We consider that implicit event declaration, along with implicit event announcement are fundamental features to decrease the verbosity of event-driven programming.

Also, promises and `AWAIT` calls give support to event bindings in the middle of reactors, allowing sequential execution and local state to become a reality for programming reactive systems.

5 The LuaGravity Implementation

LuaGravity is a set of runtime extensions to the Lua language supporting our proposed abstract language. In addition to implementing the reactive core, LuaGravity also provides higher level primitives built on top of our foundation language, easing the development of reactive applications.

Probably the two most important concepts of our proposed language to be mapped to Lua are *reactors* and *behaviors*, as they appear in virtually every reactive application. Not surprisingly, in LuaGravity, we map reactors and behaviors to the basic concepts of functions and variables of Lua.

Lua [Ierusalimschy06, Ierusalimschy-LuaManual] is a lightweight, extensible, imperative and functional scripting language. For our proof-of-concept implementation, the imperative style, along with the facilities for extending the language, are of special interest. All LuaGravity's extensions to Lua are added during runtime, with semantic modifications to the language, avoiding the need for a parsing phase.

5.1 Reactors

We cannot easily modify the Lua syntax to include a *reactor* reserved keyword. Lua's extensibility is limited to the semantics of existing concepts, not allowing syntactic modifications to the language. For instance, it is possible to redefine the meaning for `x=1`, but not the assignment token, such as in `x:=1`.

Hence, we take advantage of conventional Lua function declarations to create reactors. For instance, instead of creating a function, the following code should create a reactor:

```
function myReactor (p1, p2, ...)
    ...
end
```

In Lua, a function declaration is actually syntactic sugar for an assignment. The following code is equivalent to the previous one:

```
myReactor = function (p1, p2, ...)
    ...
end
```

That is, the variable `myReactor` is assigned a function created in runtime with an anonymous function constructor.

Moreover, global variables, such as `myReactor`, reside in a predefined table `_G`. The following code is equivalent to the previous one:

```
_G.myReactor = function (p1, p2, ...)
    ...
end
```

In order to create reactors from function definitions, we need to modify the semantics for new index operations in the global table to call the following function instead:

```
-- Call this function instead of performing ‘t.k = v‘
--     (_G.myReactor=func, above)
function (t, k, v)
    if type(v) == ’function’ then
        rawset(t, k, reactor_create(v))
    else
        rawset(t, k, v) -- ‘rawset‘ bypasses semantic extensions
    end
end
```

The function `reactor_create` receives the original Lua function and creates a wrapper around it with additional information:

```
function reactor_create (f)
    return {
        body  = f,
        state = ’ready’,
        edges = {},
        ...
    }
end
```

Every reactor is represented by a Lua table, holding its body of commands, current state, outgoing edges, and several other information omitted here for the sake of brevity.

5.2 Organisms

Organisms are the LuaGravity's counterparts to objects of object-oriented languages. Like objects, organisms are categorized in classes, and are used to encapsulate data and operations into a single abstraction. We do not provide a strict definition for organisms; instead, we currently see organisms as a natural abstraction for reactive applications. The main differences to objects are

– Organisms expose reactive variables, instead of properties (or getters & setters).

– Organisms expose reactors, instead of methods.

As the name suggests, reactive variables keep mutual dependencies automatically. They are a specialization of behaviors[1], and are presented in Section 5.3. For now, it is enough to say that reactive variables are used transparently with the Lua syntax for variables. In the following code, `a` and `b` are reactive variables:

```
org.a = 0
org.b = org.a + 1
org.a = 1
print(org.b()) --> 2
```

As is expected for behaviors, the value of `org.b` is recalculated when `org.a` changes. We use the syntax `org.b()` to access the current value of `org.b`.

To fall back to standard fields and methods in LuaGravity, we use identifiers prefixed by underscores (as in `org._var=1`).

Each class of organisms is defined in a separate file. The following example creates an abstract class to represent "visible" objects we want to draw on screen:

```
org.class 'Visible'

function _new (self)
    self.x  = nil
    self.y  = nil
    self.dx = nil
    self.dy = nil
```

[1]Behaviors in the sense of reactive languages, not of OO.

```
    self.visible = false
    self.isVisible = AND(self.visible,
                       AND(self.x,
                         AND(self.y,
                           AND(self.dx,
                                 self.dy))))
end

function _draw (self)
    -- abstract method
    error 'Abstract Method'
end
```

Every class method or reactor receives `self` as its first parameter, representing the organism being manipulated.

In the example, the constructor `_new` creates instance reactive variables for Visible organisms: `x`, `y`, `dx`, and `dy` for their bounds, a boolean `visible` to turn on/off their exhibition, and a boolean `isVisible` to indicate whether the organism is currently visible on screen. `AND` is the lifted version of Lua's `and` operator (whose semantics cannot be redefined). In typical object-oriented languages, such behaviors would need to be written with accessor methods in order to keep dependencies between them. The `_draw` method is used by the graphical subsystem, and each concrete class (Image, Text, etc) must implement it.

The following example instantiates two `Visible` organisms:

```
v1 = Visible {           -- Equivalent to:
    x=10, dx=20,         -- v1 = Visible()
    y=10, dy=20,         -- v1.x=10 ; v1.dx=20; ...
}

v2 = Visible {
    x=v1.x,       dx=v1.dx,
    y=v1.y+v1.dy, dy=v1.dy,
}
```

Note how `v2`'s bounds are defined in terms of `v1`'s. This way, whenever `v1` moves or resizes, `v2` follows it.

5.3 Behaviors

We consider that a straightforward syntax for behaviors is essential for any reactive language. In LuaGravity, behaviors appear in three flavors: reactive variables (known as *cells* in FrTime [Cooper04]), lifted expressions, and integrals & derivatives.

Reactive Variables

Reactive variables are used transparently as normal Lua variables, but have their values updated whenever a dependency changes. In the last section, we showed the use of reactive variables in organisms.

To implement reactive variables in LuaGravity, we take a similar approach to that of reactors: we intercept variable assignments, creating reactive variables instead. As `myVar=x` is equivalent to `_G.myVar=x`, we modify the semantics of `_G`'s new index operation again, now to support reactive variables as well.

Note that the same semantic changes on `_G` are applied to organisms. For LuaGravity, `_G` is actually a kind of global organism. Also, as commented previously, we still allow applications to use conventional Lua variables and functions by prefixing identifiers with underscores.

Lifted Expressions

Expressions involving behaviors are lifted to produce new behaviors. Arithmetic expressions, whose operators Lua allows to redefine, are used transparently in LuaGravity. Other operators have associated lifted functions, requiring Lua programs to be rewritten to behave reactively. We also provide the primitive `L`, which returns a lifted version of a given function.

Follows an example of an expression written in Lua, converted to behave reactively in LuaGravity:

```
-- Non-reactive Lua expression
var = tonumber(a) or b*b

-- Reactive LuaGravity expression
var = OR( L(tonumber)(a), b*b )
```

Integrals and Derivatives

LuaGravity supports the primitives `S` and `D` for creating integrals and derivatives behaviors, respectively [Cooper04]. The example below exemplifies their use:

```
s = s0 + S(v)    -- ‘s‘ is incremented with speed ‘v‘
a = D(v)         -- ‘a‘ holds the variation of speed
```

(a) Behaviors as Organisms

Behaviors are themselves implemented as organisms. The following code shows the implementation of reactive variables as instances of the *VarBehavior* class of organisms, keeping our intent of building behaviors on top of existing mechanisms, and not relying on internal tweaks.

The reactive variable constructor initializes its current value (`_value`), the data source (`_src`) it depends on, and a function to break the dependency with its source (`_brk`):

```
org.class 'VarBehavior'

function _new (self, v)
    self._value = v
    self._src   = nil
    self._brk   = nil
end
```

The reactor `set`, which is never called directly by an application programmer, is used on dependency links between behaviors created on assignments (see further):

```
function set (self, value)
    if self._value == value then
        return CANCEL
    end
    self._value = value
end
```

As an optimization, if the value being set is the same the variable currently holds, the reactor is cancelled (not propagating dependent reactors).

We altered Lua’s semantics so that the assignment `a=b` links the variables’ `set` reactors:

```
-- ‘a = b‘ is transformed into,
a._src = b
a._brk = LINK(b.set, a.set)
a.set(b._value)
```

The function to break the dependency link is kept to be called in case of further assignments.

In our real implementation, `VarBehavior` is actually a subclass of `Behavior`. We have also implemented lifts, integrals and derivatives as subclasses of `Behavior` in a similar way.

5.4 Examples

In this chapter we present two complete reactive applications written with LuaGravity.[2]

(a) The Dining Philosophers

In this example, we propose a solution to the classical synchronization problem of *The Dining Philosophers* [Dijkstra]. A common solution involves mutual exclusion to lock access to shared forks. Besides *deadlock*, a solution should avoid *starvation* and *livelock*, as well. Follows an overview of the application:

```
-- create forks
local forks = {}
for i=1, 5 do
    forks[i] = Fork()
end

-- create philosophers
local phils = {
    Phil { left=fork[1], right=fork[2] },
    Phil { left=fork[2], right=fork[3] },
    Phil { left=fork[3], right=fork[4] },
    Phil { left=fork[4], right=fork[5] },
    Phil { left=fork[5], right=fork[1] },
}
```

[2] We provide video demonstrations for these examples in the website `http://thesynchronousblog.wordpress.com/video-demonstrations/`, along with other sample LuaGravity applications.

```
-- make them live
for _,phil in ipairs(phils) do
    SPAWN(phil.run)
end
```

The code above just creates the forks and philosophers. The core of the application resides in the reactor `run` of the philosopher class:

```
org.class 'Phil'

function run (self)
    local left, right = self._left, self._right
    while true
    do
        -- think for a random time
        AWAIT(math.random(5))

        -- wait for forks
        while not (left:_isFree() and right:_isFree()) do
            AWAIT(left.put, right.put) -- await one of them
        end
        left:_hold(self)
        right:_hold(self)

        -- eat for a random time
        AWAIT(math.random(5))

        -- back to think
        left:put(self)
        right:put(self)
    end
end
```

The reactor `run` is a infinity loop where the philosopher thinks, waits for the forks and eats. The think and eat steps are just an `AWAIT` on a random time between 1 and 5 seconds.[3]

After thinking, the philosopher enters the *inner while*, checking whether his forks are available for use, otherwise he waits for them until succeeding.

[3]LuaGravity accepts numbers as parameters to `AWAIT` in order to wait for the given number in seconds.

Just after leaving the *inner while*, the philosopher acquires his forks in order to eat.

Given the zero-delay execution of reactors, all philosophers are allowed to run at every instant. Hence, our reactive scheduler provides fairness among philosophers.

Note that, in synchronous languages, there's no need for *mutexes* or other means of mutual exclusion. When the forks of a philosopher are free, the execution of the whole *inner while* followed by calls to `_hold` is atomic, and no others philosophers have the chance to acquire forks in the meantime.

After acquiring the forks, the philosopher eats for a random time. Then, he goes back to think, releasing his forks. The call to reactor `put` awakes philosopher's neighbors if they are in their *inner while*.

Recall that we used `SPAWN(phil.run)` to start the execution of philosophers, but `fork:put()` to release a fork. A call to `fork:put()` is equivalent to:

```
local promise = SPAWN(fork.put)
promise()
```

The difference is that, using Lua's function call syntax, the running reactor blocks until the spawned reactor terminates (following the expected semantics for conventional function calls).

The definition for the `Fork` class is straightforward:

```
org.class 'Fork'

function _new (self)
    self.__phil = nil
end

function _isFree (self)
    return (self.__phil == nil)
end

function _hold (self, phil)
    assert(self.__phil == nil)
    self.__phil = phil
end

function put (self, phil)
```

```
    self.__phil = nil
end
```

Variables prefixed by two underscores, such as `__phil`, are allowed non-deterministic access. Indeed, non-determinism is inherent in this example, as two philosophers may attempt to acquire forks at the same time. The lucky one, after calling `_isFree` and succeeding, calls `_hold` and lock the fork. The other one calls `_isFree` and fails. Hence, the variable `__phil` is written by the first philosopher and read by the second one, concurrently, characterizing a non-deterministic access to the variable.

We defined `put` as the only reactor in the `Fork` class to highlight it as the main synchronization mechanism in the application, accounting for the reactivity between the philosophers.

(b) Slide Show

In this example, we create a simple slideshow application. We have a series of images and correspondent captions we want to draw on screen for five seconds each. We allow the user to navigate with left and right keyboard arrows, ignoring the five seconds timeout. Follows the list of images and captions:

```
local list = {
    { file='dscn0066.jpg', desc='Bana Jelacica Square - Zagreb'  },
    { file='dscn0072.jpg', desc='Ljubljanica River - Ljubljana'  },
    { file='dscn0079.jpg', desc='Ljubljana Castle - Ljubljana'   },
    { file='dscn0108.jpg', desc='Heroe\'s Square - Budapest'     },
    { file='dscn0124.jpg', desc='Chain Bridge - Budapest'        },
    { file='dscn0143.jpg', desc='Bratislava Castle - Bratislava' },
    { file='dscn0168.jpg', desc='DevÃn Castle - Bratislava'      },
    { file='dscn0191.jpg', desc='Dancing House - Praha'          },
    { file='dscn0203.jpg', desc='Charles Bridge - Praha'         },
    { file='dscn0434.jpg', desc='Brandenburg Gate - Berlin'      },
}
```

Next, we add a black background to the screen:

```
screen:add(
    Rect {
        _fill = {r=0,g=0,b=0},
        x=0, dx=screen.dx,
        y=0, dy=screen.dy,
    })
```

The `screen` organism interfaces with the computer screen, constantly redrawing a list of `Visible` organisms (as introduced in Section 5.2) built with `screen.add`.

The variable `I` holds the current slide to be shown. The function `go` is called to change the current slide, incrementing or decrementing the value of `I` depending on a parameter:

```
local I = 1               -- starts at first slide
local go = function (i)
    I = I + i
    if I > #list then     -- cycles to the begin
        I = 1
    elseif I < 1 then     -- cycles to the end
        I = #list
    end
end
```

To navigate in the slideshow, we need to create links from key presses to the function `go`. We also create the timeout reactor, to be used further.

```
LINK(keys.RIGHT.press, function() go( 1) end)
LINK(keys.LEFT.press,  function() go(-1) end)
function timeout ()
    AWAIT(5)
    go(1)
end
```

In the slideshow *while body*, we add to the screen the current image and caption, based on the variable `I`. Then we start the `timeout` reactor and wait for its termination or a left/right key press:

```
while true
do
    local t = list[I]
    local img = screen:add(
        Image { t.file,
            x=(screen.dx-500)/2, dx=500,
            y=(screen.dy-380)/2, dy=380,
        })

    local txt = screen:add(
        Text {
            face   = 'vera.ttf',
            text   = '('..I..') '..t.desc,
            _color = {r=255,g=255,b=255},
            x=img.x+10,
            y=img.y+img.dy+10, dy=15,
        })

    SPAWN(timeout)
    AWAIT(keys.RIGHT.press, keys.LEFT.press, timeout)
    STOP(timeout)

    screen:remove(img)
    screen:remove(txt)
end
```

One of the `AWAIT` conditions updates the variable `I` to exhibit the next slide. After the *while body* is awakened, the current image and text are removed from the screen, and the next iteration is started, with `I` already pointing to the next slide to be exhibited.

Note how the access to `I` is safe. Only the function `go` changes its value. Furthermore, `go` can only be called as consequence of a timeout or a key press. As such reactors do not depend on each other, they cannot be triggered during the same propagation chain, and neither run concurrently.

5.5 Runtime System

On startup, the LuaGravity interpreter applies all modifications to Lua, such as customizing the global environment to create reactors from function definitions. It also initializes reactors representing the environment boundaries (sensors and actuators), which are globally accessible. Then, the interpreter creates and starts a reactor from the file passed as argument. In a typical scenario, such reactor creates and relates other reactors, and then awaits a terminating condition. Finally, the program enters in the main event loop, running until the main reactor terminates.

We have implemented LuaGravity following the event-driven scheme for synchronous systems of Figure 2.1. Follows an overview of the LuaGravity interpreter:

```
main = reactor_create(<filename>)
run(main)                               -- runs the main reactor
while main.state ~= 'ready' do
    local reactor, param = nextEvent()
    run(reactor, param)                 -- runs an event reactor
end
```

The main reactor is created from the given filename and executed. The event loop waits for events, which are internally translated to reactors, and executes them. The function `nextEvent` must be customized for the system where LuaGravity is embedded. In GUI systems, it will typically provide key presses, mouse clicks, clock ticks, etc. as environment boundaries.

The core of our language is its reactive scheduler, which is responsible for keeping the dependency between reactors, and execute them respecting some rules.

The dependency graph changes on calls to `LINK`, `AWAIT`, or promises, which add respective edge types to the graph. Each reactor keeps its own list of outgoing edges so that, when terminating, they iterate over the list, scheduling dependent reactors to execute.

Dependent reactors are not triggered on edge propagations, but only scheduled. The scheduler is responsible for actually triggering them, preserving their topological order and checking for inconsistencies. It is possible that a reactor is scheduled (but not yet triggered) more than once during the same reaction chain by different source reactors. That is exactly what avoids glitches in LuaGravity. When actually triggering a reactor, the scheduler passes all source reactors and parameters to it.

A complete `run` in the scheduler takes an infinitesimal amount of time, and accounts for a full propagation chain, representing our notion of an instant:

```
function run (evtReactor, param)
    trigger(evtReactor, param)
    -- iterates over scheduled reactors,
    --  respecting topological order
    while hasNext() do
        -- triggers next reactor,
        --  passing all source reactors and parameters
        trigger(getNext(), ...)
    end
end
```

The function `run` is called after an external stimulus happens (represented by `evtReactor`). While executing, such external reactor might schedule other reactors, and so on. When there are no more reactors scheduled, the function `run` returns.

Reactors bodies cannot execute as normal functions, as they may suspend on calls to `AWAIT` or promises, before terminating. On suspension, we need to capture and keep the current continuation of the running reactor in order to resume it later at the same point. We use Lua *coroutines* [De Moura] for this purpose.

To execute a reactor, the scheduler first identifies whether the reactor must be started or resumed, depending on the edge triggering it. From link edges or explicit spawns the scheduler starts the reactor; from promises or await edges the scheduler awakes it.

If starting a reactor, we first create a coroutine for its body. Then, we resume the coroutine (just created or being awakened), and check whether it has terminated or been suspended. In case the reactor terminates, we schedule its dependent reactors.

All detection of non-deterministic access to variables is done during runtime. This way, LuaGravity needs to intercept every access to normal variables (those prefixed by one underscore) in order to ensure deterministic access.

When accessing a variable, LuaGravity first identifies the running reactor. Then it checks whether the variable being accessed is in a list of variables already accessed by concurrent reactors. If the access mode is incompatible (for example, the reactor is reading a variable that was previously written), an error is thrown. Otherwise the list of accessed variables in the running reactor is updated with the variable being accessed.

We also need to identify which reactors are running concurrently and which are not. A running reactor keeps the stack of reactors that resulted in its own execution; all other running reactors are, hence, running in parallel with it.

All runtime checks lead, of course, to a dramatic performance penalty, which we are not concerned with at this moment. Even so, the performance is acceptable for the examples we have implemented so far.

6 Conclusion

In the Introduction, we described as the focus of this work the design principles, at language level, for reactive applications. As primary requirements, the language should be driven by reactivity while allowing code to be written sequentially.

In Chapter 2 we argued that asynchronous concurrency is not suitable for writing reactive applications, which must be permanently synchronized with the environment.

In Chapter 3 we presented two languages with opposite approaches for reactivity. In one side, Esterel's imperative style is more suitable for control-intensive applications, supporting a parallel operator, a rich set of preemptive constructs, and the `await` primitive. In the other side, FrTime follows the dataflow approach, where control and dependency between data is managed automatically.

In Chapter 4 we proposed a new reactive language that we believe to fulfill the requirements raised in the Introduction.

The first requirement, reactivity, is achieved with implicit invocation mechanisms for our processing units known as *reactors*. Reactors are connected in dependency relationships so that a terminating reactor triggers the execution of its dependent reactors. Besides providing reactivity, our approach strongly decreases the verbosity found in event-driven programming, where events must be explicitly declared and posted to provide reactivity. In our language, the termination of reactors implicitly broadcast themselves as events. Another simplification, compared to event-driven programming, is that events and their handlers are reified as the single concept of reactors.

The second requirement, sequential execution, is achieved by allowing reactors to suspend their own execution and wait for the termination of other reactors. When a condition reactor terminates, the suspended reactor is resumed with local references and point of execution restored.

In Section 5 we presented LuaGravity, a working implementation of our abstract language. LuaGravity extends the Lua language with reactors and behaviors as new primitives. It also introduces the concept of organisms, which can be considered the reactive counterparts of objects. We have developed several fully working reactive applications with LuaGravity.[1]

We want to emphasize the innovative features of the proposed language:

Reactive programs as dependency graphs. An application can be viewed as a graph, whose nodes are reactors, and edges represent the dependency relationships among them. From an external stimulus, a reactive scheduler traverses the graph, executing dependent reactors.

FrTime already uses a graph to represent dependency in expressions containing behaviors. We extended this idea to represent dependency between fragments of reactors (split at suspension points) in an application.

Unification of dataflow and imperative reactivity. Our language follows the imperative paradigm, whose style resembles that of Esterel. Furthermore, we do support dataflow on top the imperative primitives of the language. For this purpose, we see the `LINK` primitive, absent in Esterel, as an important feature in the language.

Determinism with shared memory. Determinism is a feature that most synchronous languages claim to achieve. Concerning access to shared variables, Esterel takes a conservative approach, as variables can only be assigned in a single thread. FrTime either completely prevents the use of shared variables in dataflow expressions or do not take determinism into account with its *benign impurities* [Cooper04].

We have developed a consistent reasoning for safe and deterministic use of shared variables. With the discrete notion of time of synchronous languages, we can detect simultaneous access to shared variables in a consistent way. Still, when non-determinism is inherent to the application, we provide means to explicitly allow non-deterministic access to specific variables.

As future work, an important direction is going towards a more static model for the language. In Section 4.2, we showed how the `SPAWN` primitive circumvents the dependency graph structure. Only with static analysis reactors could be split in several nodes based on `AWAIT`, `SPAWN`, and promises calls. This would allow the scheduling policy to be unified, relying only on graph edges to trigger reactors, and making the semantics of the language more consistent.

[1] We provide source code and video demonstrations for these applications in the website `http://thesynchronousblog.wordpress.com/video-demonstrations/`.

With the need of an additional parsing phase, a derived implementation possibility is to identify access to shared variables at compile time, creating a map of nodes that cannot run concurrently. Then, during runtime, a simple consult to this map would detect non-deterministic access to variables in concurrent reactors.

We are also not satisfied with the current approach for preventing tight cycles. The `PAUSE` primitive, which is also the solution adopted in other synchronous languages, seems an overkill feature, susceptible to misuses by application programmers.

Another possibility for future work is to allow concurrent reactors that do not share variables to run with true parallelism. A challenge is how to control updates to the dependency graph, which remains shared among reactors.

Bibliography

[Dijkstra] E. W. Dijkstra. **Hierarchical ordering of sequential processes**. *Acta Inf.*, 1:115–138, 1971. 5.4(a)

[Findler] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. **DrScheme: A programming environment for Scheme**. *Journal of Functional Programming*, 12:369–388, 2002. 3.2

[Boussinot] F. Boussinot and R. de Simone. **The Esterel language**. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991. 1, 2.2, 3, 3.3

[Berry92] G. Berry and G. Gonthier. **The ESTEREL synchronous programming language: design, semantics, implementation**. *Sci. Comput. Program.*, 19(2):87–152, 1992. 2.2, 3.3

[Berry00] G. Berry. **The foundations of Esterel**. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press. 2.3, 3.3

[Berry-Primer] G. Berry. **The Esterel-V5 language primer**. CMA and Inria, Sophia-Antipolis, France, version 5.10, release 2.0 edition, June 2000. 3.2, 3.3

[Zeldovich] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières and F. Kaashoek. **Multiprocessor support for event-driven programs**. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003. 3.4

[Meyer] B. Meyer. **The power of abstraction, reuse, and simplicity: An object-oriented library for event-driven design**. In *Essays in Memory of Ole-Johan Dahl*, pages 236–271, 2004. 1, 3.4

[Elliot] C. Elliott and P. Hudak. **Functional reactive animation**. In *International Conference on Functional Programming*, Amsterdan, 1997. 3.2

[Zhanyoung] Z. Wan and P. Hudak. **Functional reactive programming from first principles**. *SIGPLAN Not.*, 35(5):242–252, 2000. 1, 2.2, 3, 3.2

[Nilsson] H. Nilsson, A. Courtney and J. Peterson. **Functional reactive programming, continued**. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press. 3.2

[Cooper04] G. Cooper and S. Krishnamurthi. **Frtime: Functional reactive programming in PLT Scheme**. Technical Report CS-03-20, Brown University, April 2004. 1, 2.2, 3, 3.2, 4.3(a), 5.3, 6

[Cooper06] G. H. Cooper and S. Krishnamurthi. **Embedding dynamic dataflow in a call-by-value language**. In *ESOP*, pages 294–308, 2006. 3.2, 3.2, 4.4(a)

[Cooper08] G. H. Cooper. **Integrating dataflow evaluation into a practical higher-order call-by-value language**. PhD thesis, *Brown University*, 2008. 3.2, 3.2

[Garlan] D. Garlan and C. Scott. **Adding implicit invocation to traditional programming languages**. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 447–455, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 3.4

[De Moura] A. L. D. Moura, N. Rodriguez and R. Ierusalimschy. **Coroutines in Lua**. *Journal of Universal Computer Science*, 10:925, 2004. 5.5

[Ierusalimschy06] R. Ierusalimschy. **Programming in lua, second edition**. Lua.Org, 2006. 5

[Ierusalimschy-LuaManual] R. Ierusalimschy, L. H. d. Figueiredo and W. Celes. **Lua 5.1 reference manual**. Lua.Org, 2006. 5

[Ashcroft] E. A. Ashcroft and W. W. Wadge. **Lucid, a nonprocedural language with iteration**. *Commun. ACM*, 20(7):519–526, 1977. 3.2

[Halbwachs91] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. **The synchronous data-flow programming language LUSTRE**. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 2.2, 3.2

[LFGS05] L. F. G. Soares and R. F. Rodrigues. **Nested Context Model 3.0 part 1 - NCM core**. Technical Report 18, Departamento de Informática - PUC-Rio, 2005. 3.1

[LFGS06] L. F. G. Soares and R. F. Rodrigues. **Nested Context Language 3.0 part 8 - NCL digital TV profiles**. Technical Report 35, Departamento de Informática - PUC-Rio, october 2006. 1, 3, 3.1

[LFGS07] L. F. G. Soares, R. F. Rodrigues and M. F. Moreno. **Ginga-NCL: the declarative environment of the Brazilian digital TV system**. *Journal of the Brazilian Computer Society*, 13(1), 2007. 3.1

[Butucaru] D. Potop-Butucaru, R. de Simone and J.-P. Talpin. **The synchronous hypothesis and synchronous languages**. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. 2.2

[Halbwachs98] N. Halbwachs. **Synchronous programming of reactive systems**. In *Computer Aided Verification*, pages 1–16, 1998. 2.2

[Hoare] C. A. R. Hoare. **Monitors: an operating system structuring concept**. *Commun. ACM*, 17(10):549–557, October 1974. 2.1

[Nichols] B. Nichols, D. Buttlar and J. P. Farrell. **Pthreads programming**. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. 2.1

[Lea] D. Lea. **Concurrent programming in java. second edition: Design principles and patterns**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 2.1

[Birrel] A. Birrell. **An introduction to programming with C# threads**. Technical Report TR-2005-68, Microsoft Research Technical Report, May 2005. 2.1

[Le Guernic] M. L. B. Paul Le Guernic, Thierry Gautier and C. L. Maire. **Programming real-time applications with SIGNAL**. *Proceedings of the IEEE*, 79(9):1321–1336, Sep 1991. 2.2, 3.2

[Harrel] D. Harel. **Statecharts: A visual formalism for complex systems**. *Science of Computer Programming*, 8(3):231–274, June 1987. 2.2

[Lauer] H. C. Lauer and R. M. Needham. **On the duality of operating system structures**. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979. 2.1

[Sun Microsystems] Sun-Microsystems. **Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?** http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html, January 1996. 3.1

[von Behren] R. von Behren, J. Condit and E. Brewer. **Why events are a bad idea (for high-concurrency servers)**. In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, page 4, Berkeley, CA, USA, 2003. USENIX Association. 2.2

[Lee] E. A. Lee. **The problem with threads**. *Computer*, 39(5):33–42, 2006.
2.1